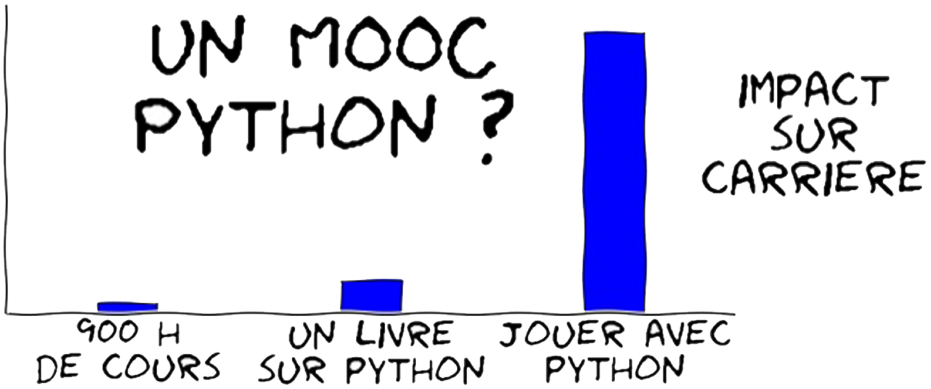
python™ 3

DES FONDAMENTAUX AU CONCEPTS AVANCÉS DU LANGAGE

SESSION 2 - 17 SEPTEMBRE 2018

Thierry PARMENTELAT

Arnaud LEGOUT



https ://www.fun-mooc.fr

Table des matières

Chapitre 1

Introduction au MOOC et aux outils Python

Chapitre 2

Notions de base, premier programme en Python

2.1 Caractères accentués

Ce complément expose quelques bases concernant les caractères accentués, et notamment les précautions à prendre pour pouvoir en insérer dans un programme Python. Nous allons voir que cette question, assez scabreuse, dépasse très largement le cadre de Python.

2.1.1 Complément - niveau basique

Un caractère n'est pas un octet

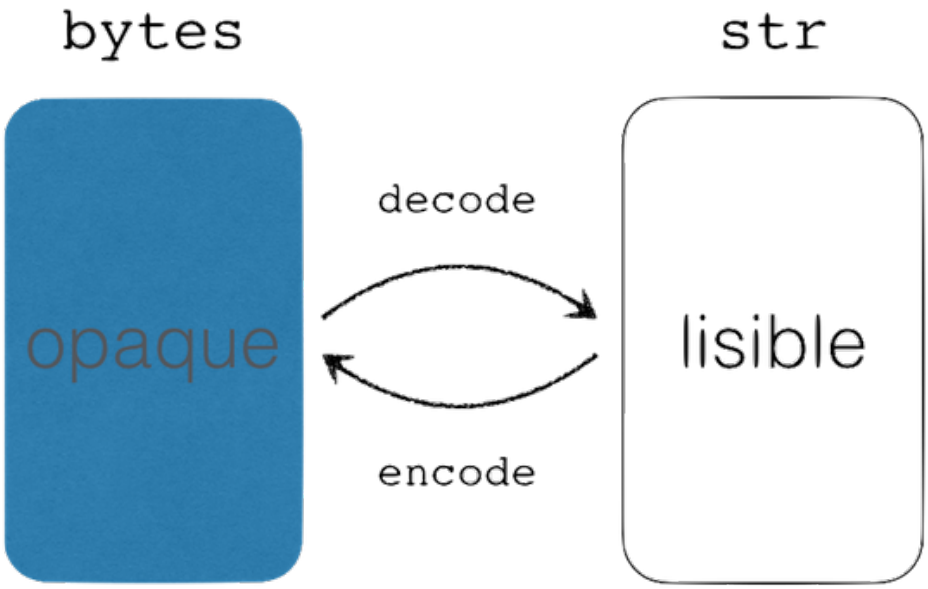
Avec Unicode, on a cassé le modèle *un caractère == un octet*. Aussi en Python 3, lorsqu'il s'agit de manipuler des données provenant de diverses sources de données :

— le type `byte` est approprié si vous voulez charger en mémoire les données binaires brutes, sous forme d'octets donc ;

— le type `str` est approprié pour représenter une chaîne de caractères - qui, à nouveau ne sont pas forcément des octets ;

— on passe de l'un à l'autre de ces types par des opérations d'encodage et décodage, comme illustré ci-dessous ;

— et pour **toutes** les opérations d'encodage et décodage, il est nécessaire de connaître l'encodage utilisé.



les types bytes et str

On peut appeler les méthodes `encode` et `decode` sans préciser l'encodage (dans ce cas Python choisit l'encodage par défaut sur votre système). Cela dit, il est de loin préférable d'être explicite et de choisir son encodage. En cas de doute, il est recommandé de **spécifier explicitement** `utf-8`, qui se généralise au détriment d'encodages anciens comme `cp1242` (Windows) et `iso8859-*`, que de laisser le système hôte choisir pour vous.

Utilisation des accents et autres cédilles

Python 3 supporte Unicode par défaut. Vous pouvez donc, maintenant, utiliser sans

aucun risque des accents ou des cédilles dans vos chaînes de caractères. Il faut cependant faire attention à deux choses :

— Python supporte Unicode, donc tous les caractères du monde, mais les ordinateurs n'ont pas forcément les polices de caractères nécessaires pour afficher ces caractères ;

— Python permet d'utiliser des caractères Unicode pour les noms de variables, mais nous vous recommandons dans toute la mesure du possible d'écrire votre code en anglais, comme c'est le cas pour la quasi-totalité du code que vous serez amenés à utiliser sous forme de bibliothèques. Ceci est particulièrement important pour les noms de lignes et de colonnes dans un dataset afin de faciliter les transferts entre logiciels, la majorité des logiciels n'acceptant pas les accents et cédilles dans les noms de variables.

Ainsi, il faut bien distinguer les chaînes de caractères qui doivent par nature être adaptées au langage des utilisateurs du programme, et le code source qui lui est destiné aux programmeurs et qui doit donc éviter d'utiliser autre chose que de l'anglais.

2.1.2 Complément - niveau intermédiaire

Où peut-on mettre des accents ?

Cela étant dit, si vous devez vraiment mettre des accents dans vos sources, voici ce qu'il faut savoir.

Noms de variables

— S'il n'était **pas possible en Python 2** d'utiliser un caractère accentué dans un **nom de variable** (ou d'un identificateur au sens large), cela est à présent **permis en Python 3** :

```
In [ ]: # pas recommandé, mais autorisé par le langage
        nb_élèves = 12
```

— On peut même utiliser des symboles, comme par exemple

```
In [ ]: from math import cos, pi as Π
        θ = Π / 4
        cos(θ)
```

```
Out[2]: 0.7071067811865476
```

— Je vous recommande toutefois de **ne pas utiliser** cette possibilité, si vous n'êtes pas extrêmement familier avec les caractères Unicode.

— Enfin, pour être exhaustif, sachez que seule une partie des caractères Unicode sont autorisés dans ce cadre, c'est heureux parce que les caractères comme, par exemple, l'espace non-sécable pourraient, s'ils étaient autorisés, être la cause de milliers d'heures de debugging à frustration garantie :)

Pour les curieux, vous pouvez en savoir plus à cet endroit de la documentation officielle (en anglais).

Chaînes de caractères

— Vous pouvez naturellement mettre des accents dans les chaînes de caractères. Cela dit, les données manipulées par un programme proviennent pour l'essentiel de sources externes, comme une base de données ou un formulaire Web, et donc le plus souvent pas directement du code source. Les chaînes de caractères présentes dans du vrai code sont bien souvent limitées à des messages de logging, et le plus souvent d'ailleurs en anglais, donc sans accent.

— Lorsque votre programme doit interagir avec les utilisateurs et qu'il doit donc parler leur langue, c'est une bonne pratique de créer un fichier spécifique, que l'on appelle fichier de ressources, qui contient toutes les chaînes de caractères spécifiques à une langue. Ainsi, la traduction de votre programme consistera à simplement traduire ce fichier de ressources.

```
message = "on peut mettre un caractère accentué dans une chaîne"
```

Commentaires

— Enfin on peut aussi bien sûr mettre dans les commentaires n'importe quel caractère Unicode, et donc notamment des caractères accentués si on choisit malgré tout d'écrire le code en français.

```
# on peut mettre un caractère accentué dans un commentaire
# ainsi que cos(Θ)  ∀ x ∈ ∫f(t)dt ∫∫∫ vous voyez l'idée générale
```

Qu'est-ce qu'un encodage ?

Comme vous le savez, la mémoire - ou le disque - d'un ordinateur ne permet que de stocker des représentations binaires. Il n'y a donc pas de façon "naturelle" de représenter un caractère comme 'A', un guillemet ou un point-virgule.

On utilise pour cela un encodage, par exemple le code `US-ASCII` - http://www.asciitable.com/ - stipule, pour faire simple, qu'un 'A' est représenté par l'octet 65 qui s'écrit en binaire 01000001. Il se trouve qu'il existe plusieurs encodages, bien sûr incompatibles, selon les systèmes et les langues. Vous trouverez plus de détails ci-dessous.

Le point important est que pour pouvoir ouvrir un fichier "proprement", il faut bien entendu disposer du **contenu** du fichier, mais il faut aussi connaître l'**encodage** qui a été utilisé pour l'écrire.

Précautions à prendre pour l'encodage de votre code source

L'encodage ne concerne pas simplement les objets chaîne de caractères, mais également votre code source. **Python 3** considère que votre code source utilise **par défaut l'encodage** `UTF-8`. Nous vous conseillons de conserver cet encodage qui est celui qui vous offrira le plus de flexibilité.

Vous pouvez malgré tout changer l'encodage **de votre code source** en faisant figurer dans vos fichiers, **en première ou deuxième ligne**, une déclaration comme ceci :

```
# -*- coding: <nom_de_l_encodage> -*-
```

ou plus simplement, comme ceci :

```
# coding: <nom_de_l_encodage>
```

Notons que la première option est également interprétée par l'éditeur de texte *Emacs* pour utiliser le même encodage. En dehors de l'utilisation d'Emacs, la deuxième option, plus simple et donc plus pythonique, est à préférer.

Le nom `UTF-8` fait référence à **Unicode** (ou pour être précis, à l'encodage le plus répandu parmi ceux qui sont définis dans la norme Unicode, comme nous le verrons plus bas). Sur

certains systèmes plus anciens vous pourrez être amenés à utiliser un autre encodage. Pour déterminer la valeur à utiliser dans votre cas précis vous pouvez faire dans l'interpréteur interactif :

```
# ceci doit être exécuté sur votre machine
import sys
print(sys.getdefaultencoding())
```

Par exemple avec d'anciennes versions de Windows (en principe de plus en plus rares) vous pouvez être amenés à écrire :

```
# coding: cp1252
```

La syntaxe de la ligne `coding` est précisée dans cette documentation et dans le PEP 263.

Le grand malentendu

Si je vous envoie un fichier contenant du français encodé avec, disons, `ISO/IEC 8859-15` (`Latin-9`) – http://en.wikipedia.org/wiki/ISO/IEC_8859-15 – vous pouvez voir dans la table qu'un caractère '€' va être matérialisé dans mon fichier par un octet '0xA4', soit 164.

Imaginez maintenant que vous essayez d'ouvrir ce même fichier depuis un vieil ordinateur Windows configuré pour le français. Si on ne lui donne aucune indication sur l'encodage, le programme qui va lire ce fichier sur Windows va utiliser l'encodage par défaut du système, c'est-à-dire `CP1252` – http://en.wikipedia.org/wiki/Windows-1252. Comme vous le voyez dans cette table, l'octet '0xA4' correspond au caractère ¤ et c'est ça que vous allez voir à la place de €.

Contrairement à ce qu'on pourrait espérer, ce type de problème ne peut pas se régler en ajoutant une balise `# coding: <nom_de_l_encodage>`, qui n'agit que sur l'encodage utilisé *pour lire le fichier source en question* (celui qui contient la balise).

Pour régler correctement ce type de problème, il vous faut préciser explicitement l'encodage à utiliser pour décoder le fichier. Et donc avoir un moyen fiable de déterminer cet encodage ; ce qui n'est pas toujours aisé d'ailleurs, mais c'est une autre discussion malheureusement. Ce qui signifie que pour être totalement propre, il faut pouvoir préciser explicitement le paramètre `encoding` à l'appel de toutes les méthodes qui sont susceptibles d'en avoir besoin.

Pourquoi ça marche en local ?

Lorsque le producteur (le programme qui écrit le fichier) et le consommateur (le programme qui le lit) tournent dans le même ordinateur, tout fonctionne bien - en général - parce que les deux programmes se ramènent à l'encodage défini comme l'encodage par défaut.

Il y a toutefois une limite, si vous utilisez un Linux configuré de manière minimale, il se peut qu'il utilise par défaut l'encodage `US-ASCII` - voir plus bas - qui étant très ancien ne "connaît" pas un simple é, ni a fortiori €. Pour écrire du français, il faut donc au minimum que l'encodage par défaut de votre ordinateur contienne les caractères français, comme par exemple :

- `ISO 8859-1` (`Latin-1`)
- `ISO 8859-15` (`Latin-9`)
- `UTF-8`

— `CP1252`

À nouveau dans cette liste, il faut clairement préférer UTF-8 lorsque c'est possible.

Un peu d'histoire sur les encodages

Le code `US-ASCII`

Jusque dans les années 1980, les ordinateurs ne parlaient pour l'essentiel que l'anglais. La première vague de standardisation avait créé l'encodage dit `ASCII`, ou encore `US-ASCII` - voir par exemple http://www.asciitable.com, ou en version longue http://en.wikipedia.org/wiki/ASCII.

Le code `US-ASCII` s'étend sur 128 valeurs, soit 7 bits, mais est le plus souvent implémenté sur un octet pour préserver l'alignement, le dernier bit pouvant être utilisé par exemple pour ajouter un code correcteur d'erreur - ce qui à l'époque des modems n'était pas superflu. Bref, la pratique courante était alors de manipuler une chaîne de caractères comme un tableau d'octets.

Les encodages `ISO8859-*` (`Latin*`)

Dans les années 1990, pour satisfaire les besoins des pays européens, ont été définis plusieurs encodages alternatifs, connus sous le nom de `ISO/IEC 8859-*`, nommés aussi `Latin-*`. Idéalement, on aurait pu et **certainement dû** définir un seul encodage pour représenter tous les nouveaux caractères, mais entre toutes les langues européennes, le nombre de caractères à ajouter était substantiel, et cet encodage unifié aurait largement dépassé 256 caractères différents, il n'aurait donc **pas été possible** de tout faire tenir sur un octet.

On a préféré préserver la "bonne propriété" du modèle *un caractère* == *un octet*, ceci afin de préserver le code existant qui aurait sinon dû être retouché ou réécrit.

Dès lors il n'y avait pas d'autre choix que de définir **plusieurs** encodages distincts, par exemple, pour le français on a utilisé à l'époque `ISO/IEC 8859-1 (Latin-1)`, pour le russe `ISO/IEC 5589-5 (Latin/Cyrillic)`.

À ce stade, le ver était dans le fruit. Depuis cette époque pour ouvrir un fichier il faut connaître son encodage.

Unicode

Lorsque l'on a ensuite cherché à manipuler aussi les langues asiatiques, il a de toute façon fallu définir de nouveaux encodages beaucoup plus larges. C'est ce qui a été fait par le standard Unicode qui définit 3 nouveaux encodages :

— `UTF-8` : un encodage à taille variable, à base d'octets, qui maximise la compatibilité avec US-ASCII ;

— `UTF-16` : un encodage à taille variable, à base de mots de 16 bits ;

— `UTF-32` : un encodage à taille fixe, à base de mots de 32 bits ;

Ces 3 standards couvrent le même jeu de caractères (113 021 tout de même dans la dernière version). Parmi ceux-ci le plus utilisé est certainement `UTF-8`. Un texte ne contenant que des caractères du code `US-ASCII` initial peut être lu avec l'encodage `UTF-8`.

Pour être enfin tout à fait exhaustif, si on sait qu'un fichier est au format Unicode, on peut déterminer quel est l'encodage qu'il utilise, en se basant sur les 4 premiers octets du document. Ainsi dans ce cas particulier (lorsqu'on est sûr qu'un document utilise un des trois encodages Unicode) il n'est plus nécessaire de connaître son encodage de manière "externe".

2.2 Les outils de base sur les chaînes de caractères (`str`)

2.2.1 Complément - niveau intermédiaire

Lire la documentation

Même après des années de pratique, il est difficile de se souvenir de toutes les méthodes travaillant sur les chaînes de caractères. Aussi il est toujours utile de recourir à la documentation embarquée

```
In [1]: 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 handler.
 |  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__(...)
 |      S.__format__(format_spec) -> str
 |
 |      Return a formatted version of S as described by format_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.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  __sizeof__(...)
|      S.__sizeof__() -> size of S in memory, in bytes
|
|  __str__(self, /)
|      Return str(self).
|
|  capitalize(...)
|      S.capitalize() -> str
|
|      Return a capitalized version of S, i.e. make the first character
|      have upper case and the rest lower case.
|
```

```
|  casefold(...)
|      S.casefold() -> str
|
|      Return a version of S suitable for caseless comparisons.
|
|  center(...)
|      S.center(width[, fillchar]) -> str
|
|      Return S centered in a string of length width. Padding is
|      done using the specified fill character (default is a space)
|
|  count(...)
|      S.count(sub[, start[, end]]) -> int
|
|      Return the number of non-overlapping occurrences of substring sub in
|      string S[start:end].  Optional arguments start and end are
|      interpreted as in slice notation.
|
|  encode(...)
|      S.encode(encoding='utf-8', errors='strict') -> bytes
|
|      Encode S using the codec registered for encoding. Default encoding
|      is 'utf-8'. errors may be given to set a different error
|      handling scheme. Default is 'strict' meaning that encoding errors raise
|      a UnicodeEncodeError. Other possible values are 'ignore', 'replace' and
|      'xmlcharrefreplace' as well as any other name registered with
|      codecs.register_error that can handle UnicodeEncodeErrors.
|
|  endswith(...)
|      S.endswith(suffix[, start[, end]]) -> bool
|
|      Return True if S ends with the specified suffix, False otherwise.
|      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(...)
|      S.expandtabs(tabsize=8) -> str
|
|      Return a copy of S where all tab characters are expanded using spaces.
|      If tabsize is not given, a tab size of 8 characters is assumed.
|
|  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 notation.
|
|      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 notation.
|
|      Raises ValueError when the substring is not found.
|
|  isalnum(...)
|      S.isalnum() -> bool
|
|      Return True if all characters in S are alphanumeric
|      and there is at least one character in S, False otherwise.
|
|  isalpha(...)
|      S.isalpha() -> bool
|
|      Return True if all characters in S are alphabetic
|      and there is at least one character in S, False otherwise.
|
|  isdecimal(...)
|      S.isdecimal() -> bool
|
|      Return True if there are only decimal characters in S,
|      False otherwise.
|
|  isdigit(...)
|      S.isdigit() -> bool
|
|      Return True if all characters in S are digits
|      and there is at least one character in S, False otherwise.
|
|  isidentifier(...)
|      S.isidentifier() -> bool
|
|      Return True if S is a valid identifier according
|      to the language definition.
|
```

```
|      Use keyword.iskeyword() to test for reserved identifiers
|      such as "def" and "class".
|
|  islower(...)
|      S.islower() -> bool
|
|      Return True if all cased characters in S are lowercase and there is
|      at least one cased character in S, False otherwise.
|
|  isnumeric(...)
|      S.isnumeric() -> bool
|
|      Return True if there are only numeric characters in S,
|      False otherwise.
|
|  isprintable(...)
|      S.isprintable() -> bool
|
|      Return True if all characters in S are considered
|      printable in repr() or S is empty, False otherwise.
|
|  isspace(...)
|      S.isspace() -> bool
|
|      Return True if all characters in S are whitespace
|      and there is at least one character in S, False otherwise.
|
|  istitle(...)
|      S.istitle() -> bool
|
|      Return True if S is a titlecased string and there is at least one
|      character in S, i.e. upper- and titlecase characters may only
|      follow uncased characters and lowercase characters only cased ones.
|      Return False otherwise.
|
|  isupper(...)
|      S.isupper() -> bool
|
|      Return True if all cased characters in S are uppercase and there is
|      at least one cased character in S, False otherwise.
|
|  join(...)
|      S.join(iterable) -> str
|
|      Return a string which is the concatenation of the strings in the
|      iterable.  The separator between elements is S.
|
|  ljust(...)
|      S.ljust(width[, fillchar]) -> str
|
|      Return S left-justified in a Unicode string of length width. Padding is
```

```
|      done using the specified fill character (default is a space).
|
|  lower(...)
|      S.lower() -> str
|
|      Return a copy of the string S converted to lowercase.
|
|  lstrip(...)
|      S.lstrip([chars]) -> str
|
|      Return a copy of the string S with leading whitespace removed.
|      If chars is given and not None, remove characters in chars instead.
|
|  partition(...)
|      S.partition(sep) -> (head, sep, tail)
|
|      Search for the separator sep in S, and return the part before it,
|      the separator itself, and the part after it.  If the separator is not
|      found, return S and two empty strings.
|
|  replace(...)
|      S.replace(old, new[, count]) -> str
|
|      Return a copy of S with all occurrences of substring
|      old replaced by new.  If the optional argument count is
|      given, only the first count 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 notation.
|
|      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 notation.
|
|      Raises ValueError when the substring is not found.
|
|  rjust(...)
|      S.rjust(width[, fillchar]) -> str
|
|      Return S right-justified in a string of length width. Padding is
|      done using the specified fill character (default is a space).
|
```

```
|  rpartition(...)
|      S.rpartition(sep) -> (head, sep, tail)
|
|      Search for the separator sep in S, starting at the end of S, and return
|      the part before it, the separator itself, and the part after it.  If the
|      separator is not found, return two empty strings and S.
|
|  rsplit(...)
|      S.rsplit(sep=None, maxsplit=-1) -> list of strings
|
|      Return a list of the words in S, using sep as the
|      delimiter string, starting at the end of the string and
|      working to the front.  If maxsplit is given, at most maxsplit
|      splits are done. If sep is not specified, any whitespace string
|      is a separator.
|
|  rstrip(...)
|      S.rstrip([chars]) -> str
|
|      Return a copy of the string S with trailing whitespace removed.
|      If chars is given and not None, remove characters in chars instead.
|
|  split(...)
|      S.split(sep=None, maxsplit=-1) -> list of strings
|
|      Return a list of the words in S, using sep as the
|      delimiter string.  If maxsplit is given, at most maxsplit
|      splits are done. If sep is not specified or is None, any
|      whitespace string is a separator and empty strings are
|      removed from the result.
|
|  splitlines(...)
|      S.splitlines([keepends]) -> list of strings
|
|      Return a list of the lines in S, 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 otherwise.
|      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(...)
|      S.strip([chars]) -> str
|
|      Return a copy of the string S with leading and trailing
|      whitespace removed.
```

```
|      If chars is given and not None, remove characters in chars instead.
|
|  swapcase(...)
|      S.swapcase() -> str
|
|      Return a copy of S with uppercase characters converted to lowercase
|      and vice versa.
|
|  title(...)
|      S.title() -> str
|
|      Return a titlecased version of S, i.e. words start with title case
|      characters, all remaining cased characters have lower case.
|
|  translate(...)
|      S.translate(table) -> str
|
|      Return a copy of the string S in which each character has been mapped
|      through the given translation table. The table must implement
|      lookup/indexing via __getitem__, for instance a dictionary or list,
|      mapping Unicode ordinals to Unicode ordinals, strings, or None. If
|      this operation raises LookupError, the character is left untouched.
|      Characters mapped to None are deleted.
|
|  upper(...)
|      S.upper() -> str
|
|      Return a copy of S converted to uppercase.
|
|  zfill(...)
|      S.zfill(width) -> str
|
|      Pad a numeric string S with zeros on the left, to fill a field
|      of the specified width. The string S is never truncated.
|
|  ----------------------------------------------------------------------
|  Static methods defined here:
|
|  maketrans(x, y=None, z=None, /)
|      Return a translation table usable for str.translate().
|
|      If there is only one argument, it must be a dictionary mapping Unicode
|      ordinals (integers) or characters to Unicode ordinals, strings 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 mapped to the
|      character at the same position in y. If there is a third argument, it
|      must be a string, whose characters will be mapped to None in the result.
```

Nous allons tenter ici de citer les méthodes les plus utilisées. Nous n'avons le temps que de les utiliser de manière très simple, mais bien souvent il est possible de passer en argument des options permettant de ne travailler que sur une sous-chaîne, ou sur la première ou dernière occurrence d'une sous-chaîne. Nous vous renvoyons à la documentation pour obtenir toutes les précisions utiles.

Découpage - assemblage : `split` **et** `join`

Les méthodes `split` et `join` permettent de découper une chaîne selon un séparateur pour obtenir une liste, et à l'inverse de reconstruire une chaîne à partir d'une liste.

`split` permet donc de découper :

```
In [2]: 'abc=:=def=:=ghi=:=jkl'.split('=:=')

Out[2]: ['abc', 'def', 'ghi', 'jkl']
```

Et à l'inverse :

```
In [3]: "=:=".join(['abc', 'def', 'ghi', 'jkl'])

Out[3]: 'abc=:=def=:=ghi=:=jkl'
```

Attention toutefois si le séparateur est un terminateur, la liste résultat contient alors une dernière chaîne vide. En pratique, on utilisera la méthode `strip`, que nous allons voir ci-dessous, avant la méthode `split` pour éviter ce problème.

```
In [4]: 'abc;def;ghi;jkl;'.split(';')

Out[4]: ['abc', 'def', 'ghi', 'jkl', '']
```

Qui s'inverse correctement cependant :

```
In [5]: ";".join(['abc', 'def', 'ghi', 'jkl', ''])

Out[5]: 'abc;def;ghi;jkl;'
```

Remplacement : `replace`

`replace` est très pratique pour remplacer une sous-chaîne par une autre, avec une limite éventuelle sur le nombre de remplacements :

```
In [6]: "abcdefabcdefabcdef".replace("abc", "zoo")

Out[6]: 'zoodefzoodefzoodef'

In [7]: "abcdefabcdefabcdef".replace("abc", "zoo", 2)

Out[7]: 'zoodefzoodefabcdef'
```

Plusieurs appels à `replace` peuvent être chaînés comme ceci :

```
In [8]: "les [x] qui disent [y]".replace("[x]", "chevaliers").replace("[y]", "Ni")

Out[8]: 'les chevaliers qui disent Ni'
```

Nettoyage : `strip`

On pourrait par exemple utiliser `replace` pour enlever les espaces dans une chaîne, ce qui peut être utile pour "nettoyer" comme ceci :

```
In [9]: " abc:def:ghi ".replace(" ", "")

Out[9]: 'abc:def:ghi'
```

Toutefois bien souvent on préfère utiliser `strip` qui ne s'occupe que du début et de la fin de la chaîne, et gère aussi les tabulations et autres retour à la ligne :

```
In [10]: " \tune chaîne avec des trucs qui dépassent \n".strip()

Out[10]: 'une chaîne avec des trucs qui dépassent'
```

On peut appliquer `strip` avant `split` pour éviter le problème du dernier élément vide :

```
In [11]: 'abc;def;ghi;jkl;'.strip(';').split(';')

Out[11]: ['abc', 'def', 'ghi', 'jkl']
```

Rechercher une sous-chaîne

Plusieurs outils permettent de chercher une sous-chaîne. Il existe `find` qui renvoie le plus petit index où on trouve la sous-chaîne :

```
In [12]: # l'indice du début de la première occurrence
         "abcdefcdefghefghijk".find("def")

Out[12]: 3

In [13]: # ou -1 si la chaîne n'est pas présente
         "abcdefcdefghefghijk".find("zoo")

Out[13]: -1
```

`rfind` fonctionne comme `find` mais en partant de la fin de la chaîne :

```
In [14]: # en partant de la fin
         "abcdefcdefghefghijk".rfind("fgh")

Out[14]: 13

In [15]: # notez que le résultat correspond
         # tout de même toujours au début de la chaîne
         "abcdefcdefghefghijk"[13]

Out[15]: 'f'
```

La méthode `index` se comporte comme `find`, mais en cas d'absence elle lève une **exception** (nous verrons ce concept plus tard) plutôt que de renvoyer `-1` :

```
In [16]: "abcdefcdefghefghijk".index("def")

Out[16]: 3
```

```
In [17]: try:
             "abcdefcdefghefghijk".index("zoo")
         except Exception as e:
             print("OOPS", type(e), e)
```

```
OOPS <class 'ValueError'> substring not found
```

Mais le plus simple pour chercher si une sous-chaîne est dans une autre chaîne est d'utiliser l'instruction `in` sur laquelle nous reviendrons lorsque nous parlerons des séquences :

```
In [18]: "def" in "abcdefcdefghefghijk"
```

```
Out[18]: True
```

La méthode `count` compte le nombre d'occurrences d'une sous-chaîne :

```
In [19]: "abcdefcdefghefghijk".count("ef")
```

```
Out[19]: 3
```

Signalons enfin les méthodes de commodité suivantes :

```
In [20]: "abcdefcdefghefghijk".startswith("abcd")
```

```
Out[20]: True
```

```
In [21]: "abcdefcdefghefghijk".endswith("ghijk")
```

```
Out[21]: True
```

S'agissant des deux dernières, remarquons que :
`chaine.startswith(sous_chaine)` $\iff$ `chaine.find(sous_chaine) == 0`
`chaine.endswith(sous_chaine)` $\iff$ `chaine.rfind(sous_chaine) == (len(chaine) - len(sous_chaine))`
On remarque ici la supériorité en terme d'expressivité des méthodes pythoniques `startswith` et `endswith`.

Changement de casse

Voici pour conclure quelques méthodes utiles qui parlent d'elles-mêmes :

```
In [22]: "monty PYTHON".upper()
```

```
Out[22]: 'MONTY PYTHON'
```

```
In [23]: "monty PYTHON".lower()
```

```
Out[23]: 'monty python'
```

```
In [24]: "monty PYTHON".swapcase()
```

```
Out[24]: 'MONTY python'
```

```
In [25]: "monty PYTHON".capitalize()
```

```
Out[25]: 'Monty python'
```

```
In [26]: "monty PYTHON".title()
```

```
Out[26]: 'Monty Python'
```

Pour en savoir plus

Tous ces outils sont documentés en détail ici (en anglais).

2.3 Formatage de chaînes de caractères

2.3.1 Complément - niveau basique

On désigne par formatage les outils qui permettent d'obtenir une présentation fine des résultats, que ce soit pour améliorer la lisibilité lorsqu'on s'adresse à des humains, ou pour respecter la syntaxe d'un outil auquel on veut passer les données pour un traitement ultérieur.

La fonction `print`

Nous avons jusqu'à maintenant presque toujours utilisé la fonction `print` pour afficher nos résultats. Comme on l'a vu, celle-ci réalise un formatage sommaire : elle insère une espace entre les valeurs qui lui sont passées.

```
In [1]: print(1, 'a', 12 + 4j)
```

```
1 a (12+4j)
```

La seule subtilité notable concernant `print` est que, par défaut, elle ajoute un saut de ligne à la fin. Pour éviter ce comportement, on peut passer à la fonction un argument `end`, qui sera inséré *au lieu* du saut de ligne. Ainsi par exemple :

```
In [2]: # une première ligne
        print("une", "seule", "ligne")
```

```
une seule ligne
```

```
In [3]: # une deuxième ligne en deux appels à print
        print("une", "autre", end=' ')
        print("ligne")
```

```
une autre ligne
```

Il faut remarquer aussi que `print` est capable d'imprimer **n'importe quel objet**. Nous l'avons déjà fait avec les listes et les tuples, voici par exemple un module :

```
In [4]: # on peut imprimer par exemple un objet 'module'
        import math

        print('le module math est', math)
```

```
le module math est <module 'math' (built-in)>
```

En anticipant un peu, voici comment `print` présente les instances de classe (ne vous inquiétez pas, nous apprendrons dans une semaine ultérieure ce que sont les classes et les instances).

```
In [5]: # pour définir la classe Personne
        class Personne:
            pass

        # et pour créer une instance de cette classe
        personne = Personne()
```

```
In [6]: # voilà comment s'affiche une instance de classe
        print(personne)

<__main__.Personne object at 0x0502DB30>
```

On rencontre assez vite les limites de `print` :

— d'une part, il peut être nécessaire de formater une chaîne de caractères sans nécessairement vouloir l'imprimer, ou en tout cas pas immédiatement ;
— d'autre part, les espaces ajoutées peuvent être plus néfastes qu'utiles ;
— enfin, on peut avoir besoin de préciser un nombre de chiffres significatifs, ou de choisir comment présenter une date.

C'est pourquoi il est plus courant de **formater** les chaînes - c'est-à-dire de calculer des chaînes en mémoire, sans nécessairement les imprimer de suite, et c'est ce que nous allons étudier dans ce complément.

Les *f-strings*

Depuis la version 3.6 de Python, on peut utiliser les f-strings, le premier mécanisme de formatage que nous étudierons. C'est le mécanisme de formatage le plus simple et le plus agréable à utiliser.

Je vous recommande tout de même de lire les sections à propos de `format` et de `%`, qui sont encore massivement utilisées dans le code existant (surtout `%` d'ailleurs, bien que essentiellement obsolète).

Mais définissons d'abord quelques données à afficher :

```
In [7]: # donnons-nous quelques variables
        prenom, nom, age = 'Jean', 'Dupont', 35

In [8]: # mon premier f-string
        f"{prenom} {nom} a {age} ans"

Out[8]: 'Jean Dupont a 35 ans'
```

Vous remarquez d'abord que le string commence par `f"`, c'est bien sûr pour cela qu'on l'appelle un *f-string*.

On peut bien entendu ajouter le `f` devant toutes les formes de strings, qu'ils commencent par `'` ou `"` ou `'''` ou `"""`.

Ensuite vous remarquez que les zones délimitées entre `{}` sont remplacées. La logique d'un *f-string*, c'est tout simplement de considérer l'intérieur d'un `{}` comme du code Python (une expression pour être précis), de l'évaluer, et d'utiliser le résultat pour remplir le `{}`.

Ça veut dire, en clair, que je peux faire des calculs à l'intérieur des `{}`.

```
In [9]: # toutes les expressions sont autorisées à l'intérieur d'un {}
        f"dans 10 ans {prenom} aura {age + 10} ans"

Out[9]: 'dans 10 ans Jean aura 45 ans'
```

```
In [10]: # on peut donc aussi mettre des appels de fonction
         notes = [12, 15, 19]
         f"nous avons pour l'instant {len(notes)} notes"

Out[10]: "nous avons pour l'instant 3 notes"
```

Nous allons en rester là pour la partie en niveau basique. Il nous reste à étudier comment chaque `{}` est formaté (par exemple comment choisir le nombre de chiffres significatifs sur un flottant), voyez plus bas pour plus de détails sur ce point.

Comme vous le voyez, les *f-strings* fournissent une méthode très simple et expressive pour formater des données dans des chaînes de caractère. Redisons-le pour être bien clair : un *f-string* **ne réalise pas d'impression**, il faut donc le passer à `print` si l'impression est souhaitée.

La méthode `format`

Avant l'introduction des *f-strings*, la technique recommandée pour faire du formatage était d'utiliser la méthode `format` qui est définie sur les objets `str` et qui s'utilise comme ceci :

```
In [11]: "{} {} a {} ans".format(prenom, nom, age)

Out[11]: 'Jean Dupont a 35 ans'
```

Dans cet exemple le plus simple, les données sont affichées en lieu et place des `{}`, dans l'ordre où elles sont fournies.

Cela convient bien lorsqu'on a peu de données. Si par la suite on veut changer l'ordre par exemple des nom et prénom, on peut bien sûr échanger l'ordre des arguments passés à format, ou encore utiliser la **liaison par position**, comme ceci :

```
In [12]: "{1} {0} a {2} ans".format(prenom, nom, age)

Out[12]: 'Dupont Jean a 35 ans'
```

Dans la pratique toutefois, cette forme est assez peu utile, on lui préfère souvent la **liaison par nom** qui se présente comme ceci :

```
In [13]: "{le_prenom} {le_nom} a {l_age} ans".format(le_nom=nom, le_prenom=prenom, l_age=age)

Out[13]: 'Jean Dupont a 35 ans'
```

Dans ce premier exemple de liaison par nom, nous avons délibérément utilisé des noms différents pour les données externes et pour les noms apparaissant dans le format, pour bien illustrer comment la liaison est résolue, mais on peut aussi bien faire tout simplement :

```
In [14]: "{prenom} {nom} a {age} ans".format(nom=nom, prenom=prenom, age=age)

Out[14]: 'Jean Dupont a 35 ans'
```

Voici qui conclut notre courte introduction à la méthode `format`.

2.3.2 Complément - niveau intermédiaire

La toute première version du formatage : l'opérateur %

`format` a été en fait introduite assez tard dans Python, pour remplacer la technique que nous allons présenter maintenant.

Étant donné le volume de code qui a été écrit avec l'opérateur %, il nous a semblé important d'introduire brièvement cette construction ici. Vous ne devez cependant pas utiliser cet opérateur dans du code moderne, la manière pythonique de formater les chaînes de caractères est le f-string.

Le principe de l'opérateur % est le suivant. On élabore comme ci-dessus un "format" c'est-à-dire le patron de ce qui doit être rendu, auquel on passe des arguments pour "remplir" les trous. Voyons les exemples de tout à l'heure rendus avec l'opérateur % :

```
In [15]: # l'ancienne façon de formater les chaînes avec %
         # est souvent moins lisible
         "%s %s a %s ans" % (prenom, nom, age)
```

```
Out[15]: 'Jean Dupont a 35 ans'
```

On pouvait également avec cet opérateur recourir à un mécanisme de liaison par nommage, en passant par un dictionnaire. Pour anticiper un tout petit peu sur cette notion que nous verrons très bientôt, voici comment

```
In [16]: variables = {'le_nom': nom, 'le_prenom': prenom, 'l_age': age}
         "%(le_nom)s, %(le_prenom)s, %(l_age)s ans" % variables
```

```
Out[16]: 'Dupont, Jean, 35 ans'
```

2.3.3 Complément - niveau avancé

De retour aux *f-strings* et à la fonction `format`, il arrive qu'on ait besoin de spécifier plus finement la façon dont une valeur doit être affichée.

Précision des arrondis

C'est typiquement le cas avec les valeurs flottantes pour lesquelles la précision de l'affichage vient au détriment de la lisibilité. Voici deux formes équivalentes pour obtenir une valeur de pi arrondie :

```
In [17]: from math import pi
```

```
In [18]: # un f-string
         f"pi avec seulement 2 chiffres apres la virgule {pi:.2f}"
```

```
Out[18]: 'pi avec seulement 2 chiffres apres la virgule 3.14'
```

```
In [19]: # avec format() et liaison par nom
         "pi avec seulement 2 chiffres apres la virgule {flottant:.2f}".format(flottant=pi)
```

```
Out[19]: 'pi avec seulement 2 chiffres apres la virgule 3.14'
```

Dans ces deux exemples, la partie à l'intérieur des `{}` et à droite du `:` s'appelle le format, ici `.2f` ; vous remarquez que c'est le même pour les *f-strings* et pour `format`, et c'est toujours le cas. C'est pourquoi on ne verra plus à partir d'ici que des exemples avec les *f-strings*.

`0` en début de nombre

Pour forcer un petit entier à s'afficher sur 4 caractères, avec des `0` ajoutés au début si nécessaire :

```
In [20]: x = 15

         f"{x:04d}"

Out[20]: '0015'
```

Ici on utilise le format `d` (toutes ces lettres `d`, `f`, `g` viennent des formats ancestraux de la libc comme `printf`). Ici avec `04d` on précise qu'on veut une sortie sur 4 caractères et qu'il faut remplir à gauche si nécessaire avec des `0`.

Largeur fixe

Dans certains cas, on a besoin d'afficher des données en colonnes de largeur fixe, on utilise pour cela les formats `<` `^` et `>` pour afficher à gauche, au centre, ou à droite d'une zone de largeur fixe :

```
In [21]: # les données à afficher
         comptes = [
          ('Apollin', 'Dupont', 127),
          ('Myrtille', 'Lamartine', 25432),
          ('Prune', 'Soc', 827465),
         ]

         for prenom, nom, solde in comptes:
             print(f"{prenom:<10} -- {nom:^12} -- {solde:>8} €")
```

```
Apollin    --    Dupont    --      127 €
Myrtille   --  Lamartine   --    25432 €
Prune      --     Soc      --   827465 €
```

Voir aussi

Nous vous invitons à vous reporter à la documentation de `format` pour plus de détails sur les formats disponibles, et notamment aux nombreux exemples qui y figurent.

2.4 Obtenir une réponse de l'utilisateur

2.4.1 Complément - niveau basique

Occasionnellement, il peut être utile de poser une question à l'utilisateur.

La fonction `input`

C'est le propos de la fonction `input`. Par exemple :

```
In [1]: nom_ville = input("Entrez le nom de la ville : ")
        print(f"nom_ville={nom_ville}")

Entrez le nom de la ville : Metropolis
nom_ville=Metropolis
```

Attention à bien vérifier/convertir

Notez bien que `input` renvoie **toujours une chaîne de caractère** (`str`). C'est assez évident, mais il est très facile de l'oublier et de passer cette chaîne directement à une fonction qui s'attend à recevoir, par exemple, un nombre entier, auquel cas les choses se passent mal :

```
>>> input("nombre de lignes ? ") + 3
nombre de lignes ? 12
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Dans ce cas il faut appeler la fonction `int` pour convertir le résultat en un entier :

```
In [2]: int(input("Nombre de lignes ? ")) + 3

Nombre de lignes ? 5


Out[2]: 8
```

Limitations

Cette fonction peut être utile pour vos premiers pas en Python.

En pratique toutefois, on utilise assez peu cette fonction, car les applications "réelles" viennent avec leur propre interface utilisateur, souvent graphique, et disposent donc d'autres moyens que celui-ci pour interagir avec l'utilisateur.

Les applications destinées à fonctionner dans un terminal, quant à elles, reçoivent traditionnellement leurs données de la ligne de commande. C'est le propos du module `argparse` que nous avons déjà rencontré en première semaine.

2.5 Expressions régulières et le module `re`

2.5.1 Complément - niveau basique

Avertissement

Après avoir joué ce cours plusieurs années de suite, l'expérience nous montre qu'il est difficile de trouver le bon moment pour appréhender les expressions régulières.

D'un côté il s'agit de manipulations de chaînes de caractères, mais d'un autre cela nécessite de créer des instances de classes, et donc d'avoir vu la programmation orientée objet. Du coup, les premières années nous les avions étudiées tout à la fin du cours, ce qui avait pu créer une certaine frustration.

C'est pourquoi nous avons décidé à présent de les étudier très tôt, dans cette séquence consacrée aux chaines de caractères. Les étudiants qui seraient décontenancés par ce contenu sont invités à y retourner après la semaine 6, consacrée à la programmation objet.

Il nous semble important de savoir que ces fonctionnalités existent dans le langage, le détail de leur utilisation n'est toutefois pas critique, et on peut parfaitement faire l'impasse sur ce complément en première lecture.

Une expression régulière est un objet mathématique permettant de décrire un ensemble de textes qui possèdent des propriétés communes. Par exemple, s'il vous arrive d'utiliser un terminal, et que vous tapez

```
$ dir *.txt
```

(ou `ls *.txt` sur linux ou mac), vous utilisez l'expression régulière `*.txt` qui désigne tous les fichiers dont le nom se termine par `.txt`. On dit que l'expression régulière *filtre* toutes les chaînes qui se terminent par `.txt` (l'expression anglaise consacrée est le *pattern matching*).

Le langage Perl a été le premier à populariser l'utilisation des expressions régulières en les supportant nativement dans le langage, et non au travers d'une librairie. En python, les expressions régulières sont disponibles de manière plus traditionnelle, via le module `re` (regular expressions) de la librairie standard. Le propos de ce complément est de vous en donner une première introduction.

```
In [1]: import re
```

Survol

Pour ceux qui ne souhaitent pas approfondir, voici un premier exemple ; on cherche à savoir si un objet `chaine` est ou non de la forme `*-*.txt`, et si oui, à calculer la partie de la chaine qui remplace le `*` :

```
In [2]: # un objet 'expression régulière' - on dit aussi "pattern"
        regexp = "(.*)-(.*)\.txt"

In [3]: # la chaine de départ
        chaine = "abcdef.txt"

In [4]: # la fonction qui calcule si la chaine "matche" le pattern
        match = re.match(regexp, chaine)
        match is None
```

```
Out[4]: True
```

Le fait que l'objet `match` vaut `None` indique que la chaine n'est pas de la bonne forme (il manque un - dans le nom) ; avec une autre chaine par contre :

```
In [5]: # la chaine de départ
        chaine = "abc-def.txt"

In [6]: match = re.match(regexp, chaine)
        match is None

Out[6]: False
```

Ici `match` est un objet, qui nous permet ensuite d'"extraire" les différentes parties, comme ceci :

```
In [7]: match[1]

Out[7]: 'abc'

In [8]: match[2]

Out[8]: 'def'
```

Bien sûr on peut faire des choses beaucoup plus élaborées avec `re`, mais en première lecture cette introduction doit vous suffire pour avoir une idée de ce qu'on peut faire avec les expressions régulières.

2.5.2 Complément - niveau intermédiaire

Approfondissons à présent :

Dans un terminal, `*.txt` est une expression régulière très simple. Le module `re` fournit le moyen de construire des expressions régulières très élaborées et plus puissantes que ce que supporte le terminal. C'est pourquoi la syntaxe des regexps de `re` est un peu différente. Par exemple comme on vient de le voir, pour filtrer la même famille de chaînes que `*-*.txt` avec le module `re`, il nous a fallu écrire l'expression régulière sous une forme légèrement différente.

Je vous conseille d'avoir sous la main la documentation du module `re` pendant que vous lisez ce complément.

Avertissement

Dans ce complément nous serons amenés à utiliser des traits qui dépendent du LOCALE, c'est-à-dire, pour faire simple, de la configuration de l'ordinateur vis-à-vis de la langue.

Tant que vous exécutez ceci dans le notebook sur la plateforme, en principe tout le monde verra exactement la même chose. Par contre, si vous faites tourner le même code sur votre ordinateur, il se peut que vous obteniez des résultats légèrement différents.

Un exemple simple

`findall` On se donne deux exemples de chaînes

```
In [9]: sentences = ['Lacus a donec, vitae gravida proin sociis.',
                     'Neque ipsum! rhoncus cras quam.']
```

On peut **chercher tous** les mots se terminant par `a` ou `m` dans une chaîne avec `findall`

```
In [10]: for sentence in sentences:
             print(f"---- dans >{sentence}<")
             print(re.findall(r"\w*[am]\W", sentence))
```

```
---- dans >Lacus a donec, vitae gravida proin sociis.<
['a ', 'gravida ']
---- dans >Neque ipsum! rhoncus cras quam.<
['ipsum!', 'quam.']
```

Ce code permet de chercher toutes (`findall`) les occurrences de l'expression régulière, qui ici est définie par le *raw-string*

```
r"\w*[am]\W"
```

Nous verrons tout à l'heure comment fabriquer des expressions régulières plus en détail, mais pour démystifier au moins celle-ci, on a mis bout à bout les morceaux suivants.

- `\w*` : on veut trouver une sous-chaîne qui commence par un nombre quelconque, y compris nul (`*`) de caractères alphanumériques (`\w`). Ceci est défini en fonction de votre LOCALE, on y reviendra.
- `[am]` : immédiatement après, il nous faut trouver un caratère `a` ou `m`.
- `\W` : et enfin, il nous faut un caractère qui ne soit **pas** alphanumérique. Ceci est important puisqu'on cherche les mots qui **se terminent** par un `a` ou un `m`, si on ne le mettait pas on obtiendrait ceci

```
In [11]: # le \W final est important
         # voici ce qu'on obtient si on l'omet
         for sentence in sentences:
             print(f"---- dans >{sentence}<")
             print(re.findall(r"\w*[am]", sentence))
```

```
---- dans >Lacus a donec, vitae gravida proin sociis.<
['La', 'a', 'vita', 'gravida']
---- dans >Neque ipsum! rhoncus cras quam.<
['ipsum', 'cra', 'quam']
```

`split`

Une autre forme simple d'utilisation des regexps est `re.split`, qui fournit une fonctionnalité voisine de `str.split`, mais ou les séparateurs sont exprimés comme une expression régulière

```
In [12]: for sentence in sentences:
             print(f"---- dans >{sentence}<")
             print(re.split(r"\W+", sentence))
             print()

---- dans >Lacus a donec, vitae gravida proin sociis.<
['Lacus', 'a', 'donec', 'vitae', 'gravida', 'proin', 'sociis', '']

---- dans >Neque ipsum! rhoncus cras quam.<
['Neque', 'ipsum', 'rhoncus', 'cras', 'quam', '']
```

Ici l'expression régulière, qui bien sûr décrit le séparateur, est simplement `\W+` c'est-à-dire toute suite d'au moins un caractère non alphanumérique.

Nous avons donc là un moyen simple, et plus puissant que `str.split`, de couper un texte en mots.

sub

Une troisième méthode utilitaire est `re.sub` qui permet de remplacer les occurrences d'une *regexp*, comme par exemple

```
In [13]: for sentence in sentences:
             print(f"---- dans >{sentence}<")
             print(re.sub(r"(\w+)", r"X\1Y", sentence))
             print()

---- dans >Lacus a donec, vitae gravida proin sociis.<
XLacusY XaY XdonecY, XvitaeY XgravidaY XproinY XsociisY.

---- dans >Neque ipsum! rhoncus cras quam.<
XNequeY XipsumY! XrhoncusY XcrasY XquamY.
```

Ici, l'expression régulière (le premier argument) contient un **groupe** : on a utilisé des parenthèses autour du `\w+`. Le second argument est la chaîne de remplacement, dans laquelle on a fait **référence au groupe** en écrivant `\1`, qui veut dire tout simplement "le premier groupe".

Donc au final, l'effet de cet appel est d'entourer toutes les suites de caractères alphanumériques par `X` et `Y`.

Pourquoi un *raw-string* ?

En guise de digression, il n'y a aucune obligation à utiliser un *raw-string*, d'ailleurs on rappelle qu'il n'y a pas de différence de nature entre un *raw-string* et une chaîne usuelle

```
In [14]: raw = r'abc'
         regular = 'abc'
```

```
        # comme on a pris une 'petite' chaîne ce sont les mêmes objets
        print(f"both compared with is → {raw is regular}")
        # et donc a fortiori
        print(f"both compared with == → {raw == regular}")
```

```
both compared with is → True
both compared with == → True
```

Il se trouve que le *backslash* \ à l'intérieur des expressions régulières est d'un usage assez courant - on l'a vu déjà plusieurs fois. C'est pourquoi on **utilise fréquemment un *raw-string*** pour décrire une expression régulière, et en général à chaque fois qu'elle comporte un *backslash*. On rappelle que le raw-string désactive l'interprétation des \ à l'intérieur de la chaîne, par exemple, `\t` est interprété comme un caractère de tabulation. Sans raw-string, il faut doubler tous les \ pour qu'il n'y ait pas d'interprétation.

Un deuxième exemple

Nous allons maintenant voir comment on peut d'abord vérifier si une chaîne est conforme au critère défini par l'expression régulière, mais aussi *extraire* les morceaux de la chaîne qui correspondent aux différentes parties de l'expression.

Pour cela, supposons qu'on s'intéresse aux chaînes qui comportent 5 parties, une suite de chiffres, une suite de lettres, des chiffres à nouveau, des lettres et enfin de nouveau des chiffres.

Pour cela on considère ces trois chaines en entrée

```
In [15]: samples = ['890hj000nnm890',    # cette entrée convient
                    '123abc456def789',   # celle-ci aussi
                    '8090abababab879',   # celle-ci non
                    ]
```

`match` Pour commencer, voyons que l'on peut facilement **vérifier si une chaîne vérifie** ou non le critère.

```
In [16]: regexp1 = "[0-9]+[A-Za-z]+[0-9]+[A-Za-z]+[0-9]+"
```

Si on applique cette expression régulière à toutes nos entrées

```
In [17]: for sample in samples:
             match = re.match(regexp1, sample)
             print(f"{sample:16s} → {match}")
```

```
890hj000nnm890   → <_sre.SRE_Match object; span=(0, 14), match='890hj000nnm890'>
123abc456def789  → <_sre.SRE_Match object; span=(0, 15), match='123abc456def789'>
8090abababab879  → None
```

Pour rendre ce résultat un peu plus lisible nous nous définissons une petite fonction de confort.

```
In [18]: # pour simplement visualiser si on a un match ou pas
         def nice(match):
             # le retour de re.match est soit None, soit un objet match
             return "no" if match is None else "Match!"
```

Avec quoi on peut refaire l'essai sur toutes nos entrées.

```
In [19]: # la même chose mais un peu moins encombrant
         print(f"REGEXP={regexp1}\n")
         for sample in samples:
             match = re.match(regexp1, sample)
             print(f"{sample:>16s} → {nice(match)}")
```

```
REGEXP=[0-9]+[A-Za-z]+[0-9]+[A-Za-z]+[0-9]+

  890hj000nnm890 → Match!
 123abc456def789 → Match!
 8090abababab879 → no
```

Ici plutôt que d'utiliser les raccourcis comme `\w` j'ai préféré écrire explicitement les ensembles de caractères en jeu. De cette façon, on rend son code indépendant du LOCALE si c'est ce qu'on veut faire. Il y a deux morceaux qui interviennent tour à tour :

— `[0-9]+` signifie une suite de au moins un caractère dans l'intervalle `[0-9]`,

— `[A-Za-z]+` pour une suite d'au moins un caractère dans l'intervalle `[A-Z]` ou dans l'intervalle `[a-z]`.

Et comme tout à l'heure on a simplement juxtaposé les morceaux dans le bon ordre pour construire l'expression régulière complète.

Nommer un morceau (un groupe)

```
In [20]: # on se concentre sur une entrée correcte
         haystack = samples[1]
         haystack
```

```
Out[20]: '123abc456def789'
```

Maintenant, on va même pouvoir **donner un nom** à un morceau de la regexp, ici on désigne par `needle` le groupe de chiffres du milieu.

```
In [21]: # la même regexp, mais on donne un nom au groupe de chiffres central
         regexp2 = "[0-9]+[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+[0-9]+"
```

Et une fois que c'est fait, on peut demander à l'outil de nous **retrouver la partie correspondante** dans la chaine initiale :

```
In [22]: print(re.match(regexp2, haystack).group('needle'))
```

```
456
```

Dans cette expression on a utilisé un **groupe nommé** `(?P<needle>[0-9]+)`, dans lequel :

— les parenthèses définissent un groupe,

— `?P<needle>` spécifie que ce groupe pourra être référencé sous le nom `needle` (cette syntaxe très absconse est héritée semble-t-il de perl).

Un troisième exemple

Enfin, et c'est un trait qui n'est pas présent dans tous les langages, on peut restreindre un morceau de chaîne à être identique à un groupe déjà vu plus tôt dans la chaîne. Dans l'exemple ci-dessus, on pourrait ajouter comme contrainte que le premier et le dernier groupes de chiffres soient identiques, comme ceci

```
In [23]: regexp3 = "(?P<id>[0-9]+)[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+(?P=id)"
```

Si bien que maintenant, avec les mêmes entrées que tout à l'heure

```
In [24]: print(f"REGEXP={regexp3}\n")
         for sample in samples:
             match = re.match(regexp3, sample)
             print(f"{sample:>16s} → {nice(match)}")
```

```
REGEXP=(?P<id>[0-9]+)[A-Za-z]+(?P<needle>[0-9]+)[A-Za-z]+(?P=id)

  890hj000nnm890 → Match!
 123abc456def789 → no
 8090abababab879 → no
```

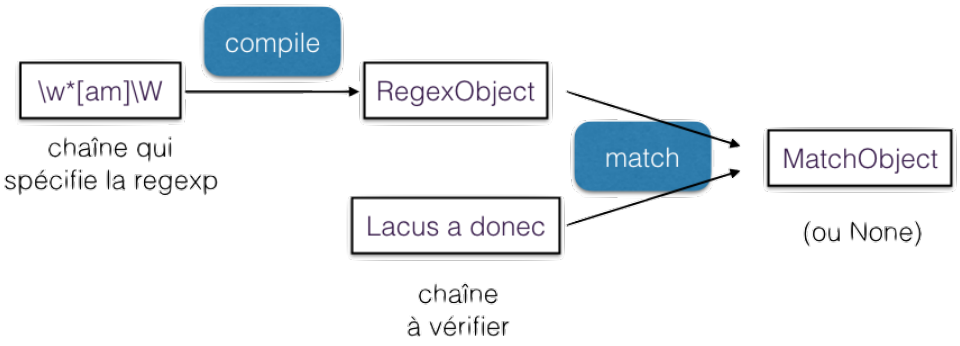
Comme précédemment on a défini le groupe nommé `id` comme étant la première suite de chiffres. La nouveauté ici est la **contrainte** qu'on a imposée sur le dernier groupe avec `(?P=id)`. Comme vous le voyez, on n'obtient un *match* qu'avec les entrées dans lesquelles le dernier groupe de chiffres est identique au premier.

Comment utiliser la librairie

Avant d'apprendre à écrire une expression régulière, disons quelques mots du mode d'emploi de la librairie.

Fonctions de commodité et *workflow*

Comme vous le savez peut-être, une expression régulière décrite sous forme de chaîne, comme par exemple `"\w*[am]\W"`, peut être traduite dans un **automate fini** qui permet de faire le filtrage avec une chaîne. C'est ce qui explique le *workflow* que nous avons résumé dans cette figure.



La méthode recommandée pour utiliser la librairie, lorsque vous avez le même *pattern* à appliquer à un grand nombre de chaînes, est de :

— compiler **une seule fois** votre chaîne en un automate, qui est matérialisé par un objet de la classe `re.RegexObject`, en utilisant `re.compile`,
— puis d'**utiliser directement cet objet** autant de fois que vous avez de chaînes.

Nous avons utilisé dans les exemples plus haut (et nous continuerons plus bas pour une meilleure lisibilité) des **fonctions de commodité** du module, qui sont pratiques, par exemple, pour mettre au point une expression régulière en mode interactif, mais qui ne **sont pas forcément** adaptées dans tous les cas.

Ces fonctions de commodité fonctionnent toutes sur le même principe :

`re.match(regexp, sample)` $\Longleftrightarrow$ `re.compile(regexp).match(sample)`

Donc à chaque fois qu'on utilise une fonction de commodité, on recompile la chaîne en automate, ce qui, dès qu'on a plus d'une chaîne à traiter, représente un surcoût.

```
In [25]: # au lieu de faire comme ci-dessus:

         # imaginez 10**6 chaînes dans samples
         for sample in samples:
             match = re.match(regexp3, sample)
             print(f"{sample:>16s} → {nice(match)}")
```

```
 890hj000nnm890 → Match!
123abc456def789 → no
8090abababab879 → no
```

```
In [26]: # dans du vrai code on fera plutôt:

         # on compile la chaîne en automate une seule fois
         re_obj3 = re.compile(regexp3)

         # ensuite on part directement de l'automate
         for sample in samples:
             match = re_obj3.match(sample)
             print(f"{sample:>16s} → {nice(match)}")
```

```
 890hj000nnm890 → Match!
123abc456def789 → no
8090abababab879 → no
```

Cette deuxième version ne compile qu'une fois la chaîne en automate, et donc est plus efficace.

Les méthodes sur la classe `RegexObject` Les objets de la classe `RegexObject` représentent donc l'automate à état fini qui est le résultat de la compilation de l'expression régulière. Pour résumer ce qu'on a déjà vu, les méthodes les plus utiles sur un objet `RegexObject` sont :

— `match` et `search`, qui cherchent un *match* soit uniquement au début (`match`) ou n'importe où dans la chaîne (`search`),
— `findall` et `split` pour chercher toutes les occurences (`findall`) ou leur négatif (`split`),
— `sub` (qui aurait pu sans doute s'appeler `replace`, mais c'est comme ça) pour remplacer les occurrences de pattern.

Exploiter le résultat

Les **méthodes** disponibles sur la classe `re.MatchObject` sont documentées en détail ici. On en a déjà rencontré quelques-unes, en voici à nouveau un aperçu rapide.

```
In [27]: # exemple
         sample = "    Isaac Newton, physicist"
         match = re.search(r"(\w+) (?P<name>\w+)", sample)
```

`re` et `string` pour retrouver les données d'entrée du match.

```
In [28]: match.string
```

```
Out[28]: '    Isaac Newton, physicist'
```

```
In [29]: match.re
```

```
Out[29]: re.compile(r'(\w+) (?P<name>\w+)', re.UNICODE)
```

`group`, `groups`, `groupdict` pour retrouver les morceaux de la chaîne d'entrée qui correspondent aux **groupes** de la regexp. On peut y accéder par rang, ou par nom (comme on l'a vu plus haut avec `needle`).

```
In [30]: match.groups()
```

```
Out[30]: ('Isaac', 'Newton')
```

```
In [31]: match.group(1)
```

```
Out[31]: 'Isaac'
```

```
In [32]: match.group('name')
```

```
Out[32]: 'Newton'
```

```
In [33]: match.group(2)
```

```
Out[33]: 'Newton'
```

```
In [34]: match.groupdict()
```

```
Out[34]: {'name': 'Newton'}
```

Comme on le voit pour l'accès par rang **les indices commencent à 1** pour des raisons historiques (on peut déjà référencer `\1` en sed depuis la fin des années 70).

On peut aussi accéder au **groupe 0** comme étant la partie de la chaîne de départ qui a effectivement été filtrée par l'expression régulière, et qui peut tout à fait être au beau milieu de la chaîne de départ, comme dans notre exemple

```
In [35]: match.group(0)
```

```
Out[35]: 'Isaac Newton'
```

`expand` permet de faire une espèce de `str.format` avec les valeurs des groupes.

```
In [36]: match.expand(r"last_name \g<name> first_name \1")
```

```
Out[36]: 'last_name Newton first_name Isaac'
```

`span` pour connaître les index dans la chaîne d'entrée pour un groupe donné.

```
In [37]: begin, end = match.span('name')
         sample[begin:end]
```

```
Out[37]: 'Newton'
```

Les différents modes (*flags*) Enfin il faut noter qu'on peut passer à `re.compile` un certain nombre de *flags* qui modifient globalement l'interprétation de la chaîne, et qui peuvent rendre service.

Vous trouverez une liste exhaustive de ces *flags* ici. Ils ont en général un nom long et parlant, et un alias court sur un seul caractère. Les plus utiles sont sans doute :

— `IGNORECASE` (*alias* `I`) pour, eh bien, ne pas faire la différence entre minuscules et majuscules,

— `UNICODE` (*alias* `U`) pour rendre les séquences `\w` et autres basées sur les propriétés des caractères dans la norme Unicode,

— `LOCALE` (*alias* `L`) cette fois `\w` dépend du `locale` courant,

— `MULTILINE` (*alias* `M`), et

— `DOTALL` (*alias* S) pour ces deux flags voir la discussion à la fin du complément.

Comme c'est souvent le cas, on doit passer à `re.compile` un **ou logique** (caractère `|`) des différents flags que l'on veut utiliser, c'est-à-dire qu'on fera par exemple

```
In [38]: regexp = "a*b+"
         re_obj = re.compile(regexp, flags=re.IGNORECASE | re.DEBUG)

MAX_REPEAT 0 MAXREPEAT
  LITERAL 97
MAX_REPEAT 1 MAXREPEAT
  LITERAL 98


In [39]: # on ignore la casse des caractères
         print(regexp, "->", nice(re_obj.match("AabB")))

a*b+ -> Match!
```

Comment construire une expression régulière

Nous pouvons à présent voir comment construire une expression régulière, en essayant de rester synthétique (la documentation du module `re` en donne une version exhaustive).

La brique de base : le caractère

Au commencement il faut spécifier des caractères.

— **un seul** caractère :

  — vous le citez tel quel, en le précédent d'un backslash `\` s'il a par ailleurs un sens spécial dans le micro-langage de regexps (comme `+`, `*`, `[`, etc.) ;

— l'**attrape-tout** (*wildcard*) :

  — un point `.` signifie "n'importe quel caractère" ;

— **un ensemble** de caractères avec la notation `[...]` qui permet de décrire par exemple :

  — `[a1=]` un ensemble in extenso, ici un caractère parmi `a`, `1`, ou `=`,

  — `[a-z]` un intervalle de caractères, ici de `a` à `z`,

  — `[15e-g]` un mélange des deux, ici un ensemble qui contiendrait `1`, `5`, `e`, `f` et `g`,

  - `[^15e-g]` une **négation**, qui a `^` comme premier caractère dans les `[]`, ici tout sauf l'ensemble précédent ;
- un **ensemble prédéfini** de caractères, qui peuvent alors dépendre de l'environnement (UNICODE et LOCALE) avec entre autres les notations :
  - `\w` les caractères alphanumériques, et `\W` (les autres),
  - `\s` les caractères "blancs" - espace, tabulation, saut de ligne, etc., et `\S` (les autres),
  - `\d` pour les chiffres, et `\D` (les autres).

```
In [40]: sample = "abcd"

         for regexp in ['abcd', 'ab[cd][cd]', 'ab[a-z]d', r'abc.', r'abc\.']:
             match = re.match(regexp, sample)
             print(f"{sample} / {regexp:<10s} → {nice(match)}")
```

```
abcd / abcd       → Match!
abcd / ab[cd][cd] → Match!
abcd / ab[a-z]d   → Match!
abcd / abc.       → Match!
abcd / abc\.      → no
```

Pour ce dernier exemple, comme on a backslashé le `.` il faut que la chaîne en entrée contienne vraiment un `.`

```
In [41]: print(nice(re.match (r"abc\.", "abc.")))
```

```
Match!
```

En série ou en parallèle Si je fais une analogie avec les montages électriques, jusqu'ici on a vu le montage en série, on met des expressions régulières bout à bout qui filtrent (`match`) la chaine en entrée séquentiellement du début à la fin. On a *un peu* de marge pour spécifier des alternatives, lorsqu'on fait par exemple

```
"ab[cd]ef"
```

mais c'est limité à **un seul** caractère. Si on veut reconnaitre deux mots qui n'ont pas grand-chose à voir comme `abc` **ou** `def`, il faut en quelque sorte mettre deux regexps en parallèle, et c'est ce que permet l'opérateur `|`

```
In [42]: regexp = "abc|def"

         for sample in ['abc', 'def', 'aef']:
             match = re.match(regexp, sample)
             print(f"{sample} / {regexp} → {nice(match)}")
```

```
abc / abc|def → Match!
def / abc|def → Match!
aef / abc|def → no
```

Fin(s) de chaîne

Selon que vous utilisez `match` ou `search`, vous précisez si vous vous intéressez uniquement à un match en début (`match`) ou n'importe où (`search`) dans la chaîne.

Mais indépendamment de cela, il peut être intéressant de "coller" l'expression en début ou en fin de ligne, et pour ça il existe des caractères spéciaux :

— `^` lorsqu'il est utilisé comme un caractère (c'est à dire pas en début de `[]`) signifie un début de chaîne ;

— `\A` a le même sens (sauf en mode MULTILINE), et je le recommande de préférence à `^` qui est déjà pas mal surchargé ;

— `$` matche une fin de ligne ;

— `\Z` est voisin mais pas tout à fait identique.

Reportez-vous à la documentation pour le détails des différences. Attention aussi à entrer le `^` correctement, il vous faut le caractère ASCII et non un voisin dans la ménagerie Unicode.

```
In [43]: sample = 'abcd'

         for regexp in [ r'bc', r'\Aabc', r'^abc',
                         r'\Abc', r'^bc', r'bcd\Z',
                         r'bcd$', r'bc\Z', r'bc$' ]:
             match = re.match(regexp, sample)
             search = re.search(regexp, sample)
             print(f"{sample} / {regexp:5s} match → {nice(match):6s} search → {nice(search)
```

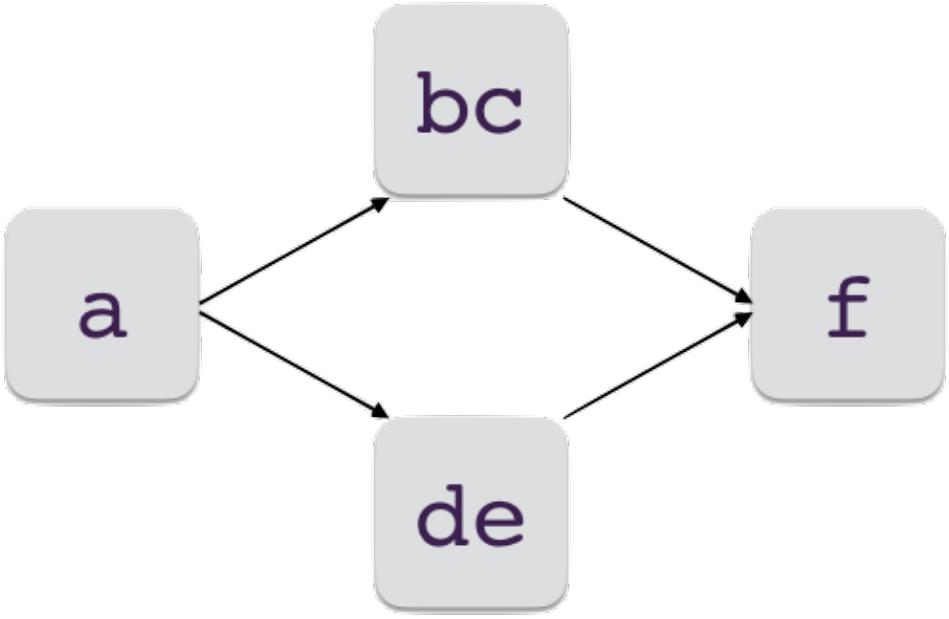
```
abcd / bc    match → no     search → Match!
abcd / \Aabc match → Match! search → Match!
abcd / ^abc  match → Match! search → Match!
abcd / \Abc  match → no     search → no
abcd / ^bc   match → no     search → no
abcd / bcd\Z match → no     search → Match!
abcd / bcd$  match → no     search → Match!
abcd / bc\Z  match → no     search → no
abcd / bc$   match → no     search → no
```

On a en effet bien le pattern `bc` dans la chaine en entrée, mais il n'est ni au début ni à la fin.

Parenthéser - (grouper)

Pour pouvoir faire des montages élaborés, il faut pouvoir parenthéser.

```
In [44]: # une parenthése dans une RE
         # pour mettre en ligne:
         # un début 'a',
         # un milieu 'bc' ou 'de'
         # et une fin 'f'
         regexp = "a(bc|de)f"
```




```
In [45]: for sample in ['abcf', 'adef',  'abef', 'abf']:
             match = re.match(regexp, sample)
             print(f"{sample:>4s} → {nice(match)}")

abcf → Match!
adef → Match!
abef → no
 abf → no
```

Les parenthèses jouent un rôle additionel de **groupe**, ce qui signifie qu'on **peut retrouver** le texte correspondant à l'expression régulière comprise dans les `()`. Par exemple, pour le premier match

```
In [46]: sample = 'abcf'
         match = re.match(regexp, sample)
         print(f"{sample}, {regexp} → {match.groups()}")

abcf, a(bc|de)f → ('bc',)
```

dans cet exemple, on n'a utilisé qu'un seul groupe `()`, et le morceau de chaîne qui correspond à ce groupe se trouve donc être le seul groupe retourné par `MatchObject.group`.

Compter les répétitions

Vous disposez des opérateurs suivants :

— `*` l'étoile qui signifie n'importe quel nombre, même nul, d'occurrences - par exemple, `(ab)*` pour indiquer `''` ou `'ab'` ou `'abab'` ou etc.,

— `+` le plus qui signifie au moins une occurrence - e.g. `(ab)+` pour `ab` ou `abab` ou `ababab` ou etc,

— `?` qui indique une option, c'est-à-dire 0 ou 1 occurence - autrement dit `(ab)?` matche `''` ou `ab`,

— `{n}` pour exactement n occurrences de `(ab)` - e.g. `(ab){3}` qui serait exactement équivalent à `ababab`,

— `{m,n}` entre m et n fois inclusivement.

```
In [47]: samples = [n*'ab' for n in [0, 1, 3, 4]] + ['baba']

         for regexp in ['(ab)*', '(ab)+', '(ab){3}', '(ab){3,4}']:
             # on ajoute \A \Z pour matcher toute la chaine
             line_regexp = r"\A{}\Z".format(regexp)
             for sample in samples:
                 match = re.match(line_regexp, sample)
                 print(f"{sample:>8s} / {line_regexp:14s} → {nice(match)}")
```

```
        / \A(ab)*\Z      → Match!
      ab / \A(ab)*\Z      → Match!
  ababab / \A(ab)*\Z      → Match!
ababbabab / \A(ab)*\Z     → Match!
    baba / \A(ab)*\Z      → no
        / \A(ab)+\Z      → no
      ab / \A(ab)+\Z      → Match!
  ababab / \A(ab)+\Z      → Match!
ababbabab / \A(ab)+\Z     → Match!
    baba / \A(ab)+\Z      → no
        / \A(ab){3}\Z    → no
      ab / \A(ab){3}\Z    → no
  ababab / \A(ab){3}\Z    → Match!
ababbabab / \A(ab){3}\Z   → no
    baba / \A(ab){3}\Z    → no
        / \A(ab){3,4}\Z  → no
      ab / \A(ab){3,4}\Z  → no
  ababab / \A(ab){3,4}\Z  → Match!
ababbabab / \A(ab){3,4}\Z → Match!
    baba / \A(ab){3,4}\Z  → no
```

Groupes et contraintes

Nous avons déjà vu un exemple de groupe nommé (voir `needle` plus haut), les opérateurs que l'on peut citer dans cette catégorie sont :

— `(...)` les parenthèses définissent un groupe anonyme,

— `(?P<name>...)` définit un groupe nommé,

— `(?:...)` permet de mettre des parenthèses mais sans créer un groupe, pour optimiser l'exécution puisqu'on n'a pas besoin de conserver les liens vers la chaîne d'entrée,

— `(?P=name)` qui ne matche que si l'on retrouve à cet endroit de l'entrée la même sous-chaîne que celle trouvée pour le groupe `name` en amont,

— enfin `(?=...)`, `(?!...)`et `(?<=...)` permettent des contraintes encore plus élaborées, nous vous laissons le soin d'expérimenter avec elles si vous êtes intéressés ; sachez toutefois que l'utilisation de telles constructions peut en théorie rendre l'interprétation de votre expression régulière beaucoup moins efficace.

Greedy *vs* non-greedy

Lorsqu'on stipule une répétition un nombre indéfini de fois, il se peut qu'il existe **plusieurs** façons de filtrer l'entrée avec l'expression régulière. Que ce soit avec `*`, ou `+`, ou `?`, l'algorithme va toujours essayer de trouver la **séquence la plus longue**, c'est pourquoi on qualifie l'approche de *greedy* - quelque chose comme glouton en français.

```
In [48]: # un fragment d'HTML
         line='<h1>Title</h1>'

         # si on cherche un texte quelconque entre crochets
         # c'est-à-dire l'expression régulière "<.*>"
         re_greedy = '<.*>'

         # on obtient ceci
         # on rappelle que group(0) montre la partie du fragment
         # HTML qui matche l'expression régulière
         match = re.match(re_greedy, line)
         match.group(0)
```

```
Out[48]: '<h1>Title</h1>'
```

Ça n'est pas forcément ce qu'on voulait faire, aussi on peut spécifier l'approche inverse, c'est-à-dire de trouver la **plus-petite** chaîne qui matche, dans une approche dite *non-greedy*, avec les opérateurs suivants :

— `*?` : `*` mais *non-greedy*,

— `+?` : `+` mais *non-greedy*,

— `??` : `?` mais *non-greedy*,

```
In [49]: # ici on va remplacer * par *? pour rendre l'opérateur * non-greedy
         re_non_greedy = re_greedy = '<.*?>'

         # mais on continue à cherche un texte entre <> naturellement
         # si bien que cette fois, on obtient
         match = re.match(re_non_greedy, line)
         match.group(0)
```

```
Out[49]: '<h1>'
```

S'agissant du traitement des fins de ligne

Il peut être utile, pour conclure cette présentation, de préciser un peu le comportement de la librairie vis-à-vis des fins de ligne.

Historiquement, les expressions régulières telles qu'on les trouve dans les librairies C, donc dans `sed`, `grep` et autre utilitaires Unix, sont associées au modèle mental où on filtre les entrées ligne par ligne.

Le module `re` en garde des traces, puisque

```
In [50]: # un exemple de traitement des 'newline'
         sample = """une entrée
         sur
         plusieurs
         lignes
         """

In [51]: match = re.compile("(.*)").match(sample)
         match.groups()

Out[51]: ('une entrée',)
```

Vous voyez donc que l'attrape-tout `'.'` en fait n'attrape pas le caractère de fin de ligne `\n`, puisque si c'était le cas et compte tenu du coté *greedy* de l'algorithme on devrait voir ici tout le contenu de `sample`. Il existe un *flag* `re.DOTALL` qui permet de faire de `.` un vrai attrape-tout qui capture aussi les *newline*

```
In [52]: match = re.compile("(.*)", flags=re.DOTALL).match(sample)
         match.groups()

Out[52]: ('une entrée\nsur\nplusieurs\nlignes\n',)
```

Cela dit, le caractère *newline* est par ailleurs considéré comme un caractère comme un autre, on peut le mentionner **dans une regexp** comme les autres. Voici quelques exemples pour illustrer tout ceci

```
In [53]: # sans mettre le flag unicode \w ne matche que l'ASCII
         match = re.compile("([\w ]*)").match(sample)
         match.groups()

Out[53]: ('une entrée',)

In [54]: # sans mettre le flag unicode \w ne matche que l'ASCII
         match = re.compile("([\w ]*)", flags=re.U).match(sample)
         match.groups()

Out[54]: ('une entrée',)

In [55]: # si on ajoute \n à la liste des caractères attendus
         # on obtient bien tout le contenu initial

         # attention ici il ne FAUT PAS utiliser un raw string,
         # car on veut vraiment écrire un newline dans la regexp

         match = re.compile("([\w \n]*)", flags=re.UNICODE).match(sample)
         match.groups()

Out[55]: ('une entrée\nsur\nplusieurs\nlignes\n',)
```

Conclusion

La mise au point d'expressions régulières est certes un peu exigeante, et demande pas mal de pratique, mais permet d'écrire en quelques lignes des fonctionnalités très puissantes, c'est un investissement très rentable :)

Je vous signale enfin l'existence de **sites web** qui évaluent une expression régulière **de manière interactive** et qui peuvent rendre la mise au point moins fastidieuse.

Je vous signale notamment https://pythex.org/, et il en existe beaucoup d'autres.

Pour en savoir plus

Pour ceux qui ont quelques rudiments de la théorie des langages, vous savez qu'on distingue en général

— l'**analyse lexicale**, qui découpe le texte en morceaux (qu'on appelle des *tokens*),

— et l'**analyse syntaxique** qui décrit pour simplifier à l'extrême l'ordre dans lequel on peut trouver les tokens.

Avec les expression régulières, on adresse le niveau de l'analyse lexicale. Pour l'analyse syntaxique, qui est franchement au delà des objectifs de ce cours, il existe de nombreuses alternatives, parmi lesquelles :

— `pyparsing`

— `PLY` (Python Lex-Yacc)

— `ANTLR` qui est un outil écrit en Java mais qui peut générer des parsers en python,

— ...

2.6 Expressions régulières

Nous vous proposons dans ce notebook quelques exercices sur les expressions régulières. Faisons quelques remarques avant de commencer :

— nous nous concentrons sur l'écriture de l'expression régulière en elle-même, et pas sur l'utilisation de la bibliothèque ;
— en particulier, tous les exercices font appel à `re.match` entre votre *regexp* et une liste de chaînes d'entrée qui servent de jeux de test.

Liens utiles Pour travailler sur ces exercices, il pourra être profitable d'avoir sous la main :

— la documentation officielle ;
— et cet outil interactif sur https://pythex.org/ qui permet d'avoir un retour presque immédiat, et donc d'accélérer la mise au point.

2.6.1 Exercice - niveau intermédiaire (1)

Identificateurs Python

```
In [ ]: # évaluez cette cellule pour charger l'exercice
        from regexp_pythonid import exo_pythonid
```

On vous demande d'écrire une expression régulière qui décrit les noms de variable en Python. Pour cet exercice on se concentre sur les caractères ASCII. On exclut donc les noms de variables qui pourraient contenir des caractères exotiques comme les caractères accentués ou autres lettres grecques.

Il s'agit donc de reconnaître toutes les chaînes qui commencent par une lettre ou un _, suivi de lettres, chiffres ou _.

```
In [ ]: # quelques exemples de résultat attendus
        exo_pythonid.example()
```

```
In [ ]: # à vous de jouer: écrivez ici
        # sous forme de chaîne votre expression régulière

        regexp_pythonid = r"<votre_regexp>"
```

```
In [ ]: # évaluez cette cellule pour valider votre code
        exo_pythonid.correction(regexp_pythonid)
```

2.6.2 Exercice - niveau intermédiaire (2)

Lignes avec nom et prénom

```
In [ ]: # pour charger l'exercice
        from corrections.regexp_agenda import exo_agenda
```

On veut reconnaître dans un fichier toutes les lignes qui contiennent un nom et un prénom.

```
In [ ]: exo_agenda.example()
```

Plus précisément, on cherche les chaînes qui :

— commencent par une suite - possiblement vide - de caractères alphanumériques (vous pouvez utiliser `\w`) ou tiret haut (-) qui constitue le prénom ;
— contiennent ensuite comme séparateur le caractère 'deux-points' : ;
— contiennent ensuite une suite - cette fois jamais vide - de caractères alphanumériques, qui constitue le nom ;
— et enfin contiennent un deuxième caractère : mais optionnellement seulement.

On vous demande de construire une expression régulière qui définit les deux groupes `nom` et `prenom`, et qui rejette les lignes qui ne satisfont pas ces critères.

```
In [ ]: # entrez votre regexp ici
        # il faudra la faire terminer par \Z
        # regardez ce qui se passe si vous ne le faites pas

        regexp_agenda = r"<votre regexp>\Z"
```

```
In [ ]: # évaluez cette cellule pour valider votre code
        exo_agenda.correction(regexp_agenda)
```

2.6.3 Exercice - niveau intermédiaire (3)

Numéros de téléphone

```
In [ ]: # pour charger l'exercice
        from corrections.regexp_phone import exo_phone
```

Cette fois on veut reconnaître des numéros de téléphone français, qui peuvent être :

— soit au format contenant 10 chiffres dont le premier est un `0` ;
— soit un format international commençant par `+33` suivie de 9 chiffres.

Dans tous les cas on veut trouver dans le groupe 'number' les 9 chiffres vraiment significatifs, comme ceci :

```
In [ ]: exo_phone.example()
```

```
In [ ]: # votre regexp
        # à nouveau il faut terminer la regexp par \Z
        regexp_phone = r"<votre regexp>\Z"
```

```
In [ ]: # évaluez cette cellule pour valider votre code
        exo_phone.correction(regexp_phone)
```

2.6.4 Exercice - niveau avancé

Vu comment sont conçus les exercices, vous ne pouvez pas passer à `re.compile` un drapeau comme `re.IGNORECASE` ou autre ; sachez cependant que vous pouvez ***embarquer* ces drapeaux dans la *regexp*** elle-même ; par exemple pour rendre la regexp insensible à la casse de caractères, au lieu d'appeler `re.compile` avec le flag `re.I`, vous pouvez utiliser `(?i)` comme ceci :

```
In [ ]: import re
```

```
In [ ]: # on peut embarquer les flags comme IGNORECASE
        # directement dans la regexp
        # c'est équivalent de faire ceci

        re_obj = re.compile("abc", flags=re.IGNORECASE)
        re_obj.match("ABC").group(0)
```

```
In [ ]: # ou cela

        re.match("(?i)abc","ABC").group(0)

In [ ]: # les flags comme (?i) doivent apparaître
        # en premier dans la regexp
        re.match("abc(?i)","ABC").group(0)
```

Pour plus de précisions sur ce trait, que nous avons laissé de côté dans le complément pour ne pas trop l'alourdir, voyez la documentation sur les expressions régulières et cherchez la première occurrence de `iLmsux`.

Décortiquer une URL

On vous demande d'écrire une expression régulière qui permette d'analyser des URLs. Voici les conventions que nous avons adoptées pour l'exercice :

- la chaîne contient les parties suivantes :
  - `<protocol>://<location>/<path>`;
- l'URL commence par le nom d'un protocole qui doit être parmi `http`, `https`, `ftp`, `ssh`;
- le nom du protocole peut contenir de manière indifférente des minuscules ou des majuscules;
- ensuite doit venir la séquence `://`;
- ensuite on va trouver une chaîne `<location>` qui contient :
  - potentiellement un nom d'utilisateur, et s'il est présent, potentiellement un mot de passe;
  - obligatoirement un nom de `hostname`;
  - potentiellement un numéro de port;
- lorsque les 4 parties sont présentes dans `<location>`, cela se présente comme ceci :
  - `<location> = <user>:<password>@<hostname>:<port>`;
- si l'on note entre crochets les parties optionnelles, cela donne :
  - `<location> = [<user>[:<password>]@]<hostname>[:<port>]`;
- le champ `<user>` ne peut contenir que des caractères alphanumériques; si le `@` est présent le champ `<user>` ne peut pas être vide;
- le champ `<password>` peut contenir tout sauf un `:` et de même, si le `:` est présent le champ `<password>` ne peut pas être vide;
- le champ `<hostname>` peut contenir un suite non-vide de caractères alphanumériques, underscores, ou `.`;
- le champ `<port>` ne contient que des chiffres, et il est non vide si le `:` est spécifié;
- le champ `<path>` peut être vide.

Enfin, vous devez définir les groupes `proto`, `user`, `password`, `hostname`, `port` et `path` qui sont utilisés pour vérifier votre résultat. Dans la case `Résultat attendu`, vous trouverez soit `None` si la regexp ne filtre pas l'intégralité de l'entrée, ou bien une liste ordonnée de tuples qui donnent la valeur de ces groupes; vous n'avez rien à faire pour construire ces tuples, c'est l'exercice qui s'en occupe.

```
In [ ]: # pour charger l'exercice
        from corrections.regexp_url import exo_url
```

```
In [ ]: # exemples du résultat attendu
        exo_url.example()

In [ ]: # n'hésitez pas à construire votre regexp petit à petit

        regexp_url = "<votre_regexp>"

In [ ]: exo_url.correction(regexp_url)
```

2.7 Les slices en Python

2.7.1 Complément - niveau basique

Ce support de cours reprend les notions de *slicing* vues dans la vidéo.

Nous allons illustrer les slices sur la chaîne suivante, rappelez-vous toutefois que ce mécanisme fonctionne avec toutes les séquences que l'on verra plus tard, comme les listes ou les tuples.

```
In [1]: chaine = "abcdefghijklmnopqrstuvwxyz"
        print(chaine)
```

```
abcdefghijklmnopqrstuvwxyz
```

Slice sans pas

On a vu en cours qu'une slice permet de désigner toute une plage d'éléments d'une séquence. Ainsi on peut écrire :

```
In [2]: chaine[2:6]
```

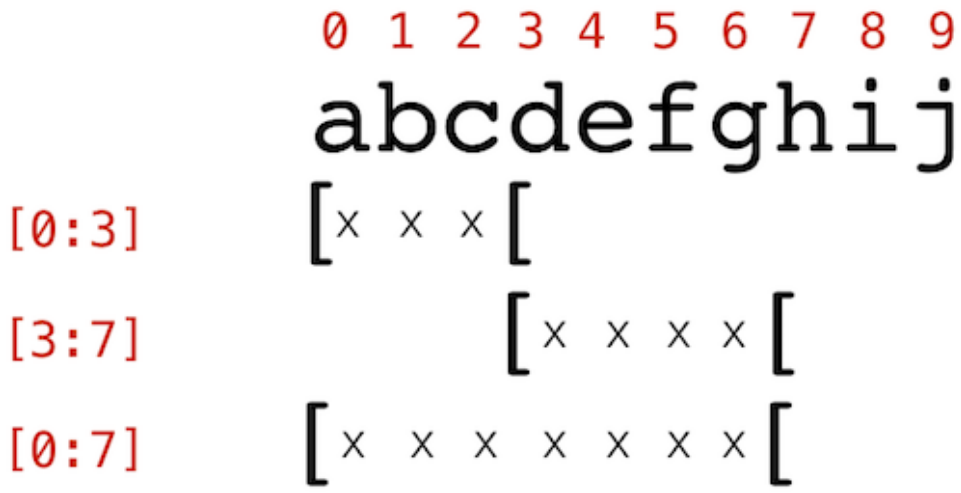
```
Out[2]: 'cdef'
```

Conventions de début et fin

Les débutants ont parfois du mal avec les bornes. Il faut se souvenir que :

— les indices **commencent** comme toujours **à zéro** ;
— le premier indice `debut` est **inclus** ;
— le second indice `fin` est **exclu** ;
— on obtient en tout `fin-debut` items dans le résultat.

Ainsi, ci-dessus, le résultat contient `6 - 2 = 4` éléments.
Pour vous aider à vous souvenir des conventions de début et de fin, souvenez-vous qu'on veut pouvoir facilement juxtaposer deux slices qui ont une borne commune.
C'est-à-dire qu'avec :



début et fin

```
In [3]: # chaine[a:b] + chaine[b:c] == chaine[a:c]
        chaine[0:3] + chaine[3:7] == chaine[0:7]
```

```
Out[3]: True
```

Bornes omises

On peut omettre une borne :

```
In [4]: # si on omet la première borne, cela signifie que
        # la slice commence au début de l'objet
        chaine[:6]
```

```
Out[4]: 'abcdef'
```

```
In [5]: # et bien entendu c'est la même chose si on omet la deuxième borne
        chaine[24:]
```

```
Out[5]: 'yz'
```

```
In [6]: # ou même omettre les deux bornes, auquel cas on
        # fait une copie de l'objet - on y reviendra plus tard
        chaine[:]
```

```
Out[6]: 'abcdefghijklmnopqrstuvwxyz'
```

Indices négatifs

On peut utiliser des indices négatifs pour compter à partir de la fin :

```
In [7]: chaine[3:-3]
```

```
Out[7]: 'defghijklmnopqrstuvw'
```

```
In [8]: chaine[-3:]
```

```
Out[8]: 'xyz'
```

Slice avec pas

Il est également possible de préciser un *pas*, de façon à ne choisir par exemple, dans la plage donnée, qu'un élément sur deux :

```
In [9]: # le pas est précisé après un deuxième deux-points (:)
        # ici on va choisir un caractère sur deux dans la plage [3:-3]
        chaine[3:-3:2]
```

```
Out[9]: 'dfhjlnprtv'
```

Comme on le devine, le troisième élément de la slice, ici `2`, détermine le pas. On ne retient donc, dans la chaîne `defghi...` que `d`, puis `f`, et ainsi de suite.

On peut préciser du coup la borne de fin (ici `-3`) avec un peu de liberté, puisqu'ici on obtiendrait un résultat identique avec `-4`.

```
In [10]: chaine[3:-4:2]
```

```
Out[10]: 'dfhjlnprtv'
```

Pas négatif

Il est même possible de spécifier un pas négatif. Dans ce cas, de manière un peu contre-intuitive, il faut préciser un début (le premier indice de la slice) qui soit *plus à droite* que la fin (le second indice).

Pour prendre un exemple, comme l'élément d'indice `-3`, c'est-à-dire `x`, est plus à droite que l'élément d'indice `3`, c'est-à-dire `d`, évidemment si on ne précisait pas le pas (qui revient à choisir un pas égal à `1`), on obtiendrait une liste vide :

```
In [11]: chaine[-3:3]
```

```
Out[11]: ''
```

Si maintenant on précise un pas négatif, on obtient cette fois :

```
In [12]: chaine[-3:3:-2]
```

```
Out[12]: 'xvtrpnljhf'
```

Conclusion

À nouveau, souvenez-vous que tous ces mécanismes fonctionnent avec de nombreux autres types que les chaînes de caractères. En voici deux exemples qui anticipent tous les deux sur la suite, mais qui devraient illustrer les vastes possibilités qui sont offertes avec les slices.

Listes Par exemple sur les listes :

```
In [13]: liste = [0, 2, 4, 8, 16, 32, 64, 128]
         liste
```

```
Out[13]: [0, 2, 4, 8, 16, 32, 64, 128]
```

```
In [14]: liste[-1:1:-2]
```

```
Out[14]: [128, 32, 8]
```

Et même ceci, qui peut être déroutant. Nous reviendrons dessus.

```
In [15]: liste[2:4] = [100, 200, 300, 400, 500]
         liste
```

```
Out[15]: [0, 2, 100, 200, 300, 400, 500, 16, 32, 64, 128]
```

2.7.2 Complément - niveau avancé

`numpy` La bibliothèque `numpy` permet de manipuler des tableaux ou des matrices. En anticipant (beaucoup) sur son usage que nous reverrons bien entendu en détail, voici un aperçu de ce que l'on peut faire avec des slices sur des objets `numpy` :

```
In [16]: # ces deux premières cellules sont à admettre
         # on construit un tableau ligne
         import numpy as np

         un_cinq = np.array([1, 2, 3, 4, 5])
         un_cinq
```

```
Out[16]: array([1, 2, 3, 4, 5])

In [17]: # ces deux premières cellules sont à admettre
         # on le combine avec lui-même - et en utilisant une slice un peu magique
         # pour former un tableau carré 5x5

         array = 10 * un_cinq[:, np.newaxis] + un_cinq
         array

Out[17]: array([[11, 12, 13, 14, 15],
                [21, 22, 23, 24, 25],
                [31, 32, 33, 34, 35],
                [41, 42, 43, 44, 45],
                [51, 52, 53, 54, 55]])
```

Sur ce tableau de taille 5x5, nous pouvons aussi faire du slicing et extraire le sous-tableau 3x3 au centre :

```
In [18]: centre = array[1:4, 1:4]
         centre

Out[18]: array([[22, 23, 24],
                [32, 33, 34],
                [42, 43, 44]])
```

On peut bien sûr également utiliser un pas :

```
In [19]: coins = array[::4, ::4]
         coins

Out[19]: array([[11, 15],
                [51, 55]])
```

Ou bien retourner complètement dans une direction :

```
In [20]: tete_en_bas = array[::-1,:]
         tete_en_bas

Out[20]: array([[51, 52, 53, 54, 55],
                [41, 42, 43, 44, 45],
                [31, 32, 33, 34, 35],
                [21, 22, 23, 24, 25],
                [11, 12, 13, 14, 15]])
```

2.8 Méthodes spécifiques aux listes

2.8.1 Complément - niveau basique

Voici quelques unes des méthodes disponibles sur le type `list`.

Trouver l'information

Pour commencer, rappelons comment retrouver la liste des méthodes définies sur le type `list` :

```
In [1]: help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __iadd__(self, value, /)
 |      Implement self+=value.
 |
 |  __imul__(self, value, /)
 |      Implement self*=value.
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
```

```
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.n
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(...)
|      L.__reversed__() -- return a reverse iterator over the list
|
|  __rmul__(self, value, /)
|      Return self*value.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(...)
|      L.__sizeof__() -- size of L in memory, in bytes
|
|  append(...)
|      L.append(object) -> None -- append object to end
|
|  clear(...)
|      L.clear() -> None -- remove all items from L
|
|  copy(...)
|      L.copy() -> list -- a shallow copy of L
|
|  count(...)
|      L.count(value) -> integer -- return number of occurrences of value
|
|  extend(...)
|      L.extend(iterable) -> None -- extend list by appending elements from the iterable
```

```
 |
 |  index(...)
 |      L.index(value, [start, [stop]]) -> integer -- return first index of value.
 |      Raises ValueError if the value is not present.
 |
 |  insert(...)
 |      L.insert(index, object) -- insert object before index
 |
 |  pop(...)
 |      L.pop([index]) -> item -- remove and return item at index (default last).
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(...)
 |      L.remove(value) -> None -- remove first occurrence of value.
 |      Raises ValueError if the value is not present.
 |
 |  reverse(...)
 |      L.reverse() -- reverse *IN PLACE*
 |
 |  sort(...)
 |      L.sort(key=None, reverse=False) -> None -- stable sort *IN PLACE*
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```

Ignorez les méthodes dont le nom commence et termine par `__` (nous parlerons de ceci en semaine 6), vous trouvez alors les méthodes utiles listées entre `append` et `sort`.

Certaines de ces méthodes ont été vues dans la vidéo sur les séquences, c'est le cas notamment de `count` et `index`.

Nous allons à présent décrire les autres, partiellement et brièvement. Un autre complément décrit la méthode `sort`. Reportez-vous au lien donné en fin de notebook pour obtenir une information plus complète.

Donnons-nous pour commencer une liste témoin :

```
In [2]: liste = [0, 1, 2, 3]
        print('liste', liste)
```

```
liste [0, 1, 2, 3]
```

Avertissements :

- soyez bien attentifs au nombre de fois où vous exécutez les cellules de ce notebook ;
- par exemple une liste renversée deux fois peut donner l'impression que `reverse` ne marche pas ;
- n'hésitez pas à utiliser le menu *Cell -> Run All* pour réexécuter en une seule fois le notebook entier.

append

La méthode `append` permet d'ajouter **un élément** à la fin d'une liste :

```
In [3]: liste.append('ap')
        print('liste', liste)

liste [0, 1, 2, 3, 'ap']
```

extend

La méthode `extend` réalise la même opération, mais avec **tous les éléments** de la liste qu'on lui passe en argument :

```
In [4]: liste2 = ['ex1', 'ex2']
        liste.extend(liste2)
        print('liste', liste)

liste [0, 1, 2, 3, 'ap', 'ex1', 'ex2']
```

append *vs* +

Ces deux méthodes `append` et `extend` sont donc assez voisines ; avant de voir d'autres méthodes de `list`, prenons un peu le temps de comparer leur comportement avec l'addition + de liste. L'élément clé ici, on l'a déjà vu dans la vidéo, est que la liste est un objet **mutable**. `append` et `extend` **modifient** la liste sur laquelle elles travaillent, alors que l'addition **crée un nouvel objet**.

```
In [5]: # pour créer une liste avec les n premiers entiers, on utilise
        # la fonction built-in range(), que l'on convertit en liste
        # on aura l'occasion d'y revenir
        a1 = list(range(3))
        print(a1)

[0, 1, 2]
```

```
In [6]: a2 = list(range(10, 13))
        print(a2)

[10, 11, 12]
```

```
In [7]: # le fait d'utiliser + crée une nouvelle liste
        a3 = a1 + a2

In [8]: # si bien que maintenant on a trois objets différents
        print('a1', a1)
        print('a2', a2)
        print('a3', a3)
```

```
a1 [0, 1, 2]
a2 [10, 11, 12]
a3 [0, 1, 2, 10, 11, 12]
```

Comme on le voit, après une addition, les deux termes de l'addition sont inchangés. Pour bien comprendre, voyons exactement le même scénario sous pythontutor :

```
In [ ]: %load_ext ipythontutor
```

Note : une fois que vous avez évalué la cellule avec `%%ipythontutor`, vous devez cliquer sur le bouton `Forward` pour voir pas à pas le comportement du programme.

```
In [ ]: %%ipythontutor height=230 ratio=0.7
        a1 = list(range(3))
        a2 = list(range(10, 13))
        a3 = a1 + a2
```

Alors que si on avait utilisé `extend`, on aurait obtenu ceci :

```
In [ ]: %%ipythontutor height=200 ratio=0.75
        e1 = list(range(3))
        e2 = list(range(10, 13))
        e3 = e1.extend(e2)
```

Ici on tire profit du fait que la liste est un objet mutable : `extend` **modifie** l'objet sur lequel on l'appelle (ici `e1`). Dans ce scénario on ne crée en tout que deux objets, et du coup il est inutile pour extend de renvoyer quoi que ce soit, et c'est pourquoi `e3` ici vaut None.

C'est pour cette raison que :

— l'addition est disponible sur tous les types séquences - on peut toujours réaliser l'addition puisqu'on crée un nouvel objet pour stocker le résultat de l'addition ;
— mais `append` et `extend` ne sont par exemple **pas disponibles** sur les chaînes de caractères, qui sont **immuables** - si `e1` était une chaîne, on ne pourrait pas la modifier pour lui ajouter des éléments.

insert

Reprenons notre inventaire des méthodes de `list`, et pour cela rappelons nous le contenu de la variable `liste` :

```
In [9]: liste
```

```
Out[9]: [0, 1, 2, 3, 'ap', 'ex1', 'ex2']
```

La méthode `insert` permet, comme le nom le suggère, d'insérer un élément à une certaine position ; comme toujours les indices commencent à zéro et donc :

```
In [10]: # insérer à l'index 2
         liste.insert(2, '1 bis')
         print('liste', liste)
```

```
liste [0, 1, '1 bis', 2, 3, 'ap', 'ex1', 'ex2']
```

On peut remarquer qu'un résultat analogue peut être obtenu avec une affectation de slice ; par exemple pour insérer au rang 5 (i.e. avant `ap`), on pourrait aussi bien faire :

```
In [11]: liste[5:5] = ['3 bis']
         print('liste', liste)
```

```
liste [0, 1, '1 bis', 2, 3, '3 bis', 'ap', 'ex1', 'ex2']
```

remove

La méthode `remove` détruit la **première occurrence** d'un objet dans la liste :

```
In [12]: liste.remove(3)
         print('liste', liste)
```

```
liste [0, 1, '1 bis', 2, '3 bis', 'ap', 'ex1', 'ex2']
```

pop

La méthode `pop` prend en argument un indice ; elle permet d'extraire l'élément à cet indice. En un seul appel on obtient la valeur de l'élément et on l'enlève de la liste :

```
In [13]: popped = liste.pop(0)
         print('popped', popped, 'liste', liste)
```

```
popped 0 liste [1, '1 bis', 2, '3 bis', 'ap', 'ex1', 'ex2']
```

Si l'indice n'est pas précisé, c'est le dernier élément de la liste qui est visé :

```
In [14]: popped = liste.pop()
         print('popped', popped, 'liste', liste)
```

```
popped ex2 liste [1, '1 bis', 2, '3 bis', 'ap', 'ex1']
```

reverse

Enfin `reverse` renverse la liste, le premier élément devient le dernier :

```
In [15]: liste.reverse()
         print('liste', liste)
```

```
liste ['ex1', 'ap', '3 bis', 2, '1 bis', 1]
```

On peut remarquer ici que le résultat se rapproche de ce qu'on peut obtenir avec une opération de slicing comme ceci :

```
In [16]: liste2 = liste[::-1]
         print('liste2', liste2)
```

```
liste2 [1, '1 bis', 2, '3 bis', 'ap', 'ex1']
```

À la différence toutefois qu'avec le slicing c'est une copie de la liste initiale qui est retournée, la liste de départ quant à elle n'est pas modifiée.

Pour en savoir plus

https://docs.python.org/3/tutorial/datastructures.html#more-on-lists

Note spécifique aux notebooks

`help` **avec** `?` Je vous signale en passant que dans un notebook vous pouvez obtenir de l'aide avec un point d'interrogation `?` inséré avant ou après un symbole. Par exemple pour obtenir des précisions sur la méthode `list.pop`, on peut faire soit :

```
In [17]: # fonctionne dans tous les environnements Python
         help(list.pop)
```

```
Help on method_descriptor:

pop(...)
    L.pop([index]) -> item -- remove and return item at index (default last).
    Raises IndexError if list is empty or index is out of range.
```

```
In [18]: # spécifique aux notebooks
         # l'affichage obtenu est légèrement différent
         # tapez la touche 'Esc' - ou cliquez la petite croix
         # pour faire disparaitre le dialogue qui apparaît en bas
         list.pop?
```

Complétion avec `Tab`

Dans un notebook vous avez aussi la complétion; si vous tapez - dans une cellule de code - le début d'un symbole connu dans l'environnement :

```
In [ ]: # placez votre curseur à la fin de la ligne après 'li'
        # et appuyez sur la touche 'Tab'
        li
```

Vous voyez apparaître un dialogue avec les noms connus qui commencent par `li` ; utilisez les flèches pour choisir, et 'Return' pour sélectionner.

2.9 Objets mutables et objets immuables

2.9.1 Complément - niveau basique

Les chaînes sont des objets immuables

Voici un exemple d'un fragment de code qui illustre le caractère immuable des chaînes de caractères. Nous l'exécutons sous pythontutor, afin de bien illustrer les relations entre variables et objets.

```
In [ ]: # il vous faut charger cette cellule
        # pour pouvoir utiliser les suivantes
        %load_ext ipythontutor
```

Note : une fois que vous avez évalué la cellule avec `%%ipythontutor`, vous devez cliquer sur le bouton `Forward` pour voir pas à pas le comportement du programme.

Le scénario est très simple, on crée deux variables `s1` et `s2` vers le même objet `'abc'`, puis on fait une opération `+=` sur la variable `s1`.

Comme l'objet est une chaîne, il est donc immuable, on ne **peut pas modifier l'objet** directement ; pour obtenir l'effet recherché (à savoir que `s1` s'allonge de `'def'`), Python **crée un deuxième objet**, comme on le voit bien sous pythontutor :

```
In [ ]: %%ipythontutor heapPrimitives=true
        # deux variables vers le même objet
        s1 = 'abc'
        s2 = s1
        # on essaie de modifier l'objet
        s1 += 'def'
        # pensez à cliquer sur `Forward`
```

Les listes sont des objets mutables

Voici ce qu'on obtient par contraste pour le même scénario mais qui cette fois utilise des listes, qui sont des objets mutables :

```
In [ ]: %%ipythontutor heapPrimitives=true ratio=0.8
        # deux variables vers le même objet
        liste1 = ['a', 'b', 'c']
        liste2 = liste1
        # on modifie l'objet
        liste1 += ['d', 'e', 'f']
        # pensez à cliquer sur `Forward`
```

Conclusion

Ce comportement n'est pas propre à l'usage de l'opérateur `+=` - que pour cette raison d'ailleurs nous avons tendance à déconseiller.

Les objets mutables et immuables ont par essence un comportement différent, il est très important d'avoir ceci présent à l'esprit.

Nous aurons notamment l'occasion d'approfondir cela dans la séquence consacrée aux références partagées, en semaine 3.

2.10 Tris de listes

2.10.1 Complément - niveau basique

Python fournit une méthode standard pour trier une liste, qui s'appelle, sans grande surprise, `sort`.

La méthode `sort`

Voyons comment se comporte `sort` sur un exemple simple :

```
In [1]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
        print('avant tri', liste)
        liste.sort()
        print('apres tri', liste)

avant tri [8, 7, 4, 3, 2, 9, 1, 5, 6]
apres tri [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

On retrouve ici, avec l'instruction `liste.sort()` un cas d'appel de méthode (ici `sort`) sur un objet (ici `liste`), comme on l'avait vu dans la vidéo sur la notion d'objet.

La première chose à remarquer est que la liste d'entrée a été modifiée, on dit "en place", ou encore "par effet de bord". Voyons cela sous pythontutor :

```
In [ ]: %load_ext ipythontutor

In [ ]: %%ipythontutor height=200 ratio=0.8
        liste = [3, 2, 9, 1]
        liste.sort()
```

On aurait pu imaginer que la liste d'entrée soit restée inchangée, et que la méthode de tri renvoie une copie triée de la liste, ce n'est pas le choix qui a été fait, cela permet d'économiser des allocations mémoire autant que possible et d'accélérer sensiblement le tri.

La fonction `sorted`

Si vous avez besoin de faire le tri sur une copie de votre liste, la fonction `sorted` vous permet de le faire :

```
In [ ]: %%ipythontutor height=200 ratio=0.8
        liste1 = [3, 2, 9, 1]
        liste2 = sorted(liste1)
```

Tri décroissant

Revenons à la méthode `sort` et aux tris *en place*. Par défaut la liste est triée par ordre croissant, si au contraire vous voulez l'ordre décroissant, faites comme ceci :

```
In [2]: liste = [8, 7, 4, 3, 2, 9, 1, 5, 6]
        print('avant tri', liste)
        liste.sort(reverse=True)
        print('apres tri décroissant', liste)
```

```
avant tri [8, 7, 4, 3, 2, 9, 1, 5, 6]
apres tri décroissant [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Nous n'avons pas encore vu à quoi correspond cette formule `reverse=True` dans l'appel à la méthode - ceci sera approfondi dans le chapitre sur les appels de fonction - mais dans l'immédiat vous pouvez utiliser cette technique telle quelle.

Chaînes de caractères

Cette technique fonctionne très bien sur tous les types numériques (enfin, à l'exception des complexes ; en guise d'exercice, pourquoi ?), ainsi que sur les chaînes de caractères :

```
In [3]: liste = ['spam', 'egg', 'bacon', 'beef']
        liste.sort()
        print('après tri', liste)

après tri ['bacon', 'beef', 'egg', 'spam']
```

Comme on s'y attend, il s'agit cette fois d'un **tri lexicographique**, dérivé de l'ordre sur les caractères. Autrement dit, c'est l'ordre du dictionnaire. Il faut souligner toutefois, pour les personnes n'ayant jamais été exposées à l'informatique, que cet ordre, quoique déterministe, est arbitraire en dehors des lettres de l'alphabet.

Ainsi par exemple :

```
In [4]: # deux caractères minuscules se comparent
        # comme on s'y attend
        'a' < 'z'

Out[4]: True
```

Bon, mais par contre :

```
In [5]: # si l'un est en minuscule et l'autre en majuscule,
        # ce n'est plus le cas
        'Z' < 'a'

Out[5]: True
```

Ce qui à son tour explique ceci :

```
In [6]: # la conséquence de 'Z' < 'a', c'est que
        liste = ['abc', 'Zoo']
        liste.sort()
        print(liste)

['Zoo', 'abc']
```

Et lorsque les chaînes contiennent des espaces ou autres ponctuations, le résultat du tri peut paraître surprenant :

```
In [7]: # attention ici notre premiere chaîne commence par une espace
        # et le caractère 'Espace' est plus petit
        # que tous les autres caractères imprimables
        liste = [' zoo', 'ane']
        liste.sort()
        print(liste)
```

```
[' zoo', 'ane']
```

À suivre

Il est possible de définir soi-même le critère à utiliser pour trier une liste, et nous verrons cela bientôt, une fois que nous aurons introduit la notion de fonction.

2.11 Indentations en Python

2.11.1 Complément - niveau basique

Imbrications

Nous l'avons vu dans la vidéo, la pratique la plus courante est d'utiliser systématiquement une indentation de 4 espaces :

```
In [1]: # la convention la plus généralement utilisée
        # consiste à utiliser une indentation de 4 espaces
        if 'g' in 'egg':
            print('OUI')
        else:
            print('NON')
```

```
OUI
```

Voyons tout de suite comment on pourrait écrire plusieurs tests imbriqués :

```
In [2]: entree = 'spam'

        # pour imbriquer il suffit d'indenter de 8 espaces
        if 'a' in entree:
            if 'b' in entree:
                cas11 = True
                print('a et b')
            else:
                cas12 = True
                print('a mais pas b')
        else:
            if 'b' in entree:
                cas21 = True
                print('b mais pas a')
            else:
                cas22 = True
                print('ni a ni b')
```

```
a mais pas b
```

Dans cette construction assez simple, remarquez bien **les deux points ' : '** à chaque début de bloc, c'est-à-dire à chaque fin de ligne `if` ou `else`.

Cette façon d'organiser le code peut paraître très étrange, notamment aux gens habitués à un autre langage de programmation, puisqu'en général les syntaxes des langages sont conçues de manière à être insensibles aux espaces et à la présentation.

Comme vous le constaterez à l'usage cependant, une fois qu'on s'y est habitué cette pratique est très agréable, une fois qu'on a écrit la dernière ligne du code, on n'a pas à réfléchir à refermer le bon nombre d'accolades ou de *end*.

Par ailleurs, comme pour tous les langages, votre éditeur favori connaît cette syntaxe et va vous aider à respecter la règle des 4 caractères. Nous ne pouvons pas publier ici une liste des commandes disponibles par éditeur, nous vous invitons le cas échéant à échanger entre vous sur le forum pour partager les recettes que vous utilisez avec votre éditeur / environnement de programmation favori.

2.11.2 Complément - niveau intermédiaire

Espaces *vs* tabulations

Version courte

Il nous faut par contre donner quelques détails sur un problème que l'on rencontre fréquemment sur du code partagé entre plusieurs personnes quand celles-ci utilisent des environnement différents.

Pour faire court, ce problème est **susceptible d'apparaître dès qu'on utilise des tabulations**, plutôt que des espaces, pour implémenter les indentations. Aussi, le message à retenir ici est **de ne jamais utiliser de tabulations dans votre code Python**. Tout bon éditeur Python devrait faire cela par défaut.

Version longue

En version longue, il existe un code ASCII pour un caractère qui s'appelle *Tabulation* (alias Control-i, qu'on note aussi `^I`); l'interprétation de ce caractère n'étant pas clairement spécifiée, il arrive qu'on se retrouve dans une situation comme la suivante.

Bernard utilise l'éditeur `vim`; sous cet éditeur il lui est possible de mettre des tabulations dans son code, et de choisir la valeur de ces tabulations. Aussi il va dans les préférences de `vim`, choisit Tabulation=4, et écrit un programme qu'il voit comme ceci :

```
In [3]: if 'a' in entree:
            if 'b' in entree:
                cas11 = True
                print('a et b')
            else:
                cas12 = True
                print('a mais pas b')

a mais pas b
```

Sauf qu'en fait, il a mis un mélange de tabulations et d'espaces, et en fait le fichier contient (avec `^I` pour tabulation) :

```
if 'a' in entree:
^Iif 'b' in entree:
^I^Icas11 = True
    ^Iprint('a et b')
^Ielse:
^I^Icas12 = True
    ^Iprint('a mais pas b')
```

Remarquez le mélange de Tabulations et d'espaces dans les deux lignes avec `print`. Bernard envoie son code à Alice qui utilise `emacs`. Dans son environnement, `emacs` affiche une tabulation comme 8 caractères. Du coup Alice "voit" le code suivant :

```
if 'a' in entree:
        if 'b' in entree:
                cas11 = True
            print('a et b')
        else:
                cas12 = True
            print('a mais pas b')
```

Bref, c'est la confusion la plus totale. Aussi répétons-le, **n'utilisez jamais de tabulations dans votre code Python**.

Ce qui ne veut pas dire qu'il ne faut pas utiliser la touche `Tab` avec votre éditeur - au contraire, c'est une touche très utilisée - mais faites bien la différence entre le fait d'appuyer sur la touche `Tab` et le fait que le fichier sauvé sur disque contient effectivement un caractère tabulation. Votre éditeur favori propose très certainement une option permettant de faire les remplacements idoines pour ne pas écrire de tabulation dans vos fichiers, tout en vous permettant d'indenter votre code avec la touche `Tab`.

Signalons enfin que Python 3 est plus restrictif que Python 2 à cet égard, et interdit de mélanger des espaces et des tabulations sur une même ligne. Ce qui n'enlève rien à notre recommandation.

2.11.3 Complément - niveau avancé

Vous pouvez trouver du code qui ne respecte pas la convention des 4 caractères.

Version courte

En version courte : **Utilisez toujours des indentations de 4 espaces**.

Version longue

En version longue, et pour les curieux : Python **n'impose pas** que les indentations soient de 4 caractères. Aussi vous pouvez rencontrer un code qui ne respecte pas cette convention, et il nous faut, pour être tout à fait précis sur ce que Python accepte ou non, préciser ce qui est réellement requis par Python.

La règle utilisée pour analyser votre code, c'est que toutes les instructions **dans un même bloc** soient présentées avec le même niveau d'indentation. Si deux lignes successives - modulo les blocs imbriqués - ont la même indentation, elles sont dans le même bloc.

Voyons quelques exemples. Tout d'abord le code suivant est **légal**, quoique, redisons-le pour la dernière fois, **pas du tout recommandé** :

```
In [4]: # code accepté mais pas du tout recommandé
        if 'a' in 'pas du tout recommande':
          succes = True
          print('OUI')
```

```
        else:
              print('NON')
```

```
OUI
```

En effet, les deux blocs (après `if` et après `else`) sont des blocs distincts, ils sont libres d'utiliser deux indentations différentes (ici 2 et 6).

Par contre la construction ci-dessous n'est pas légale :

```
In [5]: # ceci n'est pas correct et est rejeté par Python
        if 'a' in entree:
            if 'b' in entree:
                cas11 = True
                print('a et b')
              else:
                cas12 = True
                print('a mais pas b')
```

```
  File "<tokenize>", line 6
    else:
    ^
IndentationError: unindent does not match any outer indentation level
```

En effet les deux lignes `if` et `else` font logiquement partie du même bloc, elles **doivent** donc avoir la même indentation. Avec cette présentation le lecteur Python émet une erreur et ne peut pas interpréter le code.

2.12 Bonnes pratiques de présentation de code

2.12.1 Complément - niveau basique

La PEP-008

On trouve dans la PEP-008 (en anglais) les conventions de codage qui s'appliquent à toute la librairie standard, et qui sont certainement un bon point de départ pour vous aider à trouver le style de présentation qui vous convient.

Nous vous recommandons en particulier les sections sur

— l'indentation

— les espaces

— les commentaires

Un peu de lecture : le module `pprint`

Voici par exemple le code du module `pprint` (comme PrettyPrint) de la librairie standard qui permet d'imprimer des données.

La fonction du module - le pretty printing - est évidemment accessoire ici, mais vous pouvez y voir illustré

— le *docstring* pour le module : les lignes de 11 à 35,

— les indentations, comme nous l'avons déjà mentionné sont à 4 espaces, et sans tabulation,

— l'utilisation des espaces, notamment autour des affectations et opérateurs, des définitions de fonction, des appels de fonctions...

— les lignes qui restent dans une largeur "raisonnable" (79 caractères)

— vous pouvez regarder notamment la façon de couper les lignes pour respecter cette limite en largeur.

```
In [ ]: from modtools import show_module_html
        import pprint
        show_module_html(pprint)
```

Espaces

Comme vous pouvez le voir dans `pprint.py`, les règles principales concernant les espaces sont les suivantes.

— S'agissant des **affectations** et **opérateurs**, on fera

`x = y + z`

Et non pas

~~`x=y+z`~~

Ni

~~`x =y+z`~~

Ni encore

~~`x=y +z`~~

L'idée étant d'aérer de manière homogène pour faciliter la lecture.

— On **déclare une fonction** comme ceci
`def foo(x, y, z):`
Et non pas comme ceci (un espace en trop avant la parenthèse ouvrante)
~~`def foo (x, y, z) :`~~
Ni surtout comme ceci (pas d'espace entre les paramètres)
~~`def foo (x,y,z) :`~~

— La même règle s'applique naturellement aux **appels de fonction** :
`foo(x, y, z)`
et non pas ~~`foo (x,y,z)`~~
ni ~~`def foo (x, y, z) :`~~

Il est important de noter qu'il s'agit ici de **règles d'usage** et non pas de règles syntaxiques ; tous les exemples barrés ci-dessus sont en fait **syntaxiquement corrects**, l'interpréteur les accepterait sans souci ; mais ces règles sont **très largement adoptées**, et obligatoires pour intégrer du code dans la librairie standard.

Coupures de ligne

Nous allons à présent zoomer dans ce module pour voir quelques exemples de coupure de ligne. Par contraste avec ce qui précède, il s'agit cette fois surtout de **règles syntaxiques**, qui peuvent rendre un code non valide si elles ne sont pas suivies.

Coupure de ligne sans *backslash* (\)

```
In [ ]: show_module_html(pprint,
                        beg="def pprint",
                        end="def pformat")
```

La fonction `pprint` (ligne ~47) est une commodité (qui crée une instance de `PrettyPrinter`, sur lequel on envoie la méthode `pprint`).

Vous voyez ici qu'il n'est **pas nécessaire** d'insérer un *backslash* (\) à la fin des lignes 50 et 51, car il y a une parenthèse ouvrante qui n'est pas fermée à ce stade.

De manière générale, lorsqu'une parenthèse ouvrante `(` - idem avec les crochets `[` et accolades `{` - n'est pas fermée sur la même ligne, l'interpréteur suppose qu'elle sera fermée plus loin et n'impose pas de *backslash*.

Ainsi par exemple on peut écrire sans *backslash* :

```
valeurs = [
    1,
    2,
    3,
    5,
    7,
]
```

Ou encore

```
x = un_nom_de_fonction_tres_tres_long(
        argument1, argument2,
        argument3, argument4,
    )
```

À titre de rappel, signalons aussi les chaînes de caractères à base de `"""` ou `'''` qui permettent elles aussi d'utiliser pluseurs lignes consécutives sans *backslash*, comme

```
texte = """ Les sanglots longs
Des violons
De l'automne"""
```

Coupure de ligne avec *backslash* (\)

Par contre il est des cas où le backslash est nécessaire :

```
In [ ]: show_module_html(pprint,
                         beg="components), readable, recursive",
                         end="elif len(object) ",
                         lineno_width=3)
```

Dans ce fragment au contraire, vous voyez en ligne 522 qu'**il a fallu cette fois** insérer un *backslash* \ comme caractère de continuation pour que l'instruction puisse se poursuivre en ligne 523.

Coupures de lignes - épilogue

Dans tous le cas où une instruction est répartie sur plusieurs lignes, c'est naturellement l'indentation de **la première ligne** qui est significative pour savoir à quel bloc rattacher cette instruction.

Notez bien enfin qu'on peut toujours mettre un *backslash* même lorsque ce n'est pas nécessaire, mais on évite cette pratique en règle générale car les *backslash* nuisent à la lisibilité.

2.12.2 Complément - niveau intermédiaire

Outils liés à PEP008

Il existe plusieurs outils liés à la PEP0008, pour vérifier si votre code est conforme, ou même le modifier pour qu'il le devienne.

Ce qui nous donne un excellent prétexte pour parler un peu de https://pypi.python.org, qui est la plateforme qui distribue les logiciels disponibles via l'outil `pip3`.

Je vous signale notamment :

— https://pypi.python.org/pypi/pep8/pour vérifier, et
— https://pypi.python.org/pypi/autopep8/ pour modifier automatiquement votre code et le rendre conforme.

Les deux-points ' : '

Dans un autre registre entièrement, vous pouvez vous reporter à ce lien si vous êtes intéressé par la question de savoir pourquoi on a choisi un délimiteur (le caractère deux-points :) pour terminer les instructions comme `if`, `for` et `def`.

2.13 L'instruction `pass`

2.13.1 Complément - niveau basique

Nous avons vu qu'en Python les blocs de code sont définis par leur indentation.

Une fonction vide

Cette convention a une limitation lorsqu'on essaie de définir un bloc vide. Voyons par exemple comment on définirait en C une fonction qui ne fait rien :

```
/* une fonction C qui ne fait rien */
void foo() {}
```

Comme en Python on n'a pas d'accolade pour délimiter les blocs de code, il existe une instruction `pass`, qui ne fait rien. À l'aide de cette instruction on peut à présent définir une fonction vide comme ceci :

```
In [1]: # une fonction Python qui ne fait rien
        def foo():
            pass
```

Une boucle vide

Pour prendre un second exemple un peu plus pratique, et pour anticiper un peu sur l'instruction `while` que nous verrons très bientôt, voici un exemple d'une boucle vide, c'est à dire sans corps, qui permet de "dépiler" dans une liste jusqu'à l'obtention d'une certaine valeur :

```
In [2]: liste = list(range(10))
        print('avant', liste)
        while liste.pop() != 5:
            pass
        print('après', liste)

avant [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
après [0, 1, 2, 3, 4]
```

On voit qu'ici encore l'instruction `pass` a toute son utilité.

2.13.2 Complément - niveau intermédiaire

Un `if` sans `then`

```
In [3]: # on utilise dans ces exemples une condition fausse
        condition = False
```

Imaginons qu'on parte d'un code hypothétique qui fasse ceci :

```
In [4]: # la version initiale
        if condition:
            print("non")
        else:
            print("bingo")
```

```
bingo
```

Et que l'on veuille modifier ce code pour simplement supprimer l'impression de `non`. La syntaxe du langage **ne permet pas** de simplement commenter le premier `print` :

```
# si on commente le premier print
# la syntaxe devient incorrecte
if condition:
#    print "non"
else:
    print "bingo"
```

Évidemment ceci pourrait être récrit autrement en inversant la condition, mais parfois on s'efforce de limiter au maximum l'impact d'une modification sur le code. Dans ce genre de situation on préférera écrire plutôt :

```
In [5]: # on peut s'en sortir en ajoutant une instruction pass
        if condition:
        #    print "non"
            pass
        else:
            print("bingo")
```

```
bingo
```

Une classe vide

Enfin, comme on vient de le voir dans la vidéo, on peut aussi utiliser `pass` pour définir une classe vide comme ceci :

```
In [6]: class Foo:
            pass

In [7]: foo = Foo()
```

2.14 Fonctions avec ou sans valeur de retour

2.14.1 Complément - niveau basique

Le style procédural

Une procédure est une fonction qui se contente de dérouler des instructions. Voici un exemple d'une telle fonction :

```
In [1]: def affiche_carre(n):
            print("le carre de", n, "vaut", n*n)
```

qui s'utiliserait comme ceci :

```
In [2]: affiche_carre(12)

le carre de 12 vaut 144
```

Le style fonctionnel

Mais en fait, dans notre cas, il serait beaucoup plus commode de définir une fonction qui **retourne** le carré d'un nombre, afin de pouvoir écrire quelque chose comme :

```
surface = carre(15)
```

quitte à imprimer cette valeur ensuite si nécessaire. Jusqu'ici nous avons fait beaucoup appel à `print`, mais dans la pratique, imprimer n'est pas un but en soi.

L'instruction `return`

Voici comment on pourrait écrire une fonction `carre` qui **retourne** (on dit aussi **renvoie**) le carré de son argument :

```
In [3]: def carre(n):
            return n*n

        if carre(8) <= 100:
            print('petit appartement')

petit appartement
```

La sémantique (le mot savant pour "comportement") de l'instruction `return` est assez simple. La fonction qui est en cours d'exécution **s'achève** immédiatement, et l'objet cité dans l'instruction `return` est retourné à l'appelant, qui peut utiliser cette valeur comme n'importe quelle expression.

Le singleton `None`

Le terme même de fonction, si vous vous rappelez vos souvenirs de mathématiques, suggère qu'on calcule un résultat à partir de valeurs d'entrée. Dans la pratique il est assez rare qu'on définisse une fonction qui ne retourne rien.

En fait **toutes** les fonctions retournent quelque chose. Lorsque le programmeur n'a pas prévu d'instruction `return`, Python retourne un objet spécial, baptisé `None`. Voici par exemple ce qu'on obtient si on essaie d'afficher la valeur de retour de notre première fonction, qui, on le rappelle, ne retourne rien :

```
In [4]: # ce premier appel provoque l'impression d'une ligne
        retour = affiche_carre(15)

le carre de 15 vaut 225


In [5]: # voyons ce qu'a retourné la fonction affiche_carre
        print('retour =', retour)

retour = None
```

L'objet `None` est un singleton prédéfini par Python, un peu comme `True` et `False`. Ce n'est pas par contre une valeur booléenne, nous aurons l'occasion d'en reparler.

Un exemple un peu plus réaliste

Pour illustrer l'utilisation de `return` sur un exemple plus utile, voyons le code suivant :

```
In [6]: def premier(n):
            """
            Retourne un booléen selon que n est premier ou non
            Retourne None pour les entrées négatives ou nulles
            """
            # retourne None pour les entrées non valides
            if n <= 0:
                return
            # traiter le cas singulier
            elif n == 1:
                return False
            # chercher un diviseur dans [2..n-1]
            # bien sûr on pourrait s'arrêter à la racine carrée de n
            # mais ce n'est pas notre sujet
            else:
                for i in range(2, n):
                    if n % i == 0:
                        # on a trouvé un diviseur,
                        # on peut sortir de la fonction
                        return False
            # à ce stade, le nombre est bien premier
            return True
```

Cette fonction teste si un entier est premier ou non ; il s'agit naturellement d'une version d'école, il existe d'autres méthodes beaucoup plus adaptées à cette tâche. On peut toutefois vérifier que cette version est fonctionnelle pour de petits entiers comme suit. On rappelle que `1` n'est pas considéré comme un nombre premier :

```
In [7]: for test in [-2, 1, 2, 4, 19, 35]:
            print(f"premier({test:2d}) = {premier(test)}")
```

```
premier(-2) = None
premier( 1) = False
premier( 2) = True
premier( 4) = False
premier(19) = True
premier(35) = False
```

`return` **sans valeur** Pour les besoins de cette discussion, nous avons choisi de retourner `None` pour les entiers négatifs ou nuls, une manière comme une autre de signaler que la valeur en entrée n'est pas valide.

Ceci n'est pas forcément une bonne pratique, mais elle nous permet ici d'illustrer que dans le cas où on ne mentionne pas de valeur de retour, Python retourne `None`.

`return` **interrompt la fonction** Comme on peut s'en convaincre en instrumentant le code - ce que vous pouvez faire à titre d'exercice en ajoutant des fonctions `print` - dans le cas d'un nombre qui n'est pas premier la boucle `for` ne va pas jusqu'à son terme.

On aurait pu d'ailleurs tirer profit de cette propriété pour écrire la fonction de manière légèrement différente comme ceci :

```
In [8]: def premier_sans_else(n):
            """
            Retourne un booléen selon que n est premier ou non
            Retourne None pour les entrées négatives ou nulles
            """
            # retourne None pour les entrées non valides
            if n <= 0:
                return
            # traiter le cas singulier
            if n == 1:
                return False
            # par rapport à la première version, on a supprimé
            # la clause else: qui est inutile
            for i in range(2, n):
                if n % i == 0:
                    # on a trouve un diviseur
                    return False
            # a ce stade c'est que le nombre est bien premier
            return True
```

C'est une question de style et de goût. En tout cas, les deux versions sont tout à fait équivalentes, comme on le voit ici :

```
In [9]: for test in [-2, 2, 4, 19, 35]:
            print(f"pour n = {test:2d} : premier → {premier(test)}\n"
                  f"    premier_sans_else → {premier_sans_else(test)}\n")
```

```
pour n = -2 : premier → None
    premier_sans_else → None
```

```
pour n =  2 : premier → True
    premier_sans_else → True

pour n =  4 : premier → False
    premier_sans_else → False

pour n = 19 : premier → True
    premier_sans_else → True

pour n = 35 : premier → False
    premier_sans_else → False
```

Digression sur les chaînes Vous remarquerez dans cette dernière cellule, si vous regardez bien le paramètre de `print`, qu'on peut accoler deux chaînes (ici deux *f-strings*) sans même les ajouter ; un petit détail pour éviter d'alourdir le code :

```
In [10]: # quand deux chaînes apparaissent immédiatement
         # l'une après l'autre sans opérateur, elles sont concaténées
         "abc" "def"

Out[10]: 'abcdef'
```

2.15 Formatage des chaines de caractères

2.15.1 Exercice - niveau basique

```
In [ ]: # charger l'exercice
        from exo_label import exo_label
```

Vous devez écrire une fonction qui prend deux arguments :

— une chaîne de caractères qui désigne le prénom d'un élève ;
— un entier qui indique la note obtenue.

Elle devra retourner une chaîne de caractères selon que la note est

— $0 \leqslant note < 10$
— $10 \leqslant note < 16$
— $16 \leqslant note \leqslant 20$

comme on le voit sur les exemples :

```
In [ ]: exo_label.example()
```

```
In [ ]: # à vous de jouer
        def label(prenom, note):
            "votre code"
```

```
In [ ]: # pour corriger
        exo_label.correction(label)
```

2.16 Séquences

2.16.1 Exercice - niveau basique

Slicing

Commençons par créer une chaîne de caractères. Ne vous inquiétez pas si vous ne comprenez pas encore le code d'initialisation utilisé ci-dessous.

Pour les plus curieux, l'instruction `import` permet de charger dans votre programme une boîte à outils que l'on appelle un module. Python vient avec de nombreux modules qui forment la bibliothèque standard. Le plus difficile avec les modules de la bibliothèque standard est de savoir qu'ils existent. En effet, il y en a un grand nombre et bien souvent il existe un module pour faire ce que vous souhaitez.

Ici en particulier nous utilisons le module `string`.

```
In [1]: import string
        chaine = string.ascii_lowercase
        print(chaine)
```

```
abcdefghijklmnopqrstuvwxyz
```

Pour chacune des sous-chaînes ci-dessous, écrire une expression de slicing sur `chaine` qui renvoie la sous-chaîne. La cellule de code doit retourner `True`.
Par exemple, pour obtenir "def" :

```
In [2]: chaine[3:6] == "def"
```

```
Out[2]: True
```

1) Écrivez une slice pour obtenir "vwx" (n'hésitez pas à utiliser les indices négatifs) :

```
In [ ]: chaine[ <votre_code> ] == "vwx"
```

2) Une slice pour obtenir "wxyz" (avec une seule constante) :

```
In [ ]: chaine[ <votre_code> ] == "wxyz"
```

3) Une slice pour obtenir "dfhjlnprtvxz" (avec deux constantes) :

```
In [ ]: chaine[ <votre_code> ] == "dfhjlnprtvxz"
```

4) Une slice pour obtenir "xurolifc" (avec deux constantes) :

```
In [ ]: chaine[ <votre_code> ] == "xurolifc"
```

2.16.2 Exercice - niveau intermédiaire

Longueur

```
In [ ]: # il vous faut évaluer cette cellule magique
        # pour charger l'exercice qui suit
        # et autoévaluer votre réponse
        from corrections.exo_inconnue import exo_inconnue
```

On vous donne une chaîne `composite` dont on sait qu'elle a été calculée à partir de deux chaînes `inconnue` et `connue` comme ceci :

```
composite = connue + inconnue + connue
```

On vous donne également la chaîne `connue`. Imaginez par exemple que vous avez (ce ne sont pas les vraies valeurs) :

```
connue = '0bf1'
composite = '0bf1a9730e150bf1'
```

alors, dans ce cas :

```
inconnue = 'a9730e15'
```

L'exercice consiste à écrire une fonction qui retourne la valeur de `inconnue` à partir de celles de `composite` et `connue`. Vous pouvez admettre que `connue` n'est pas vide, c'est-à-dire qu'elle contient au moins un caractère.

Vous pouvez utiliser du *slicing*, et la fonction `len()`, qui retourne la longueur d'une chaîne :

```
In [3]: len('abcd')

Out[3]: 4

In [ ]: # à vous de jouer
        def inconnue(composite, connue):
            "votre code"
```

Une fois votre code évalué, vous pouvez évaluer la cellule suivante pour vérifier votre résultat.

```
In [ ]: # correction
        exo_inconnue.correction(inconnue)
```

Lorsque vous évaluez cette cellule, la correction vous montre :

— dans la première colonne l'appel qui est fait à votre fonction ;
— dans la seconde colonne la valeur attendue pour `inconnue` ;
— dans la troisième colonne ce que votre code a réellement calculé.

Si toutes les lignes sont **en vert** c'est que vous avez réussi cet exercice.
Vous pouvez essayer autant de fois que vous voulez, mais il vous faut alors à chaque itération :

— évaluer votre cellule-réponse (là où vous définissez la fonction `inconnue`) ;
— et ensuite évaluer la cellule correction pour la mettre à jour.

2.17 Listes

2.17.1 Exercice - niveau basique

```
In [ ]: from corrections.exo_laccess import exo_laccess
```

Vous devez écrire une fonction `laccess` qui prend en argument une liste, et qui retourne :

— `None` si la liste est vide ;
— sinon le dernier élément de la liste si elle est de taille paire ;
— et sinon l'élément du milieu.

```
In [ ]: exo_laccess.example()
```

```
In [ ]: # écrivez votre code ici
        def laccess(liste):
            return "votre code"
```

```
In [ ]: # pour le corriger
        exo_laccess.correction(laccess)
```

Une fois que votre code fonctionne, vous pouvez regarder si par hasard il marcherait aussi avec des chaînes :

```
In [ ]: from corrections.exo_laccess import exo_laccess_strings
```

```
In [ ]: exo_laccess_strings.correction(laccess)
```

2.18 Instruction `if` et fonction `def`

2.18.1 Exercice - niveau basique

Fonction de divisibilité

```
In [ ]: # chargement de l'exercice
        from corrections.exo_divisible import exo_divisible
```

L'exercice consiste à écrire une fonction baptisée `divisible` qui retourne une valeur booléenne, qui indique si un des deux arguments est divisible par l'autre.

Vous pouvez supposer les entrées `a` et `b` entiers et non nuls, mais pas forcément positifs.

```
In [ ]: def divisible(a, b):
            "<votre_code>"
```

Vous pouvez à présent tester votre code en évaluant ceci, qui écrira un message d'erreur si un des jeux de test ne donne pas le résultat attendu.

```
In [ ]: # tester votre code
        exo_divisible.correction(divisible)
```

2.18.2 Exercice - niveau basique

Fonction définie par morceaux

```
In [ ]: # chargement de l'exercice
        from corrections.exo_morceaux import exo_morceaux
```

On veut définir en Python une fonction qui est définie par morceaux :

$$f : x \longrightarrow \begin{cases} -x-5 & \text{si } x \leqslant -5 \\ 0 & \text{si } x \in [-5,5] \\ \frac{1}{5}x-1 & \text{si } x \geqslant 5 \end{cases}$$

```
In [ ]: # donc par exemple
        exo_morceaux.example()
```

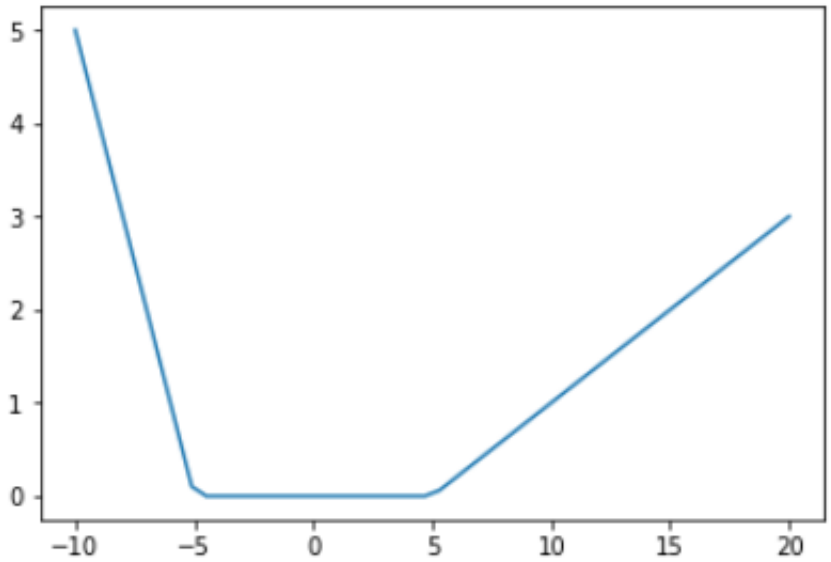
```
In [ ]: # à vous de jouer

        def morceaux(x):
            return 0 # "votre code"
```

```
In [ ]: # pour corriger votre code
        exo_morceaux.correction(morceaux)
```

Représentation graphique L'exercice est maintenant terminé, mais nous allons voir ensemble maintenant comment vous pourriez visualiser votre fonction.

Voici ce qui est attendu comme courbe pour `morceaux` (image fixe) :



En partant de votre code, vous pouvez produire votre propre courbe en utilisant `numpy` et `matplotlib` comme ceci :

```
In [ ]: # on importe les bibliothèques
        import numpy as np
        import matplotlib.pyplot as plt

In [ ]: # un échantillon des X entre -10 et 20
        X = np.linspace(-10, 20)

        # et les Y correspondants
        Y = np.vectorize(morceaux)(X)

In [ ]: # on n'a plus qu'à dessiner
        plt.plot(X, Y)
        plt.show()
```

2.19 Comptage dans les chaines

2.19.1 Exercice - niveau basique

Nous remercions Benoit Izac pour cette contribution aux exercices.

2.19.2 La commande UNIX wc(1)

Sur les systèmes de type UNIX, la commande wc permet de compter le nombre de lignes, de mots et d'octets (ou de caractères) présents sur l'entrée standard ou contenus dans un fichier.

L'exercice consiste à écrire une fonction nommée *wc* qui prendra en argument une chaîne de caractères et retournera une liste contenant trois éléments :

1. le nombre de lignes (plus précisément le nombre de retours à la ligne) ;
2. le nombre de mots (un mot étant séparé par des espaces) ;
3. le nombre de caractères (on utilisera uniquement le jeu de caractères ASCII).

```
In [ ]: # chargement de l'exercice
        from corrections.exo_wc import exo_wc
```

```
In [ ]: # exemple
        exo_wc.example()
```

Indice : nous avons vu rapidement la boucle `for`, sachez toutefois qu'on peut tout à fait résoudre l'exercice en utilisant uniquement la bibliothèque standard.

Remarque : usuellement, ce genre de fonctions retournerait plutôt un tuple qu'une liste, mais comme nous ne voyons les tuples que la semaine prochaine..

À vous de jouer :

```
In [ ]: # la fonction à implémenter
        def wc(string):
            # remplacer pass par votre code
            pass
```

```
In [ ]: # correction
        exo_wc.correction(wc)
```

2.20 Compréhensions (1)

2.20.1 Exercice - niveau basique

Liste des valeurs d'une fonction

```
In [ ]: # Pour charger l'exercice
        from corrections.exo_liste_p import exo_liste_P
```

On se donne une fonction polynomiale :
$P(x) = 2x^2 - 3x - 2$
On vous demande d'écrire une fonction `liste_P` qui prend en argument une liste de nombres réels x et qui retourne la liste des valeurs $P(x)$.

```
In [ ]: # voici un exemple de ce qui est attendu
        exo_liste_P.example()
```

Écrivez votre code dans la cellule suivante (*On vous suggère d'écrire une fonction `P` qui implémente le polynôme mais ça n'est pas strictement indispensable, seul le résultat de `liste_P` compte*) :

```
In [ ]: def P(x):
            "<votre code>"

        def liste_P(liste_x):
            "votre code"
```

Et vous pouvez le vérifier en évaluant cette cellule :

```
In [ ]: # pour vérifier votre code
        exo_liste_P.correction(liste_P)
```

2.20.2 Récréation

Si vous avez correctement implémenté la fonction `liste_P` telle que demandé dans le premier exercice, vous pouvez visualiser le polynôme `P` en utilisant `matplotlib` avec le code suivant :

```
In [ ]: # on importe les bibliothèques
        import numpy as np
        import matplotlib.pyplot as plt
```

```
In [ ]: # un échantillon des X entre -10 et 10
        X = np.linspace(-10, 10)

        # et les Y correspondants
        Y = liste_P(X)
```

```
In [ ]: # on n'a plus qu'à dessiner
        plt.plot(X, Y)
        plt.show()
```

2.21 Compréhensions (2)

2.21.1 Exercice - niveau intermédiaire

Mise au carré

```
In [ ]: # chargement de l'exercice
        from corrections.exo_carre import exo_carre
```

On vous demande à présent d'écrire une fonction dans le même esprit que ci-dessus. Cette fois, chaque ligne contient, séparés par des points-virgules, une liste d'entiers, et on veut obtenir une nouvelle chaîne avec les carrés de ces entiers, séparés par des deux-points.

À nouveau les lignes peuvent être remplies de manière approximative, avec des espaces, des tabulations, ou même des points-virgules en trop, que ce soit au début, à la fin, ou au milieu d'une ligne.

```
In [ ]: # exemples
        exo_carre.example()
```

```
In [ ]: # écrivez votre code ici
        def carre(ligne):
            "<votre_code>"
```

```
In [ ]: # pour corriger
        exo_carre.correction(carre)
```