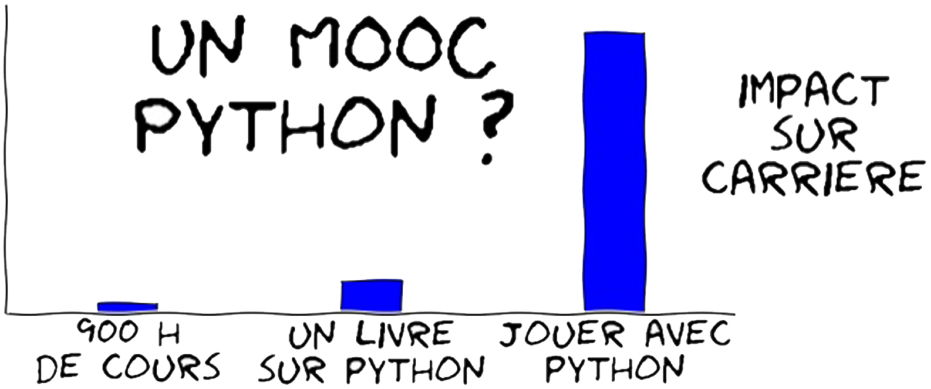
python™ 3

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

Chapitre 3

Renforcement des notions de base, références partagées

Chapitre 4

Fonctions et portée des variables

4.1 Passage d'arguments par référence

4.1.1 Complément - niveau intermédiaire

Entre le code qui appelle une fonction et le code de la fonction elle-même

```
In [1]: def ma_fonction(dans_fonction):
            print(dans_fonction)

        dans_appelant = ["texte"]
        ma_fonction(dans_appelant)
```

```
['texte']
```

on peut se demander quelle est exactement la nature de la relation entre l'appelant et l'appelé, c'est-à-dire ici `dans_appelant` et `dans_fonction`.

C'est l'objet de ce complément.

Passage par valeur - passage par référence

Si vous avez appris d'autres langages de programmation comme C ou C++, on a pu vous parler de deux modes de passage de paramètres :

— par valeur : cela signifie qu'on communique à la fonction, non pas l'entité dans l'appelant, mais seulement **sa valeur** ; en clair, **une copie** ;

— par référence : cela signifie qu'on passe à la fonction une **référence** à l'argument dans l'appelant, donc essentiellement les deux codes **partagent** la même mémoire.

Python fait du passage par référence

Certains langages comme Pascal - et C++ si on veut - proposent ces deux modes. En Python, tous les passages de paramètres se font **par référence**.

```
In [ ]: # chargeons la magie pour pythontutor
        %load_ext ipythontutor
```

```
In [ ]: %%ipythontutor curInstr=4
        def ma_fonction(dans_fonction):
            print(dans_fonction)

        dans_appelant = ["texte"]
        ma_fonction(dans_appelant)
```

Ce qui signifie qu'on peut voir le code ci-dessus comme étant - pour simplifier - équivalent à ceci :

```
In [2]: dans_appelant = ["texte"]

        # ma_fonction (dans_appelant)
        # → on entre dans la fonction
        dans_fonction = dans_appelant
        print(dans_fonction)
```

```
['texte']
```

On peut le voir encore d'une autre façon en instrumentant le code comme ceci – on rappelle que la fonction built-in `id` retourne l'adresse mémoire d'un objet :

```
In [3]: def ma_fonction(dans_fonction):
            print('dans ma_fonction', dans_fonction , id(dans_fonction))

        dans_appelant = ["texte"]
        print('dans appelant   ', dans_appelant, id(dans_appelant))
        ma_fonction(dans_appelant)
```

```
dans appelant    ['texte'] 69872896
dans ma_fonction ['texte'] 69872896
```

Des références partagées

On voit donc que l'appel de fonction crée des références partagées, exactement comme l'affectation, et que tout ce que nous avons vu au sujet des références partagées s'applique exactement à l'identique :

```
In [4]: # on ne peut pas modifier un immuable dans une fonction
        def increment(n):
            n += 1

        compteur = 10
        increment(compteur)
        print(compteur)
```

```
10
```

```
In [5]: # on peut par contre ajouter dans une liste
        def insert(liste, valeur):
            liste.append(valeur)

        liste = ["un"]
        insert(liste, "texte")
        print(liste)
```

```
['un', 'texte']
```

Pour cette raison, il est important de bien préciser, quand vous documentez une fonction, si elle fait des effets de bord sur ses arguments (c'est-à-dire qu'elle modifie ses arguments), ou si elle produit une copie. Rappelez-vous par exemple le cas de la méthode `sort` sur les listes, et de la fonction de commodité `sorted`, que nous avions vues en semaine 2.

De cette façon, on saura s'il faut ou non copier l'argument avant de le passer à votre fonction.

4.2 Rappels sur *docstring*

4.2.1 Complément - niveau basique

Comment documenter une fonction

Pour rappel, il est recommandé de toujours documenter les fonctions en ajoutant une chaîne comme première instruction.

```
In [1]: def flatten(containers):
            "returns a list of the elements of the elements in containers"
            return [element for container in containers for element in container]
```

Cette information peut être consultée, soit interactivement :

```
In [2]: help(flatten)
```

```
Help on function flatten in module __main__:

flatten(containers)
    returns a list of the elements of the elements in containers
```

Soit programmativement :

```
In [3]: flatten.__doc__
```

```
Out[3]: 'returns a list of the elements of the elements in containers'
```

Sous quel format ?

L'usage est d'utiliser une chaîne simple (délimitée par « " » ou « ' ») lorsque le *docstring* tient sur une seule ligne, comme ci-dessus.

Lorsque ce n'est pas le cas - et pour du vrai code, c'est rarement le cas - on utilise des chaînes multi-lignes (délimitées par « """ » ou « ''' »). Dans ce cas le format est très flexible, car le *docstring* est normalisé, comme on le voit sur ces deux exemples, où le rendu final est identique :

```
In [4]: # un style de docstring multi-lignes
        def flatten(containers):
            """
        provided that containers is a list (or more generally an iterable)
        of elements that are themselves iterables, this function
        returns a list of the items in these elements
            """
            return [element for container in containers for element in container]

        help(flatten)
```

```
Help on function flatten in module __main__:

flatten(containers)
```

```
    provided that containers is a list (or more generally an iterable)
    of elements that are themselves iterables, this function
    returns a list of the items in these elements
```

```
In [5]: # un autre style, qui donne le même résultat
        def flatten(containers):
            """
            provided that containers is a list (or more generally an iterable)
            of elements that are themselves iterables, this function
            returns a list of the items in these elements
            """
            return [element for container in containers for element in container]

        help(flatten)
```

```
Help on function flatten in module __main__:

flatten(containers)
    provided that containers is a list (or more generally an iterable)
    of elements that are themselves iterables, this function
    returns a list of the items in these elements
```

Quelle information ?

On remarquera que dans ces exemples, le *docstring* ne répète pas le nom de la fonction ou des arguments (en mots savants, sa *signature*), et que ça n'empêche pas `help` de nous afficher cette information.

Le PEP 257 qui donne les conventions autour du *docstring* précise bien ceci :

> The one-line docstring should NOT be a "signature" reiterating the function/method parameters (which can be obtained by introspection). Don't do :

```
def function(a, b):
    """function(a, b) -> list"""
```

> <...>
>
> The preferred form for such a docstring would be something like :

```
def function(a, b):
    """Do X and return a list."""
```

> (Of course "Do X" should be replaced by a useful description !)

Pour en savoir plus

Vous trouverez tous les détails sur *docstring* dans le PEP 257.

4.3 `isinstance`

4.3.1 Complément - niveau basique

Typage dynamique

En première semaine, nous avons rapidement mentionné les concepts de typage statique et dynamique.

Avec la fonction prédéfinie `isinstance` - qui peut être par ailleurs utile dans d'autres contextes - vous pouvez facilement :

— vérifier qu'un argument d'une fonction a bien le type attendu,

— traiter différemment les entrées selon leur type.

Voyons tout de suite sur un exemple simple comment on pourrait définir une fonction qui travaille sur un entier, mais qui par commodité peut aussi accepter un entier passé comme une chaîne de caractères, ou même une liste d'entiers (auquel cas on renvoie la liste des factorielles) :

```
In [1]: def factoriel(argument):
            # si on reçoit un entier
            if isinstance(argument, int):              # (*)
                return 1 if argument <= 1 else argument * factoriel(argument - 1)
            # convertir en entier si on reçoit une chaîne
            elif isinstance(argument, str):
                return factoriel(int(argument))
            # la liste des résultats si on reçoit un tuple ou une liste
            elif isinstance(argument, (tuple, list)):  # (**)
                return [factoriel(i) for i in argument]
            # sinon on lève une exception
            else:
                raise TypeError(argument)

In [2]: print("entier", factoriel(4))
        print("chaine", factoriel("8"))
        print("tuple", factoriel((4, 8)))

entier 24
chaine 40320
tuple [24, 40320]
```

Remarquez que la fonction `isinstance` **possède elle-même** une logique de ce genre, puisqu'en ligne 3 `(*)` nous lui avons passé en deuxième argument un type (`int`), alors qu'en ligne 11 `(**)` on lui a passé un tuple de deux types. Dans ce second cas naturellement, elle vérifie si l'objet (le premier argument) est **de l'un des types** mentionnés dans le tuple.

4.3.2 Complément - niveau intermédiaire

Le module `types`

Le module `types` définit un certain nombre de constantes qui peuvent être utiles dans ce contexte - vous trouverez une liste exhaustive à la fin de ce notebook. Par exemple :

```
In [3]: from types import FunctionType
        isinstance(factoriel, FunctionType)
```

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

Mais méfiez-vous toutefois des fonctions *built-in*, qui sont de type `BuiltinFunctionType`

```
In [4]: from types import BuiltinFunctionType
        isinstance(len, BuiltinFunctionType)
```

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

```
In [5]: # alors qu'on pourrait penser que
        isinstance(len, FunctionType)
```

```
Out[5]: False
```

`isinstance` *vs* `type`

Il est recommandé d'utiliser `isinstance` par rapport à la fonction `type`. Tout d'abord, cela permet, on vient de le voir, de prendre en compte plusieurs types.

Mais aussi et surtout `isinstance` supporte la notion d'héritage qui est centrale dans le cadre de la programmation orientée objet, sur laquelle nous allons anticiper un tout petit peu par rapport aux présentations de la semaine prochaine.

Avec la programmation objet, vous pouvez définir vos propres types. On peut par exemple définir une classe `Animal` qui convient pour tous les animaux, puis définir une sous-classe `Mammifere`. On dit que la classe `Mammifere` *hérite* de la classe `Animal`, et on l'appelle sous-classe parce qu'elle représente une partie des animaux; et donc tout ce qu'on peut faire sur les animaux peut être fait sur les mammifères.

En voici une implémentation très rudimentaire, uniquement pour illustrer le principe de l'héritage. Si ce qui suit vous semble difficile à comprendre, pas d'inquiétude, nous reviendrons sur ce sujet lorsque nous parlerons des classes.

```
In [6]: class Animal:
            def __init__(self, name):
                self.name = name

        class Mammifere(Animal):
            def __init__(self, name):
                Animal.__init__(self, name)
```

Ce qui nous intéresse dans l'immédiat c'est que `isinstance` permet dans ce contexte de faire des choses qu'on ne peut pas faire directement avec la fonction `type`, comme ceci :

```
In [7]: # c'est comme ceci qu'on peut créer un objet de type `Animal` (méthode __init__)
        requin = Animal('requin')
        # idem pour un Mammifere
        baleine = Mammifere('baleine')

        # bien sûr ici la réponse est 'True'
        print("l'objet baleine est-il un mammifere ?", isinstance(baleine, Mammifere))
```

```
l'objet baleine est-il un mammifere ? True
```

```
In [8]: # ici c'est moins évident, mais la réponse est 'True' aussi
        print("l'objet baleine est-il un animal ?", isinstance(baleine, Animal))
```

```
l'objet baleine est-il un animal ? True
```

Vous voyez qu'ici, bien que l'objet baleine soit de type `Mammifere`, on peut le considérer comme étant **aussi** de type `Animal`.

Ceci est motivé de la façon suivante : comme on l'a dit plus haut, tout ce qu'on peut faire (en matière notamment d'envoi de méthodes) sur un objet de type `Animal`, on peut le faire sur un objet de type `Mammifere`. Dit en termes ensemblistes, l'ensemble des mammifères est inclus dans l'ensemble des animaux.

Annexe - Les symboles du module `types`

Vous pouvez consulter la documentation du module `types`.

```
In [9]: # voici par ailleurs la liste de ses attributs
        import types
        dir(types)
```

```
Out[9]: ['AsyncGeneratorType',
         'BuiltinFunctionType',
         'BuiltinMethodType',
         'CodeType',
         'CoroutineType',
         'DynamicClassAttribute',
         'FrameType',
         'FunctionType',
         'GeneratorType',
         'GetSetDescriptorType',
         'LambdaType',
         'MappingProxyType',
         'MemberDescriptorType',
         'MethodType',
         'ModuleType',
         'SimpleNamespace',
         'TracebackType',
         '_GeneratorWrapper',
         '__all__',
         '__builtins__',
         '__cached__',
         '__doc__',
         '__file__',
         '__loader__',
         '__name__',
         '__package__',
         '__spec__',
         '_ag',
```

```
        '_calculate_meta',
        '_collections_abc',
        '_functools',
        'coroutine',
        'new_class',
        'prepare_class']
```

4.4 *Type hints*

4.4.1 Complément - niveau intermédiaire

Langages compilés

Nous avons évoqué en première semaine le typage, lorsque nous avons comparé Python avec les langages compilés. Dans un langage compilé avec typage statique, on **doit fournir du typage**, ce qui fait qu'on écrit typiquement une fonction comme ceci :

```
int factoriel(int n) {
  return (n<=1) ? 1 : n * factoriel(n-1);
}
```

ce qui signifie que la fonction factoriel prend un premier argument qui est un entier, et qu'elle retourne également un entier.

Nous avons vu également que, par contraste, pour écrire une fonction en Python, on n'a **pas besoin** de préciser **le type** des arguments ni du retour de la fonction.

Vous pouvez aussi typer votre code python

Cependant depuis la version 3.5, python supporte un mécanisme **totalement optionnel** qui vous permet d'annoter les arguments des fonctions avec des informations de typage, ce mécanisme est connu sous le nom de *type hints*, et ça se présente comme ceci :

typer une variable

```
In [1]: # pour typer une variable avec les type hints
        nb_items : int = 0

In [2]: nb_items

Out[2]: 0
```

typer les paramètres et le retour d'une fonction

```
In [3]: # une fonction factorielle avec des type hints
        def fact(n : int) -> int:
            return 1 if n <= 1 else n * fact(n-1)

In [4]: fact(12)

Out[4]: 479001600
```

Usages

À ce stade, on peut entrevoir les usages suivants à ce type d'annotation :

— tout d'abord, et évidemment, cela peut permettre de mieux documenter le code ;
— les environnements de développement sont susceptibles de vous aider de manière plus effective ; si quelque part vous écrivez `z = fact(12)`, le fait de savoir que `z` est entier permet de fournir une complétion plus pertinente lorsque vous commencez à écrire `z.[TAB]` ;

— on peut espérer trouver des erreurs dans les passages d'arguments à un stade plus précoce du développement.

Par contre ce qui est très très clairement annoncé également, c'est que ces informations de typage sont **totalement facultatives**, et que le langage les **ignore totalement**.

```
In [5]: # l'interpréteur ignore totalement ces informations
        def fake_fact(n : str) -> str:
            return 1 if n <= 1 else n * fake_fact(n-1)

        # on peut appeler fake_fact avec un int alors
        # que c'est déclaré pour des str
        fake_fact(12)
```

```
Out[5]: 479001600
```

Le modèle préconisé est d'utiliser des **outils extérieurs**, qui peuvent faire une analyse statique du code pour exploiter ces informations à des fins de validation. Dans cette catégorie, le plus célèbre est sans doute `mypy`. Notez aussi que les IDE comme PyCharm sont également capables de tirer parti de ces annotations.

Est-ce répandu ?

Parce qu'ils ont été introduits pour la première fois avec python-3.5, en 2015 donc, puis améliorés dans la 3.6 pour le typage des variables, l'usage des *type hints* n'est pour l'instant pas très répandu, en proportion de code en tous cas. En outre, il aura fallu un temps de latence avant que tous les outils (IDE's, producteurs de documentation, outils de test, validateurs...) ne soient améliorés pour en tirer un profit maximal.

On peut penser que cet usage va se répandre avec le temps, peut-être / sans doute pas de manière systématique, mais *a minima* pour lever certaines ambiguïtés.

Comment annoter son code

Maintenant que nous en avons bien vu la finalité, voyons un très bref aperçu des possibilités offertes pour la construction des types dans ce contexte de *type hints*. N'hésitez pas à vous reporter à la documentation officielle du module `typing` pour un exposé plus exhaustif.

le module `typing`

L'ensemble des symboles que nous allons utiliser dans la suite de ce complément provient du module `typing`

exemples simples

```
In [6]: from typing import List
```

```
In [7]: # une fonction qui
        # attend un paramètre qui soit une liste d'entiers,
        # et qui retourne une liste de chaînes
        def foo(x: List[int]) -> List[str]:
            pass
```

avertissement : `list` vs `List`

Remarquez bien dans l'exemple ci-dessus que nous avons utilisé `typing.List` plutôt que le type *built-in* `list`, alors que l'on a pu par contre utiliser `int` et `str`.

Les raisons pour cela sont de deux ordres :

— tout d'abord, si je devais utiliser `list` pour construire un type comme *liste d'entiers*, il me faudrait écrire quelque chose comme `list(int)` ou encore `list[int]`, et cela serait source de confusion car ceci a déjà une signification dans le langage ;

— de manière plus profonde, il faut distinguer entre `list` qui est un type concret (un objet qui sert à construire des instances), de `List` qui dans ce contexte doit plus être vu comme un type abstrait.

Pour bien voir cela, considérez l'exemple suivant :

```
In [8]: from typing import Iterable

In [9]: def lower_split(sep: str, inputs : Iterable[str]) -> str:
            return sep.join([x.lower() for x in inputs])

In [10]: lower_split('--', ('AB', 'CD', 'EF'))

Out[10]: 'ab--cd--ef'
```

On voit bien dans cet exemple que `Iterable` ne correspond pas à un type concret particulier, c'est un type abstrait dans le sens du *duck typing*.

un exemple plus complet

Voici un exemple tiré de la documentation du module `typing` qui illustre davantage de types construits à partir des types *builtin* du langage :

```
In [11]: from typing import Dict, Tuple, List

         ConnectionOptions = Dict[str, str]
         Address = Tuple[str, int]
         Server = Tuple[Address, ConnectionOptions]

         def broadcast_message(message: str, servers: List[Server]) -> None:
             ...

         # The static type checker will treat the previous type signature as
         # being exactly equivalent to this one.
         def broadcast_message(
                 message: str,
                 servers: List[Tuple[Tuple[str, int], Dict[str, str]]]) -> None:
             ...
```

J'en profite d'ailleurs (ça n'a rien a voir, mais...) pour vous signaler un objet python assez étrange :

```
In [12]: # L'objet ... existe bel et bien en Python
         el = ...
         el
```

```
Out[12]: Ellipsis
```

qui sert principalement pour le slicing multidimensionnel de numpy. Mais ne nous égarons pas...

typage partiel

Puisque c'est un mécanisme optionnel, vous pouvez tout à fait ne typer qu'une partie de vos variables et paramètres :

```
In [13]: # imaginez que vous ne typez pas n2, ni la valeur de retour

         # c'est équivalent de dire ceci
         def partially_typed(n1: int, n2):
             return None
```

```
In [14]: # ou cela
         from typing import Any

         def partially_typed(n1: int, n2: Any) -> Any:
             return None
```

alias

On peut facilement se définir des alias; lorsque vous avez implémenté un système d'identifiants basé sur le type `int`, il est préférable de faire :

```
In [15]: from typing import NewType

         UserId = NewType('UserId', int)

         user1_id : UserId = 0
```

plutôt que ceci, qui est beaucoup moins parlant :

```
In [16]: user1_id : int = 0
```

4.4.2 Complément - niveau avancé

`Generic` Pour ceux qui connaissent déjà la notion de classe (les autres peuvent ignorer la fin de ce complément) :

Grâce aux constructions `TypeVar` et `Generic`, il est possible de manipuler une notion de *variable de type*, que je vous montre sur un exemple tiré à nouveau de la documentation du module `typing` :

```
In [17]: from typing import TypeVar, Generic
         from logging import Logger

         T = TypeVar('T')

         class LoggedVar(Generic[T]):
```

```
def __init__(self, value: T, name: str, logger: Logger) -> None:
    self.name = name
    self.logger = logger
    self.value = value

def set(self, new: T) -> None:
    self.log('Set ' + repr(self.value))
    self.value = new

def get(self) -> T:
    self.log('Get ' + repr(self.value))
    return self.value

def log(self, message: str) -> None:
    self.logger.info('%s: %s', self.name, message)
```

qui vous donne je l'espère une idée de ce qu'il est possible de faire, et jusqu'où on peut aller avec les *type hints*. Si vous êtes intéressé par cette fonctionnalité, je vous invite à poursuivre la lecture ici.

Pour en savoir plus

— la documentation officielle sur le module typing ;

— la page d'accueil de l'outil mypy.

— le PEP-525 sur le typage des paramètres et retours de fonctions, implémenté dans python-3.5 ;

— le PEP-526 sur le typage des variables, implémenté dans 3.6.

4.5 Conditions & Expressions Booléennes

4.5.1 Complément - niveau basique

Nous présentons rapidement dans ce notebook comment construire la condition qui contrôle l'exécution d'un `if`.

Tests considérés comme vrai

Lorsqu'on écrit une instruction comme

```
if <expression>:
   <do_something>
```

le résultat de l'expression peut **ne pas être un booléen**.

Par exemple, pour n'importe quel type numérique, la valeur 0 est considérée comme fausse. Cela signifie que

```
In [1]: # ici la condition s'évalue à 0, donc on ne fait rien
        if 3 - 3:
            print("ne passera pas par là")
```

```
In [2]: # par contre si vous vous souvenez de notre cours sur les flottants
        # ici la condition donne un tout petit réel mais pas 0.
        if 0.1 + 0.2 - 0.3:
            print("par contre on passe ici")
```

```
par contre on passe ici
```

De même, une chaîne vide, une liste vide, un tuple vide, sont considérés comme faux. Bref, vous voyez l'idée générale.

```
In [3]: if "":
            print("ne passera pas par là")
        if []:
            print("ne passera pas par là")
        if ():
            print("ne passera pas par là")
```

```
In [4]: # assez logiquement, None aussi
        # est considéré comme faux
        if None:
            print("ne passe toujours pas par ici")
```

Égalité

Les tests les plus simples se font à l'aide des opérateurs d'égalité, qui fonctionnent sur presque tous les objets. L'opérateur == vérifie si deux objets ont la même valeur :

```
In [5]: bas = 12
        haut = 25.82

        # égalité ?
        if bas == haut:
            print('==')
```

```
In [6]: # non égalité ?
        if bas != haut:
            print('!=')

!=
```

En général, deux objets de types différents ne peuvent pas être égaux.

```
In [7]: # ces deux objets se ressemblent
        # mais ils ne sont pas du même type !
        if [1, 2] != (1, 2):
            print('!=')

!=
```

Par contre, des `float`, des `int` et des `complex` peuvent être égaux entre eux :

```
In [8]: bas_reel = 12.

In [9]: print(bas, bas_reel)

12 12.0


In [10]: # le réel 12 et
         # l'entier 12 sont égaux
         if bas == bas_reel:
             print('int == float')

int == float


In [11]: # ditto pour int et complex
         if (12 + 0j) == 12:
             print('int == complex')

int == complex
```

Signalons à titre un peu anecdotique une syntaxe ancienne : historiquement et **seulement en Python 2** on pouvait aussi noter <> le test de non égalité. On trouve ceci dans du code ancien mais il faut éviter de l'utiliser :

```
In [12]: %%python2

         # l'ancienne forme de !=
         if 12 <> 25:
             print("<> est obsolete et ne fonctionne qu'en python2")

Couldn't find program: 'python2'
```

Les opérateurs de comparaison

Sans grande surprise on peut aussi écrire

```
In [13]: if bas <= haut:
             print('<=')
         if bas < haut:
             print('<')
```

```
<=
<
```

```
In [14]: if haut >= bas:
             print('>=')
         if haut > bas:
             print('>')
```

```
>=
>
```

À titre de curiosité, on peut même écrire en un seul test une appartenance à un intervalle, ce qui donne un code plus lisible

```
In [15]: x = (bas + haut) / 2
         print(x)
```

```
18.91
```

```
In [16]: # deux tests en une expression
         if bas <= x <= haut:
             print("dans l'intervalle")
```

```
dans l'intervalle
```

On peut utiliser les comparaisons sur une palette assez large de types, comme par exemple avec les listes

```
In [17]: # on peut comparer deux listes, mais ATTENTION
         [1, 2] <= [2, 3]
```

```
Out[17]: True
```

Il est parfois utile de vérifier le sens qui est donné à ces opérateurs selon le type ; ainsi par exemple sur les ensembles ils se réfèrent à l'**inclusion**.

Il faut aussi se méfier avec les types numériques, si un complexe est impliqué, comme dans l'exemple suivant :

```
In [18]: # on ne peut pas par contre comparer deux nombres complexes
         try:
             2j <= 3j
         except Exception as e:
             print("OOPS", type(e), e)
```

```
OOPS <class 'TypeError'> '<=' not supported between instances of 'complex' and 'complex'
```

Connecteurs logiques et / ou / non

On peut bien sûr combiner facilement plusieurs expressions entre elles, grâce aux opérateurs `and`, `or` et `not`

```
In [19]: # il ne faut pas faire ceci, mettez des parenthèses
         if 12 <= 25. or [1, 2] <= [2, 3] and not 12 <= 32:
             print("OK mais pourrait être mieux")
```

```
OK mais pourrait être mieux
```

En matière de priorités : le plus simple si vous avez une expression compliquée reste de mettre les parenthèses qui rendent son évaluation claire et lisible pour tous. Aussi on préfèrera de beaucoup la formulation équivalente :

```
In [20]: # c'est mieux avec un parenthésage
         if 12 <= 25. or ([1, 2] <= [2, 3] and not 12 <= 32):
             print("OK, c'est équivalent et plus clair")
```

```
OK, c'est équivalent et plus clair
```

```
In [21]: # attention, si on fait un autre parenthésage, on change le sens
         if (12 <= 25. or [1, 2] <= [2, 3]) and not 12 <= 32 :
             print("ce n'est pas équivalent, ne passera pas par là")
```

Pour en savoir plus

Reportez-vous à la section sur les opérateurs booléens dans la documentation python.

4.6 Évaluation des tests

4.6.1 Complément - niveau basique

Quels tests sont évalués ?

On a vu dans la vidéo que l'instruction conditionnelle `if` permet d'implémenter simplement des branchements à plusieurs choix, comme dans cet exemple :

```
In [1]: s = 'berlin'
        if 'a' in s:
            print('avec a')
        elif 'b' in s:
            print('avec b')
        elif 'c' in s:
            print('avec c')
        else:
            print('sans a ni b ni c')
```

```
avec b
```

Comme on s'en doute, les expressions conditionnelles **sont évaluées jusqu'à obtenir un résultat vrai** - ou considéré comme vrai -, et le bloc correspondant est alors exécuté. Le point important ici est qu'**une fois qu'on a obtenu un résultat vrai**, on sort de l'expression conditionnelle **sans évaluer les autres conditions**. En termes savants, on parle d'évaluation paresseuse : on s'arrête dès qu'on peut.

Dans notre exemple, on aura évalué à la sortie `'a' in s`, et aussi `'b' in s`, mais pas `'c' in s`

Pourquoi c'est important ?

C'est important de bien comprendre quels sont les tests qui sont réellement évalués pour deux raisons :

— d'abord, pour des raisons de performance ; comme on n'évalue que les tests nécessaires, si un des tests prend du temps, il est peut-être préférable de le faire en dernier ;
— mais aussi et surtout, il se peut tout à fait qu'un test fasse des **effets de bord**, c'est-à-dire qu'il modifie un ou plusieurs objets.

Dans notre premier exemple, les conditions elles-mêmes sont inoffensives ; la valeur de `s` reste *identique*, que l'on *évalue ou non* les différentes conditions.

Mais nous allons voir ci-dessous qu'il est relativement facile d'écrire des conditions qui **modifient** par **effet de bord** les objets mutables sur lesquelles elles opèrent, et dans ce cas il est crucial de bien assimiler la règle des évaluations des expressions dans un `if`.

4.6.2 Complément - niveau intermédiaire

Rappel sur la méthode `pop`

Pour illustrer la notion d'**effet de bord**, nous revenons sur la méthode de liste `pop()` qui, on le rappelle, renvoie un élément de liste **après l'avoir effacé** de la liste.

```
In [2]: # on se donne une liste
        liste = ['premier', 'deuxieme', 'troisieme']
        print(f"liste={liste}")

liste=['premier', 'deuxieme', 'troisieme']


In [3]: # pop(0) renvoie le premier élément de la liste, et raccourcit la liste
        element = liste.pop(0)
        print(f"après pop(0), element={element} et liste={liste}")

après pop(0), element=premier et liste=['deuxieme', 'troisieme']


In [4]: # et ainsi de suite
        element = liste.pop(0)
        print(f"après pop(0), element={element} et liste={liste}")

après pop(0), element=deuxieme et liste=['troisieme']
```

Conditions avec effet de bord

Une fois ce rappel fait, voyons maintenant l'exemple suivant :

```
In [5]: liste = list(range(5))
        print('liste en entree:', liste, 'de taille', len(liste))

liste en entree: [0, 1, 2, 3, 4] de taille 5


In [6]: if liste.pop(0) <= 0:
            print('cas 1')
        elif liste.pop(0) <= 1:
            print('cas 2')
        elif liste.pop(0) <= 2:
            print('cas 3')
        else:
            print('cas 4')
        print('liste en sortie de taille', len(liste))

cas 1
liste en sortie de taille 4
```

Avec cette entrée, le premier test est vrai (car `pop(0)` renvoie 0), aussi on n'exécute en tout `pop()` qu'**une seule fois**, et donc à la sortie la liste n'a été raccourcie que d'un élément.

Exécutons à présent le même code avec une entrée différente :

```
In [7]: liste = list(range(5, 10))
        print('en entree: liste=', liste, 'de taille', len(liste))
```

```
en entree: liste= [5, 6, 7, 8, 9] de taille 5
```

```
In [8]: if liste.pop(0) <= 0:
            print('cas 1')
        elif liste.pop(0) <= 1:
            print('cas 2')
        elif liste.pop(0) <= 2:
            print('cas 3')
        else:
            print('cas 4')
        print('en sortie: liste=', liste, 'de taille', len(liste))
```

```
cas 4
en sortie: liste= [8, 9] de taille 2
```

On observe que cette fois la liste a été **raccourcie 3 fois**, car les trois tests se sont révélés faux.

Cet exemple vous montre qu'il faut être attentif avec des conditions qui font des effets de bord. Bien entendu, ce type de pratique est de manière générale à utiliser avec beaucoup de discernement.

Court-circuit (*short-circuit*)

La logique que l'on vient de voir est celle qui s'applique aux différentes branches d'un `if` ; c'est la même logique qui est à l'œuvre aussi lorsque python évalue une condition logique à base de `and` et `or`. C'est ici aussi une forme d'évaluation paresseuse.

Pour illustrer cela, nous allons nous définir deux fonctions toutes simples qui renvoient `True` et `False` mais avec une impression de sorte qu'on voit lorsqu'elles sont exécutées :

```
In [9]: def true():
            print('true')
            return True

In [10]: def false():
             print('false')
             return False

In [11]: true()

true


Out[11]: True
```

Ceci va nous permettre d'illustrer notre point, qui est que lorsque python évalue un `and` ou un `or`, il **n'évalue la deuxième condition que si c'est nécessaire**. Ainsi par exemple :

```
In [12]: false() and true()

false
```

```
Out[12]: False
```

Dans ce cas, python évalue la première partie du `and` - qui provoque l'impression de `false` - et comme le résultat est faux, il n'est **pas nécessaire** d'évaluer la seconde condition, on sait que de toute façon le résultat du `and` est forcément faux. C'est pourquoi vous ne voyez pas l'impression de `true`.

De manière symétrique avec un `or` :

```
In [13]: true() or false()

true


Out[13]: True
```

À nouveau ici il n'est pas nécessaire d'évaluer `false()`, et donc seul `true` est imprimé à l'évaluation.

À titre d'exercice, essayez de dire combien d'impressions sont émises lorsqu'on évalue cette expression un peu plus compliquée :

```
In [14]: true() and (false() or true()) or (true () and false())

true
false
true


Out[14]: True
```

4.7 Une forme alternative du `if`

4.7.1 Complément - niveau basique

Expressions et instructions

Les constructions python que nous avons vues jusqu'ici peuvent se ranger en deux familles :

— d'une part les **expressions** sont les fragments de code qui **retournent une valeur** ;

  — c'est le cas lorsqu'on invoque n'importe quel opérateur numérique, pour les appels de fonctions, ...

— d'autre part les **instructions** ;

  — dans cette famille, nous avons vu par exemple l'affectation et `if`, et nous en verrons bien d'autres.

La différence essentielle est que les expressions peuvent être combinées entre elles pour faire des expressions arbitrairement grosses. Aussi, si vous avez un doute pour savoir si vous avez affaire à une expression ou à une instruction, demandez-vous si vous pourriez utiliser ce code **comme membre droit d'une affectation**. Si oui, vous avez une expression.

`if` est une instruction

La forme du `if` qui vous a été présentée pendant la vidéo ne peut pas servir à renvoyer une valeur, c'est donc une **instruction**.

Imaginons maintenant qu'on veuille écrire quelque chose d'aussi simple que *"affecter à y la valeur 12 ou 35, selon que x est vrai ou non"*.

Avec les notions introduites jusqu'ici, il nous faudrait écrire ceci :

```
In [1]: x = True  # ou quoi que ce soit d'autre
        if x:
            y = 12
        else:
            y = 35
        print(y)
```

```
12
```

Expression conditionnelle

Il existe en Python une expression qui fait le même genre de test ; c'est la forme dite d'**expression conditionnelle**, qui est une **expression à part entière**, avec la syntaxe :

```
<résultat_si_vrai> if <condition> else <résultat_si_faux>
```

Ainsi on pourrait écrire l'exemple ci-dessus de manière plus simple et plus concise comme ceci :

```
In [2]: y = 12 if x else 35
        print(y)
```

```
12
```

Cette construction peut souvent rendre le style de programmation plus fonctionnel et plus fluide.

4.7.2 Complément - niveau intermédiaire

Imbrications

Puisque cette forme est une expression, on peut l'utiliser dans une autre expression conditionnelle, comme ici :

```
In [3]: # on veut calculer en fonction d'une entrée x
        # une sortie qui vaudra
        # -1 si x < -10
        # 0 si -10 <= x <= 10
        # 1 si x > 10

        x = 5 # ou quoi que ce soit d'autre

        valeur = -1 if x < -10 else (0 if x <= 10 else 1)

        print(valeur)
```

```
0
```

Remarquez bien que cet exemple est équivalent à la ligne

```
valeur = -1 if x < -10 else 0 if x <= 10 else 1
```

mais qu'il est fortement recommandé d'utiliser, comme on l'a fait, un parenthésage pour lever toute ambiguïté.

Pour en savoir plus

— La section sur les expressions conditionnelles de la documentation Python.
— Le PEP308 qui résume les discussions ayant donné lieu au choix de la syntaxe adoptée.

De manière générale, les PEP rassemblent les discussions préalables à toutes les évolutions majeures du langage Python.

4.8 Récapitulatif sur les conditions dans un `if`

4.8.1 Complément - niveau basique

Dans ce complément nous résumons ce qu'il faut savoir pour écrire une condition dans un `if`.

Expression *vs* instruction

Nous avons déjà introduit la différence entre instruction et expression, lorsque nous avons vu l'expression conditionnelle :

— une expression est un fragment de code qui "retourne quelque chose", item alors qu'une instruction permet bien souvent de faire une action, mais ne retourne rien.

Ainsi parmi les notions que nous avons vues jusqu'ici, nous pouvons citer dans un ordre arbitraire :

Instructions	Expressions
affectation	appel de fonction
`import`	opérateurs `is`, `in`, `==`, ...
instruction `if`	expression conditionnelle
instruction `for`	compréhension(s)

Toutes les expressions sont éligibles

Comme condition d'une instruction `if`, on peut mettre n'importe quelle expression. On l'a déjà signalé, il n'est pas nécessaire que cette expression retourne un booléen :

```
In [1]: # dans ce code le test
        # if n % 3:
        # est équivalent à
        # if n % 3 != 0:

        for n in (18, 19):
            if n % 3:
                print(f"{n} non divisible par trois")
            else:
                print(f"{n} divisible par trois")
```

```
18 divisible par trois
19 non divisible par trois
```

Une valeur est-elle "vraie" ?

Se pose dès lors la question de savoir précisément quelles valeurs sont considérées comme *vraies* par l'instruction `if`.

Parmi les types de base, nous avons déjà eu l'occasion de l'évoquer, les valeurs *fausses* sont typiquement :

— 0 pour les valeurs numériques ;

— les objets vides pour les chaînes, listes, ensembles, dictionnaires, etc.

Pour en avoir le cœur net, pensez à utiliser dans le terminal interactif la fonction `bool`. Comme pour toutes les fonctions qui portent le nom d'un type, la fonction `bool` est un constructeur qui fabrique un objet booléen.

Si vous appelez `bool` sur un objet, la valeur de retour - qui est donc par construction une valeur booléenne - vous indique, cette fois sans ambiguïté - comment se comportera `if` avec cette entrée.

```
In [2]: def show_bool(x):
            print(f"condition {repr(x):>10} considérée comme {bool(x)}")

In [3]: for exp in [None, "", 'a', [], [1], (), (1, 2), {}, {'a': 1}, set(), {1}]:
            show_bool(exp)

condition       None considérée comme False
condition         '' considérée comme False
condition        'a' considérée comme True
condition         [] considérée comme False
condition        [1] considérée comme True
condition         () considérée comme False
condition     (1, 2) considérée comme True
condition         {} considérée comme False
condition   {'a': 1} considérée comme True
condition      set() considérée comme False
condition        {1} considérée comme True
```

Quelques exemples d'expressions

Référence à une variable et dérivés

```
In [4]: a = list(range(4))
        print(a)

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

```
In [5]: if a:
            print("a n'est pas vide")
        if a[0]:
            print("on ne passe pas par ici")
        if a[1]:
            print("a[1] n'est pas nul")

a n'est pas vide
a[1] n'est pas nul
```

Appels de fonction ou de méthode

```
In [6]: chaine = "jean"
        if chaine.upper():
            print("la chaine mise en majuscule n'est pas vide")
```

```
la chaine mise en majuscule n'est pas vide
```

```
In [7]: # on rappelle qu'une fonction qui ne fait pas 'return' retourne None
        def procedure(a, b, c):
            "cette fonction ne retourne rien"
            pass

        if procedure(1, 2, 3):
            print("ne passe pas ici car procedure retourne None")
        else:
            print("par contre on passe ici")
```

```
par contre on passe ici
```

Compréhensions

Il découle de ce qui précède qu'on peut tout à fait mettre une compréhension comme condition, ce qui peut être utile pour savoir si au moins un élément remplit une condition, comme par exemple :

```
In [8]: inputs = [23, 65, 24]

        # y a-t-il dans inputs au moins un nombre
        # dont le carré est de la forme 10*n+5
        def condition(n):
            return (n * n) % 10 == 5

        if [value for value in inputs if condition(value)]:
            print("au moins une entrée convient")
```

```
au moins une entrée convient
```

Opérateurs

Nous avons déjà eu l'occasion de rencontrer la plupart des opérateurs de comparaison du langage, dont voici à nouveau les principaux :

Famille	Exemples
Égalité	`==`, `!=`, `is`, `is not`
Appartenance	`in`
Comparaison	`<=`, `<`, `>`, `>=`
Logiques	`and`, `or`, `not`

4.8.2 Complément - niveau intermédiaire

Remarques sur les opérateurs

Voici enfin quelques remarques sur ces opérateurs

opérateur d'égalité ==

L'opérateur == ne fonctionne en général (sauf pour les nombres) que sur des objets de même type ; c'est-à-dire que notamment un tuple ne sera jamais égal à une liste :

```
In [9]: [] == ()

Out[9]: False

In [10]: [1, 2] == (1, 2)

Out[10]: False
```

opérateur logiques

Comme c'est le cas avec par exemple les opérateurs arithmétiques, les opérateurs logiques ont une *priorité*, qui précise le sens des phrases non parenthésées. C'est-à-dire pour être explicite, que de la même manière que

```
12 + 4 * 8
```

est équivalent à

```
12 + ( 4 * 8 )
```

pour les booléens il existe une règle de ce genre et

```
a and not b or c and d
```

est équivalent à

```
(a and (not b)) or (c and d)
```

Mais en fait, il est assez facile de s'emmêler dans ces priorités, et c'est pourquoi il est **très fortement conseillé** de parenthéser.

opérateurs logiques (2)

Remarquez aussi que les opérateurs logiques peuvent être appliqués à des valeurs qui ne sont pas booléennes :

```
In [11]: 2 and [1, 2]

Out[11]: [1, 2]

In [12]: None or "abcde"

Out[12]: 'abcde'
```

Dans la logique de l'évaluation paresseuse qu'on a vue récemment, remarquez que lorsque l'évaluation d'un `and` ou d'un `or` ne peut pas être court-circuitée, le résultat est alors toujours le résultat de la dernière expression évaluée :

```
In [13]: 1 and 2 and 3
```

```
Out[13]: 3

In [14]: 1 and 2 and 3 and '' and 4

Out[14]: ''

In [15]: [] or "" or {}

Out[15]: {}

In [16]: [] or "" or {} or 4 or set()

Out[16]: 4
```

Expression conditionnelle dans une instruction `if`

En toute rigueur on peut aussi mettre un `<> if <> else <>` - donc une expression conditionnelle - comme condition dans une instruction `if`. Nous le signalons pour bien illustrer la logique du langage, mais cette pratique n'est bien sûr pas du tout conseillée.

```
In [17]: # cet exemple est volontairement tiré par les cheveux
         # pour bien montrer qu'on peut mettre n'importe quelle expression comme condition
         a = 1
         # ceci est franchement illisible
         if 0 if not a else 2:
             print("une construction illisible")
         # et encore pire
         if 0 if a else 3 if a + 1 else 2:
             print("encore pire")
```

```
une construction illisible
```

Pour en savoir plus

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

Types définis par l'utilisateur

Pour anticiper un tout petit peu, nous verrons que les classes en Python vous donnent le moyen de définir vos propres types d'objets. Nous verrons à cette occasion qu'il est possible d'indiquer à python quels sont les objets de type `MaClasse` qui doivent être considérés comme `True` ou comme `False`.

De manière plus générale, tous les traits natifs du langage sont redéfinissables sur les classes. Nous verrons par exemple également comment donner du sens à des phrases comme

```
mon_objet = MaClasse()
if mon_objet:
    <faire quelque chose>
```

ou encore

```
mon_objet = MaClasse()
for partie in mon_objet:
    <faire quelque chose sur partie>
```

Mais n'anticipons pas trop, rendez-vous en semaine 6.

4.9 L'instruction `if`

4.9.1 Exercice - niveau basique

Répartiteur (1)

```
In [ ]: # on charge l'exercice
        from corrections.exo_dispatch import exo_dispatch1
```

On vous demande d'écrire une fonction `dispatch1`, qui prend en argument deux entiers `a` et `b`, et qui renvoie selon les cas :

	a *pair*	a *impair*
b *pair*	$a^2 + b^2$	$(a-1) * b$
b *impair*	$a * (b-1)$	$a^2 - b^2$

```
In [ ]: # un petit exemple
        exo_dispatch1.example()
```

```
In [ ]: def dispatch1(a, b):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_dispatch1.correction(dispatch1)
```

4.9.2 Exercice - niveau basique

Répartiteur (2)

```
In [ ]: # chargement de l'exercice
        from corrections.exo_dispatch import exo_dispatch2
```

Dans une seconde version de cet exercice, on vous demande d'écrire une fonction `dispatch2` qui prend en arguments :

- `a` et `b` deux entiers
- `A` et `B` deux ensembles (chacun pouvant être matérialisé par un ensemble, une liste ou un tuple)

et qui renvoie selon les cas :

	$a \in A$	$a \notin A$
$b \in B$	$a^2 + b^2$	$(a-1) * b$
$b \notin B$	$a * (b-1)$	$a^2 + b^2$

```
In [ ]: def dispatch2(a, b, A, B):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_dispatch2.correction(dispatch2)
```

4.10 Expression conditionnelle

4.10.1 Exercice - niveau basique

Analyse et mise en forme

```
In [ ]: # Pour charger l'exercice
        from corrections.exo_libelle import exo_libelle
```

Un fichier contient, dans chaque ligne, des informations (champs) séparées par des virgules. Les espaces et tabulations présentes dans la ligne ne sont pas significatives et doivent être ignorées.

Dans cet exercice de niveau basique, on suppose que chaque ligne a exactement 3 champs, qui représentent respectivement le prénom, le nom, et le rang d'une personne dans un classement. Une fois les espaces et tabulations ignorées, on ne fait pas de vérification sur le contenu des 3 champs.

On vous demande d'écrire la fonction `libelle`, qui sera appelée pour chaque ligne du fichier. Cette fonction :

— prend en argument une ligne (chaîne de caractères)

— retourne une chaîne de caractères mise en forme (voir plus bas)

— ou bien retourne `None` si la ligne n'a pas pu être analysée, parce qu'elle ne vérifie pas les hypothèses ci-dessus (c'est notamment le cas si on ne trouve pas exactement les 3 champs)

La mise en forme consiste à retourner

```
Nom.Prenom (message)
```

le *message* étant lui-même le *rang* mis en forme pour afficher '1er', '2nd' ou '*n*-ème' selon le cas. Voici quelques exemples

```
In [ ]: # voici quelques exemples de ce qui est attendu
        exo_libelle.example()
```

```
In [ ]: # écrivez votre code ici
        def libelle(ligne):
            "<votre_code>"
```

```
In [ ]: # pour le vérifier
        exo_libelle.correction(libelle)
```

4.11 La boucle `while` ... `else`

4.11.1 Complément - niveau basique

Boucles sans fin - `break`

Utiliser `while` plutôt que `for` est une affaire de style et d'habitude. Cela dit en Python, avec les notions d'itérable et d'itérateur, on a tendance à privilégier l'usage du `for` pour les boucles finies et déterministes.

Le `while` reste malgré tout d'un usage courant, et notamment avec une condition `True`.
Par exemple le code de l'interpréteur interactif de python pourrait ressembler, vu de très loin, à quelque chose comme ceci :

```
while True:
    print(eval(read()))
```

Notez bien par ailleurs que les instructions `break` et `continue` fonctionnent, à l'intérieur d'une boucle `while`, exactement comme dans un `for`, c'est-à-dire que :

— `continue` termine l'itération courante mais reste dans la boucle, alors que

— `break` interrompt l'itération courante et sort également de la boucle.

4.11.2 Complément - niveau intermédiaire

Rappel sur les conditions

On peut utiliser dans une boucle `while` toutes les formes de conditions que l'on a vues à l'occasion de l'instruction `if`.

Dans le contexte de la boucle `while` on comprend mieux, toutefois, pourquoi le langage autorise d'écrire des conditions dont le résultat n'est **pas nécessairement un booléen**. Voyons cela sur un exemple simple :

```
In [1]: # une autre façon de parcourir une liste
        liste = ['a', 'b', 'c']

        while liste:
            element = liste.pop()
            print(element)
```

```
c
b
a
```

Une curiosité : la clause `else`

Signalons enfin que la boucle `while` - au même titre d'ailleurs que la boucle `for`, peut être assortie d'une clause `else`, qui est exécutée à la fin de la boucle, **sauf dans le cas d'une sortie avec** `break`

```
In [2]: # Un exemple de while avec une clause else
```

```
        # si break_mode est vrai on va faire un break
        # après le premier élément de la liste
        def scan(liste, break_mode):

            # un message qui soit un peu parlant
            message = "avec break" if break_mode else "sans break"
            print(message)
            while liste:
                print(liste.pop())
                if break_mode:
                    break
            else:
                print('else...')

In [3]: # sortie de la boucle sans break
        # on passe par else
        scan(['a'], False)

sans break
a
else...


In [4]: # on sort de la boucle par le break
        scan(['a'], True)

avec break
a
```

Ce trait est toutefois **très rarement** utilisé.

4.12 Calculer le PGCD

4.12.1 Exercice - niveau basique

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

On vous demande d'écrire une fonction qui calcule le PGCD de deux entiers, en utilisant l'algorithme d'Euclide.

Les deux paramètres sont supposés être des entiers positifs ou nuls (pas la peine de le vérifier).

Dans le cas où un des deux paramètres est nul, le PGCD vaut l'autre paramètre. Ainsi par exemple :

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

Remarque on peut tout à fait utiliser une fonction récursive pour implémenter l'algorithme d'Euclide. Par exemple cette version de `pgcd` fonctionne très bien aussi (en supposant a>=b)

```
def pgcd(a, b):
   "Le PGCD avec une fonction récursive"
   if not b:
       return a
   return pgcd(b, a % b)
```

Cependant, il vous est demandé ici d'utiliser une boucle `while`, qui est le sujet de la séquence, pour implémenter `pgcd`.

```
In [ ]: # à vous de jouer
        def pgcd(a, b):
            "<votre code>"
```

```
In [ ]: # pour vérifier votre code
        exo_pgcd.correction(pgcd)
```

4.13 Exercice

4.13.1 Niveau basique

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

On se propose d'écrire une fonction `taxes` qui calcule le montant de l'impôt sur le revenu au Royaume-Uni.

Le barème est publié ici par le gouvernement anglais, voici les données utilisées pour l'exercice :

Tranche	Revenu imposable	Taux
Non imposable	jusque £11.500	0%
Taux de base	£11.501 à £45.000	20%
Taux élevé	£45.001 à £150.000	40%
Taux supplémentaire	au delà de £150.000	45%

Donc naturellement il s'agit d'écrire une fonction qui prend en argument le revenu imposable, et retourne le montant de l'impôt, **arrondi à l'entier inférieur**.

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

Indices

— évidemment on parle ici d'une fonction continue ;
— aussi en termes de programmation, je vous encourage à séparer la définition des tranches de la fonction en elle-même.

```
In [ ]: def taxes(income):
            # ce n'est pas la bonne réponse
            return (income-11_500) * (20/100)
```

```
In [ ]: exo_taxes.correction(taxes)
```

Représentation graphique Comme d'habitude vous pouvez voir la représentation graphique de votre fonction :

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
```

```
In [ ]: %matplotlib inline
        plt.ion()
```

```
In [ ]: X = np.linspace(0, 200_000)
        Y = [taxes(x) for x in X]
        plt.plot(X, Y);
```

```
In [ ]: # et pour changer la taille de la figure
        plt.figure(figsize=(10, 8))
        plt.plot(X, Y);
```

4.14 Le module `builtins`

4.14.1 Complément - niveau avancé

Ces noms qui viennent de nulle part

Nous avons vu déjà un certain nombre de **fonctions *built-in*** comme par exemple

```
In [1]: open, len, zip
```

```
Out[1]: (<function io.open>, <function len>, zip)
```

Ces noms font partie du **module** `builtins`. Il est cependant particulier puisque tout se passe **comme si** on avait fait avant toute chose :

```
from builtins import *
```

sauf que cet import est implicite.

On peut réaffecter un nom *built-in*

Quoique ce soit une pratique déconseillée, il est tout à fait possible de redéfinir ces noms ; on peut faire par exemple

```
In [2]: # on réaffecte le nom open à un nouvel objet fonction
        def open(encoding='utf-8', *args):
            print("ma fonction open")
            pass
```

qui est naturellement **très vivement déconseillé**. Notez, cependant, que la coloration syntaxique vous montre clairement que le nom que vous utilisez est un *built-in* (en vert dans un notebook).

On ne peut pas réaffecter un mot clé

À titre de digression, rappelons que les noms prédéfinis dans le module `builtins` sont, à cet égard aussi, très différents des mots-clés comme `if`, `def`, `with` et autres `for` qui eux, ne peuvent pas être modifiés en aucune manière :

```
>>> lambda = 1
  File "<stdin>", line 1
    lambda = 1
           ^
SyntaxError: invalid syntax
```

Retrouver un objet *built-in*

Il faut éviter de redéfinir un nom prédéfini dans le module `builtins` ; un bon éditeur de texte vous signalera les fonctions *built-in* avec une coloration syntaxique spécifique. Cependant, on peut vouloir redéfinir un nom *built-in* pour changer un comportement par défaut, puis vouloir revenir au comportement original.

Sachez que vous pouvez toujours “retrouver” alors la fonction *built-in* en l’important explicitement du module `builtins`. Par exemple, pour réaliser notre ouverture de fichier, nous pouvons toujours faire :

```
In [3]: # nous ne pouvons pas utiliser open puisque
        open()

ma fonction open


In [4]: # pour être sûr d'utiliser la bonne fonction open

        import builtins

        with builtins.open("builtins.txt", "w", encoding="utf-8") as f:
            f.write("quelque chose")
```

Ou encore, de manière équivalente :

```
In [5]: from builtins import open as builtins_open

        with builtins_open("builtins.txt", "r", encoding="utf-8") as f:
            print(f.read())

quelque chose
```

Liste des fonctions prédéfinies

Vous pouvez trouver la liste des fonctions prédéfinies ou *built-in* avec la fonction `dir` sur le module `builtins` comme ci-dessous (qui vous montre aussi les exceptions prédéfinies, qui commencent par une majuscule), ou dans la documentation sur les fonctions prédéfinies :

```
In [6]: dir(builtins)

Out[6]: ['ArithmeticError',
         'AssertionError',
         'AttributeError',
         'BaseException',
         'BlockingIOError',
         'BrokenPipeError',
         'BufferError',
         'BytesWarning',
         'ChildProcessError',
         'ConnectionAbortedError',
         'ConnectionError',
         'ConnectionRefusedError',
         'ConnectionResetError',
         'DeprecationWarning',
         'EOFError',
         'Ellipsis',
         'EnvironmentError',
         'Exception',
         'False',
         'FileExistsError',
         'FileNotFoundError',
         'FloatingPointError',
```

```
'FutureWarning',
'GeneratorExit',
'IOError',
'ImportError',
'ImportWarning',
'IndentationError',
'IndexError',
'InterruptedError',
'IsADirectoryError',
'KeyError',
'KeyboardInterrupt',
'LookupError',
'MemoryError',
'ModuleNotFoundError',
'NameError',
'None',
'NotADirectoryError',
'NotImplemented',
'NotImplementedError',
'OSError',
'OverflowError',
'PendingDeprecationWarning',
'PermissionError',
'ProcessLookupError',
'RecursionError',
'ReferenceError',
'ResourceWarning',
'RuntimeError',
'RuntimeWarning',
'StopAsyncIteration',
'StopIteration',
'SyntaxError',
'SyntaxWarning',
'SystemError',
'SystemExit',
'TabError',
'TimeoutError',
'True',
'TypeError',
'UnboundLocalError',
'UnicodeDecodeError',
'UnicodeEncodeError',
'UnicodeError',
'UnicodeTranslateError',
'UnicodeWarning',
'UserWarning',
'ValueError',
'Warning',
'WindowsError',
'ZeroDivisionError',
'__IPYTHON__',
```

```
'__build_class__',
'__debug__',
'__doc__',
'__import__',
'__loader__',
'__name__',
'__package__',
'__spec__',
'abs',
'all',
'any',
'ascii',
'bin',
'bool',
'bytearray',
'bytes',
'callable',
'chr',
'classmethod',
'compile',
'complex',
'copyright',
'credits',
'delattr',
'dict',
'dir',
'display',
'divmod',
'enumerate',
'eval',
'exec',
'filter',
'float',
'format',
'frozenset',
'get_ipython',
'getattr',
'globals',
'hasattr',
'hash',
'help',
'hex',
'id',
'input',
'int',
'isinstance',
'issubclass',
'iter',
'len',
'license',
'list',
```

```
'locals',
'map',
'max',
'memoryview',
'min',
'next',
'object',
'oct',
'open',
'ord',
'pow',
'print',
'property',
'range',
'repr',
'reversed',
'round',
'set',
'setattr',
'slice',
'sorted',
'staticmethod',
'str',
'sum',
'super',
'tuple',
'type',
'vars',
'zip']
```

Vous remarquez que les exceptions (les symboles qui commencent par des majuscules) représentent à elles seules une proportion substantielle de cet espace de noms.

4.15 Visibilité des variables de boucle

4.15.1 Complément - niveau basique

Une astuce

Dans ce complément, nous allons beaucoup jouer avec le fait qu'une variable soit définie ou non. Pour nous simplifier la vie, et surtout rendre les cellules plus indépendantes les unes des autres si vous devez les rejouer, nous allons utiliser la formule un peu magique suivante :

```
In [1]: # on détruit la variable i si elle existe
        if 'i' in locals():
            del i
```

qui repose d'une part sur l'instruction `del` que nous avons déjà vue, et sur la fonction *built-in* `locals` que nous verrons plus tard ; cette formule a l'avantage qu'on peut l'exécuter dans n'importe quel contexte, que `i` soit définie ou non.

Une variable de boucle reste définie au-delà de la boucle

Une variable de boucle est définie (assignée) dans la boucle et **reste *visible*** une fois la boucle terminée. Le plus simple est de le voir sur un exemple :

```
In [2]: # La variable 'i' n'est pas définie
        try:
            i
        except NameError as e:
            print('OOPS', e)
```

```
OOPS name 'i' is not defined
```

```
In [3]: # si à présent on fait une boucle
        # avec i comme variable de boucle
        for i in [0]:
            pass

        # alors maintenant i est définie
        i
```

```
Out[3]: 0
```

On dit que la variable *fuite* (en anglais "*leak*"), dans ce sens qu'elle continue d'exister au delà du bloc de la boucle à proprement parler.

On peut être tenté de tirer profit de ce trait, en lisant la valeur de la variable après la boucle ; l'objet de ce complément est de vous inciter à la prudence, et d'attirer votre attention sur certains points qui peuvent être sources d'erreur.

Attention aux boucles vides

Tout d'abord, il faut faire attention à ne pas écrire du code qui dépende de ce trait **si la boucle peut être vide**. En effet, si la boucle ne s'exécute pas du tout, la variable n'est **pas affectée** et donc elle n'est **pas définie**. C'est évident, mais ça peut l'être moins quand on lit du code réel, comme par exemple :

```
In [4]: # on détruit la variable i si elle existe
        if 'i' in locals():
            del i

In [5]: # une façon très scabreuse de calculer la longueur de l
        def length(l):
            for i, x in enumerate(l):
                pass
            return i + 1

        length([1, 2, 3])

Out[5]: 3
```

Ça a l'air correct, sauf que :

```
In [6]: length([])


        ---------------------------------------------------------------------------
        UnboundLocalError                         Traceback (most recent call last)

        <ipython-input-6-8c0f554916d9> in <module>()
    ----> 1 length([])


        <ipython-input-5-54c4139d6f55> in length(l)
          3     for i, x in enumerate(l):
          4         pass
    ----> 5     return i + 1
          6
          7 length([1, 2, 3])


        UnboundLocalError: local variable 'i' referenced before assignment
```

Ce résultat mérite une explication. Nous allons voir très bientôt l'exception `UnboundLocalError`, mais pour le moment sachez qu'elle se produit lorsqu'on a dans une fonction une variable locale et une variable globale de même nom. Alors, pourquoi l'appel `length([1, 2, 3])` retourne-t-il sans encombre, alors que pour l'appel `length([])` il y a une exception ? Cela est lié à la manière dont python détermine qu'une variable est locale.

Une variable est locale dans une fonction si elle est assignée dans la fonction explicitement (avec une opération d'affectation) ou implicitement (par exemple avec une boucle `for` comme ici) ; nous reviendrons sur ce point un peu plus tard. Mais pour les fonctions, pour une raison d'efficacité, une variable est définie comme locale à la phase de pré-compilation, c'est-à-dire *avant* l'exécution du code. Le pré-compilateur ne peut pas savoir quel sera l'argument passé à la fonction, il peut simplement savoir qu'il y a une boucle `for` utilisant la variable `i`, il en conclut que `i` est locale pour toute la fonction.

Lors du premier appel, on passe une liste à la fonction, liste qui est parcourue par la boucle `for`. En sortie de boucle, on a bien une variable locale `i` qui vaut 3. Lors du deuxième appel par contre, on passe une liste vide à la fonction, la boucle `for` ne peut rien parcourir, donc elle termine immédiatement. Lorsque l'on arrive à la ligne `return i + 1` de la fonction, la variable `i` n'a pas de valeur (on doit donc chercher `i` dans le module), mais `i` a été définie par le pré-compilateur comme étant locale, on a donc dans la même fonction une variable `i` locale et une référence à une variable `i` globale, ce qui provoque l'exception `UnboundLocalError`.

Comment faire alors ?

Utiliser une autre variable

La première voie consiste à déclarer une variable externe à la boucle et à l'affecter à l'intérieur de la boucle, c'est-à-dire :

```
In [7]: # on veut chercher le premier de ces nombres qui vérifie une condition
        candidates = [3, -15, 1, 8]

        # pour fixer les idées disons qu'on cherche un multiple de 5, peu importe
        def checks(candidate):
            return candidate % 5 == 0
```

```
In [8]: # plutôt que de faire ceci
        for item in candidates:
            if checks(item):
                break
        print('trouvé solution', item)
```

```
trouvé solution -15
```

```
In [9]: # il vaut mieux faire ceci
        solution = None
        for item in candidates:
            if checks(item):
                solution = item
                break

        print('trouvé solution', solution)
```

```
trouvé solution -15
```

Au minimum initialiser la variable

Au minimum, si vous utilisez la variable de boucle après la boucle, il est vivement conseillé de l'**initialiser** explicitement **avant** la boucle, pour vous prémunir contre les boucles vides, comme ceci :

```
In [10]: # la fonction length de tout à l'heure
         def length1(l):
             for i, x in enumerate(l):
                 pass
             return i + 1
```

```
In [11]: # une version plus robuste
         def length2(l):
             # on initialise i explicitement
             # pour le cas où l est vide
             i = -1
             for i, x in enumerate(l):
                 pass
             # comme cela i est toujours déclarée
             return i + 1
```

```
In [12]: length1([])
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)

<ipython-input-12-ebdd61d1db90> in <module>()
----> 1 length1([])


<ipython-input-10-6a59feac567b> in length1(l)
  3     for i, x in enumerate(l):
  4         pass
----> 5     return i + 1


UnboundLocalError: local variable 'i' referenced before assignment
```

```
In [13]: length2([])
```

```
Out[13]: 0
```

Les compréhensions

Notez bien que par contre, les variables de compréhension **ne fuient pas** (contrairement à ce qui se passait en Python 2) :

```
In [14]: # on détruit la variable i si elle existe
         if 'i' in locals():
             del i
```

```
In [15]: # en Python 3, les variables de compréhension ne fuitent pas
         [i**2 for i in range(3)]
```

```
Out[15]: [0, 1, 4]
```

```
In [16]: # ici i est à nouveau indéfinie
         try:
             i
         except NameError as e:
             print("OOPS", e)
```

```
OOPS name 'i' is not defined
```

4.16 L'exception `UnboundLocalError`

4.16.1 Complément - niveau intermédiaire

Nous résumons ici quelques cas simples de portée de variables.

Variable locale

Les **arguments** attendus par la fonction sont considérés comme des variables **locales**, c'est-à-dire dans l'espace de noms de la fonction.

Pour définir une autre variable locale, il suffit de la définir (l'affecter), elle devient alors accessible en lecture :

```
In [1]: def ma_fonction1():
            variable1 = "locale"
            print(variable1)

        ma_fonction1()
```

```
locale
```

et ceci que l'on ait ou non une variable globale de même nom

```
In [2]: variable2 = "globale"

        def ma_fonction2():
            variable2 = "locale"
            print(variable2)

        ma_fonction2()
```

```
locale
```

Variable globale

On peut accéder **en lecture** à une variable globale sans précaution particulière :

```
In [3]: variable3 = "globale"

        def ma_fonction3():
            print(variable3)

        ma_fonction3()
```

```
globale
```

Mais il faut choisir!

Par contre on ne **peut pas** faire la chose suivante dans une fonction. On ne peut pas utiliser **d'abord** une variable comme une variable **globale**, **puis** essayer de l'affecter localement - ce qui signifie la déclarer comme une **locale** :

```
In [4]: # cet exemple ne fonctionne pas et lève UnboundLocalError
        variable4 = "globale"

        def ma_fonction4():
            # on référence la variable globale
            print(variable4)
            # et maintenant on crée une variable locale
            variable4 = "locale"

        # on "attrape" l'exception
        try:
            ma_fonction4()
        except Exception as e:
            print(f"OOPS, exception {type(e)}:\n{e}")
```

```
OOPS, exception <class 'UnboundLocalError'>:
local variable 'variable4' referenced before assignment
```

Comment faire alors ?

L'intérêt de cette erreur est d'interdire de mélanger des variables locales et globales de même nom dans une même fonction. On voit bien que ça serait vite incompréhensible. Donc une variable dans une fonction peut être **ou bien** locale si elle est affectée dans la fonction **ou bien** globale, mais **pas les deux à la fois**. Si vous avez une erreur `UnboundLocalError`, c'est qu'à un moment donné vous avez fait cette confusion.

Vous vous demandez peut-être à ce stade, mais comment fait-on alors pour modifier une variable globale depuis une fonction ? Pour cela il faut utiliser l'instruction `global` comme ceci :

```
In [5]: # Pour résoudre ce conflit il faut explicitement
        # déclarer la variable  comme globale
        variable5 = "globale"

        def ma_fonction5():
            global variable5
            # on référence la variable globale
            print("dans la fonction", variable5)
            # cette fois on modifie la variable globale
            variable5 = "changée localement"

        ma_fonction5()
        print("après la fonction", variable5)
```

```
dans la fonction globale
après la fonction changée localement
```

Nous reviendrons plus longuement sur l'instruction `global` dans la prochaine vidéo.

Bonnes pratiques

Cela étant dit, l'utilisation de variables globales est généralement considérée comme une mauvaise pratique.

Le fait d'utiliser une variable globale en *lecture seule* peut rester acceptable, lorsqu'il s'agit de matérialiser une constante qu'il est facile de changer. Mais dans une application aboutie, ces constantes elles-mêmes peuvent être modifiées par l'utilisateur via un système de configuration, donc on préférera passer en argument un objet *config*.

Et dans les cas où votre code doit recourir à l'utilisation de l'instruction `global`, c'est très probablement que quelque chose peut être amélioré au niveau de la conception de votre code.

Il est recommandé, au contraire, de passer en argument à une fonction tout le contexte dont elle a besoin pour travailler ; et à l'inverse d'utiliser le résultat d'une fonction plutôt que de modifier une variable globale.

4.17 Les fonctions `globals` et `locals`

4.17.1 Complément - niveau intermédiaire

Un exemple

python fournit un accès à la liste des noms et valeurs des variables visibles à cet endroit du code. Dans le jargon des langages de programmation on appelle ceci **l'environnement**.

Cela est fait grâce aux fonctions *builtins* `globals` et `locals`, que nous allons commencer par essayer sur quelques exemples. Nous avons pour cela écrit un module dédié :

```
In [1]: import env_locals_globals
```

Dont voici le code

```
In [2]: from modtools import show_module
        show_module(env_locals_globals)

Fichier /home/jovyan/modules/env_locals_globals.py
----------------------------------------
|"""
 |un module pour illustrer les fonctions globals et locals
 |"""
 |
 |globale = "variable globale au module"
 |
 |def display_env(env):
 |    """
 |    affiche un environnement
 |    on affiche juste le nom et le type de chaque variable
 |    """
 |    for variable, valeur in sorted(env.items()):
 |        print("{:>20} → {}".format(variable, type(valeur).__name__))
 |
 |def temoin(x):
 |    "la fonction témoin"
 |    y = x ** 2
 |    print(20 * '-', 'globals:')
 |    display_env(globals())
 |    print(20 * '-', 'locals:')
 |    display_env(locals())
 |
 |class Foo:
 |    "une classe vide"
```

et voici ce qu'on obtient lorsqu'on appelle

```
In [3]: env_locals_globals.temoin(10)

-------------------- globals:
                 Foo → type
        __builtins__ → dict
```

```
          __cached__ → str
             __doc__ → str
            __file__ → str
          __loader__ → SourceFileLoader
            __name__ → str
         __package__ → str
            __spec__ → ModuleSpec
         display_env → function
             globale → str
              temoin → function
------------------- locals:
                   x → int
                   y → int
```

Interprétation

Que nous montre cet exemple ?

— D'une part la fonction `globals` nous donne la liste des symboles définis au niveau de **l'espace de noms du module**. Il s'agit évidemment du module dans lequel est définie la fonction, pas celui dans lequel elle est appelée. Vous remarquerez que ceci englobe **tous** les symboles du module `env_locals_globals`, et non pas seulement ceux définis avant `temoin`, c'est-à-dire la variable `globale`, les deux fonctions `display_env` et `temoin`, et la classe `Foo`.

— D'autre part `locals` nous donne les variables locales qui sont accessibles **à cet endroit du code**, comme le montre ce second exemple qui se concentre sur `locals` à différents points d'une même fonction.

```
In [4]: import env_locals
```

```
In [5]: # le code de ce module
        show_module(env_locals)
```

```
Fichier /home/jovyan/modules/env_locals.py
------------------------------------------
|"""
 |un module pour illustrer la fonction locals
 |"""
 |
 |# pour afficher
 |from env_locals_globals import display_env
 |
 |def temoin(x):
 |    "la fonction témoin"
 |    y = x ** 2
 |    print(20*'-', 'locals - entrée:')
 |    display_env(locals())
 |
 |    for i in (1,):
 |        for j in (1,):
 |            print(20*'-', 'locals - boucles for:')
 |            display_env(locals())
```

```
|
```

```
In [6]: env_locals.temoin(10)

-------------------- locals - entrée:
                   x → int
                   y → int
-------------------- locals - boucles for:
                   i → int
                   j → int
                   x → int
                   y → int
```

4.17.2 Complément - niveau avancé

NOTE : cette section est en pratique devenue obsolète maintenant que les *f-strings* sont présents dans la version 3.6.

Nous l'avons conservée pour l'instant toutefois, pour ceux d'entre vous qui ne peuvent pas encore utiliser les *f-strings* en production. N'hésitez pas à passer si vous n'êtes pas dans ce cas.

Usage pour le formatage de chaînes

Les deux fonctions `locals` et `globals` ne sont pas d'une utilisation très fréquente. Elles peuvent cependant être utiles dans le contexte du formatage de chaînes, comme on peut le voir dans les deux exemples ci-dessous.

Avec `format`

On peut utiliser `format` qui s'attend à quelque chose comme :

```
In [7]: "{nom}".format(nom="Dupont")

Out[7]: 'Dupont'
```

que l'on peut obtenir de manière équivalente, en anticipant sur la prochaine vidéo, avec le passage d'arguments en `**` :

```
In [8]: "{nom}".format(**{'nom': 'Dupont'})

Out[8]: 'Dupont'
```

En versant la fonction `locals` dans cette formule on obtient une forme relativement élégante

```
In [9]: def format_et_locals(nom, prenom, civilite, telephone):
            return "{civilite} {prenom} {nom} : Poste {telephone}".format(**locals())

        format_et_locals('Dupont', 'Jean', 'Mr', '7748')

Out[9]: 'Mr Jean Dupont : Poste 7748'
```

Avec l'opérateur %

De manière similaire, avec l'opérateur % - dont nous rappelons qu'il est obsolète - on peut écrire

```
In [10]: def pourcent_et_locals(nom, prenom, civilite, telephone):
             return "%(civilite)s %(prenom)s %(nom)s : Poste %(telephone)s"%locals()

         pourcent_et_locals('Dupont', 'Jean', 'Mr', '7748')

Out[10]: 'Mr Jean Dupont : Poste 7748'
```

Avec un *f-string*

Pour rappel si vous disposez de python 3.6, vous pouvez alors écrire simplement - et sans avoir recours, donc, à `locals()` ou autre :

```
In [11]: # attention ceci nécessite python-3.6
         def avec_f_string(nom, prenom, civilite, telephone):
             return f"{civilite} {prenom} {nom} : Poste {telephone}"

         avec_f_string('Dupont', 'Jean', 'Mr', '7748')

Out[11]: 'Mr Jean Dupont : Poste 7748'
```

4.18 Passage d'arguments

4.18.1 Complément - niveau intermédiaire

Motivation

Jusqu'ici nous avons développé le modèle simple qu'on trouve dans tous les langages de programmation, à savoir qu'une fonction a un nombre fixe, supposé connu, d'arguments. Ce modèle a cependant quelques limitations ; les mécanismes de passage d'arguments que propose python, et que nous venons de voir dans les vidéos, visent à lever ces limitations.

Voyons de quelles limitations il s'agit.

Nombre d'arguments non connu à l'avance

Ou encore : introduction à la forme `*arguments`

Pour prendre un exemple aussi simple que possible, considérons la fonction `print`, qui nous l'avons vu, accepte un nombre quelconque d'arguments.

```
In [1]: print("la fonction", "print", "peut", "prendre", "plein", "d'arguments")
```

```
la fonction print peut prendre plein d'arguments
```

Imaginons maintenant que nous voulons implémenter une variante de `print`, c'est-à-dire une fonction `error`, qui se comporte exactement comme `print` sauf qu'elle ajoute en début de ligne une balise `ERROR`.

Se posent alors deux problèmes :

— D'une part il nous faut un moyen de spécifier que notre fonction prend un nombre quelconque d'arguments.

— D'autre part il faut une syntaxe pour repasser tous ces arguments à la fonction `print`.

On peut faire tout cela avec la notation en `*` comme ceci :

```
In [2]: # accepter n'importe quel nombre d'arguments
        def error(*print_args):
            # et les repasser à l'identique à print en plus de la balise
            print('ERROR', *print_args)

        # on peut alors l'utiliser comme ceci
        error("problème", "dans", "la", "fonction", "foo")
        # ou même sans argument
        error()
```

```
ERROR problème dans la fonction foo
ERROR
```

Légère variation

Pour sophistiquer un peu cet exemple, on veut maintenant imposer à la fonction erreur qu'elle reçoive un argument obligatoire de type entier qui représente un code d'erreur, plus à nouveau un nombre quelconque d'arguments pour `print`.

Pour cela, on peut définir une signature (les paramètres de la fonction) qui

— prévoit un argument traditionnel en première position, qui sera obligatoire lors de l'appel,

— et le tuple des arguments pour `print`, comme ceci :

```
In [3]: # le premier argument est obligatoire
        def error1(error_code, *print_args):
            message = f"message d'erreur code {error_code}"
            print("ERROR", message, '--', *print_args)

        # que l'on peut à présent appeler comme ceci
        error1(100, "un", "petit souci avec", [1, 2, 3])
```

```
ERROR message d'erreur code 100 -- un petit souci avec [1, 2, 3]
```

Remarquons que maintenant la fonction `error1` ne peut plus être appelée sans argument, puisqu'on a mentionné un paramètre **obligatoire** `error_code`.

Ajout de fonctionnalités

Ou encore : la forme `argument=valeur_par_defaut`

Nous envisageons à présent le cas - tout à fait indépendant de ce qui précède - où vous avez écrit une librairie graphique, dans laquelle vous exposez une fonction `ligne` définie comme suit. Évidemment pour garder le code simple, nous imprimons seulement les coordonnées du segment :

```
In [4]: # première version de l'interface pour dessiner une ligne
        def ligne(x1, y1, x2, y2):
            "dessine la ligne (x1, y1) -> (x2, y2)"
            # restons simple
            print(f"la ligne ({x1}, {y1}) -> ({x2}, {y2})")
```

Vous publiez cette librairie en version 1, vous avez des utilisateurs ; et quelque temps plus tard vous écrivez une version 2 qui prend en compte la couleur. Ce qui vous conduit à ajouter un paramètre pour `ligne`.

Si vous le faites en déclarant

```
def ligne(x1, y1, x2, y2, couleur):
    ...
```

alors tous les utilisateurs de la version 1 vont **devoir changer leur code** - pour rester à fonctionnalité égale - en ajoutant un cinquième argument `'noir'` à leurs appels à `ligne`.

Vous pouvez éviter cet inconvénient en définissant la deuxième version de `ligne` comme ceci :

```
In [5]: # deuxième version de l'interface pour dessiner une ligne
        def ligne(x1, y1, x2, y2, couleur="noir"):
            "dessine la ligne (x1, y1) -> (x2, y2) dans la couleur spécifiée"
            # restons simple
            print(f"la ligne ({x1}, {y1}) -> ({x2}, {y2}) en {couleur}")
```

Avec cette nouvelle définition, on peut aussi bien

```
In [6]: # faire fonctionner du vieux code sans le modifier
        ligne(0, 0, 100, 100)
        # ou bien tirer profit du nouveau trait
        ligne(0, 100, 100, 0, 'rouge')
```

```
la ligne (0, 0) -> (100, 100) en noir
la ligne (0, 100) -> (100, 0) en rouge
```

Les paramètres par défaut sont très utiles

Notez bien que ce genre de situation peut tout aussi bien se produire sans que vous ne publiiez de librairie, à l'intérieur d'une seule application. Par exemple, vous pouvez être amené à ajouter un argument à une fonction parce qu'elle doit faire face à de nouvelles situations imprévues, et que vous n'avez pas le temps de modifier tout le code.

Ou encore plus simplement, vous pouvez choisir d'utiliser ce passage de paramètres dès le début de la conception; une fonction `ligne` réaliste présentera une interface qui précise les points concernés, la couleur du trait, l'épaisseur du trait, le style du trait, le niveau de transparence, etc. Il n'est vraiment pas utile que tous les appels à `ligne` reprécisent tout ceci intégralement, aussi une bonne partie de ces paramètres seront très constructivement déclarés avec une valeur par défaut.

4.18.2 Complément - niveau avancé

Écrire un wrapper

Ou encore : la forme `**keywords`

La notion de *wrapper* - emballage, en anglais - est très répandue en informatique, et consiste, à partir d'un morceau de code souche existant (fonction ou classe) à définir une variante qui se comporte comme la souche, mais avec quelques légères différences.

La fonction `error` était déjà un premier exemple de *wrapper*. Maintenant nous voulons définir un *wrapper* `ligne_rouge`, qui sous-traite à la fonction `ligne` mais toujours avec la couleur rouge.

Maintenant que l'on a injecté la notion de paramètre par défaut dans le système de signature des fonctions, se repose la question de savoir comment passer à l'identique les arguments de `ligne_rouge` à `ligne`.

Évidemment, une première option consiste à regarder la signature de `ligne` :

```
def ligne(x1, y1, x2, y2, couleur="noir")
```

Et à en déduire une implémentation de `ligne_rouge` comme ceci

```
In [7]: # la version naïve - non conseillée - de ligne_rouge
        def ligne_rouge(x1, y1, x2, y2):
            return ligne(x1, y1, x2, y2, couleur='rouge')

        ligne_rouge(0, 0, 100, 100)
```

```
la ligne (0, 0) -> (100, 100) en rouge
```

Toutefois, avec cette implémentation, si la signature de `ligne` venait à changer, on serait vraisemblablement amené à changer **aussi** celle de `ligne_rouge`, sauf à perdre en fonctionnalité. Imaginons en effet que `ligne` devienne dans une version suivante

```
In [8]: # on ajoute encore une fonctionnalité à la fonction ligne
        def ligne(x1, y1, x2, y2, couleur="noir", epaisseur=2):
            print(f"la ligne ({x1}, {y1}) -> ({x2}, {y2})"
                  f" en {couleur} - ep. {epaisseur}")
```

Alors le wrapper ne nous permet plus de profiter de la nouvelle fonctionnalité. De manière générale, on cherche au maximum à se prémunir contre de telles dépendances. Aussi, il est de beaucoup préférable d'implémenter `ligne_rouge` comme suit, où vous remarquerez que **la seule hypothèse** faite sur `ligne` est qu'elle accepte un argument nommé `couleur`.

```
In [9]: def ligne_rouge(*arguments, **keywords):
            # c'est le seul endroit où on fait une hypothèse sur la fonction `ligne`
            # qui est qu'elle accepte un argument nommé 'couleur'
            keywords['couleur'] = "rouge"
            return ligne(*arguments, **keywords)
```

Ce qui permet maintenant de faire

```
In [10]: ligne_rouge(0, 100, 100, 0, epaisseur=4)
```

```
la ligne (0, 100) -> (100, 0) en rouge - ep. 4
```

Pour en savoir plus - la forme générale

Une fois assimilé ce qui précède, vous avez de quoi comprendre une énorme majorité (99% au moins) du code Python.

Dans le cas général, il est possible de combiner les 4 formes d'arguments :

— arguments "normaux", dits positionnels

— arguments nommés, comme `nom=<valeur>`

— forme `*args`

— forme `**dargs`

Vous pouvez vous reporter à cette page pour une description détaillée de ce cas général.

À l'appel d'une fonction, il faut résoudre les arguments, c'est-à-dire associer une valeur à chaque paramètre formel (ceux qui apparaissent dans le `def`) à partir des valeurs figurant dans l'appel.

L'idée est que pour faire cela, les arguments de l'appel ne sont pas pris dans l'ordre où ils apparaissent, mais les arguments positionnels sont utilisés en premier. La logique est que, naturellement les arguments positionnels (ou ceux qui proviennent d'une `*expression`) viennent sans nom, et donc ne peuvent pas être utilisés pour résoudre des arguments nommés.

Voici un tout petit exemple pour vous donner une idée de la complexité de ce mécanisme lorsqu'on mélange toutes les 4 formes d'arguments à l'appel de la fonction (alors qu'on a défini la fonction avec 4 paramètres positionnels)

```
In [11]: # une fonction qui prend 4 paramètres simples
         def foo(a, b, c, d):
             print(a, b, c, d)

In [12]: # on peut l'appeler par exemple comme ceci
         foo(1, c=3, *(2,), **{'d':4})
```

```
1 2 3 4
```

```
In [13]: # mais pas comme cela
         try:
             foo (1, b=3, *(2,), **{'d':4})
         except Exception as e:
             print(f"OOPS, {type(e)}, {e}")
```

```
OOPS, <class 'TypeError'>, foo() got multiple values for argument 'b'
```

Si le problème ne vous semble pas clair, vous pouvez regarder la documentation python décrivant ce problème.

4.19 Un piège courant

4.19.1 Complément - niveau basique

N'utilisez pas d'objet mutable pour les valeurs par défaut

En Python il existe un piège dans lequel il est très facile de tomber. Aussi si vous voulez aller à l'essentiel : **n'utilisez pas d'objet mutable pour les valeurs par défaut** lors de la définition d'une fonction.

Si vous avez besoin d'écrire une fonction qui prend en argument par défaut une liste ou un dictionnaire vide, voici comment faire

```
In [1]: # ne faites SURTOUT PAS ça
        def ne_faites_pas_ca(options={}):
            "faire quelque chose"

In [2]: # mais plutôt comme ceci
        def mais_plutot_ceci(options=None):
            if options is None:
                options = {}
            "faire quelque chose"
```

4.19.2 Complément - niveau intermédiaire

Que se passe-t-il si on le fait ?

Considérons le cas relativement simple d'une fonction qui calcule une valeur - ici un entier aléatoire entre 0 et 10 -, et l'ajoute à une liste passée par l'appelant.

Et pour rendre la vie de l'appelant plus facile, on se dit qu'il peut être utile de faire en sorte que si l'appelant n'a pas de liste sous la main, on va créer pour lui une liste vide. Et pour ça on fait :

```
In [3]: import random

        # l'intention ici est que si l'appelant ne fournit pas
        # la liste en entrée, on crée pour lui une liste vide
        def ajouter_un_aleatoire(resultats=[]):
            resultats.append(random.randint(0, 10))
            return resultats
```

Si on appelle cette fonction une première fois, tout semble bien aller

```
In [4]: ajouter_un_aleatoire()

Out[4]: [6]
```

Sauf que, si on appelle la fonction une deuxième fois, on a une surprise !

```
In [5]: ajouter_un_aleatoire()

Out[5]: [6, 9]
```

Pourquoi ?

Le problème ici est qu'une valeur par défaut - ici l'expression `[]` - est évaluée **une fois** au moment de la **définition** de la fonction.

Toutes les fois où la fonction est appelée avec cet argument manquant, on va utiliser comme valeur par défaut **le même objet**, qui la première fois est bien une liste vide, mais qui se fait modifier par le premier appel.

Si bien que la deuxième fois on réutilise la même liste **qui n'est plus vide**. Pour aller plus loin, vous pouvez regarder la documentation Python sur ce problème.

4.20 Arguments *keyword-only*

4.20.1 Complément - niveau intermédiaire

Rappel

Nous avons vu dans un précédent complément les 4 familles de paramètres qu'on peut déclarer dans une fonction :

1. paramètres positionnels (usuels)
2. paramètres nommés (forme *name=default*)
3. paramètres **args* qui attrape dans un tuple le reliquat des arguments positionnels
4. paramètres ***kwds* qui attrape dans un dictionnaire le reliquat des arguments nommés

Pour rappel :

```
In [1]: # une fonction qui combine les différents
        # types de paramètres
        def foo(a, b=100, *args, **kwds):
            print(f"a={a}, b={b}, args={args}, kwds={kwds}")

In [2]: foo(1)

a=1, b=100, args=(), kwds={}


In [3]: foo(1, 2)

a=1, b=2, args=(), kwds={}


In [4]: foo(1, 2, 3)

a=1, b=2, args=(3,), kwds={}


In [5]: foo(1, 2, 3, bar=1000)

a=1, b=2, args=(3,), kwds={'bar': 1000}
```

Un seul paramètre attrape-tout

Notez également que, de bon sens, on ne peut déclarer qu'un seul paramètre de chacune des formes d'attrape-tout ; on ne peut pas par exemple déclarer

```
# c'est illégal de faire ceci
def foo(*args1, *args2):
    pass
```

car évidemment on ne saurait pas décider de ce qui va dans `args1` et ce qui va dans `args2`.

Ordre des déclarations

L'ordre dans lequel sont déclarés les différents types de paramètres d'une fonction est imposé par le langage. Ce que vous avez peut-être en tête si vous avez appris **Python 2**, c'est qu'à l'époque on devait impérativement les déclarer dans cet ordre :

positionnels, nommés, forme `*`, forme `**`

comme dans notre fonction `foo`.

Ça reste une bonne approximation, mais depuis Python-3, les choses ont un petit peu changé suite à l'adoption du PEP 3102, qui vise à introduire la notion de paramètre qu'il faut impérativement nommer lors de l'appel (en anglais : *keyword-only* argument)

Pour résumer, il est maintenant possible de déclarer des **paramètres nommés après la forme `*`**

Voyons cela sur un exemple

```
In [6]: # on peut déclarer un paramètre nommé **après** l'attrape-tout *args
        def bar(a, *args, b=100, **kwds):
                print(f"a={a}, b={b}, args={args}, kwds={kwds}")
```

L'effet de cette déclaration est que, si je veux passer un argument au paramètre `b`, **je dois le nommer**

```
In [7]: # je peux toujours faire ceci
        bar(1)

a=1, b=100, args=(), kwds={}


In [8]: # mais si je fais ceci l'argument 2 va aller dans args
        bar(1, 2)

a=1, b=100, args=(2,), kwds={}


In [9]: # pour passer b=2, je **dois** nommer mon argument
        bar(1, b=2)

a=1, b=2, args=(), kwds={}
```

Ce trait n'est objectivement pas utilisé massivement en Python, mais cela peut être utile de le savoir :

— en tant qu'utilisateur d'une bibliothèque, car cela vous impose une certaine façon d'appeler une fonction ;
— en tant que concepteur d'une fonction, car cela vous permet de manifester qu'un paramètre optionnel joue un rôle particulier.

4.21 Passage d'arguments

4.21.1 Exercice - niveau basique

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

Vous devez écrire une fonction `distance` qui prend un nombre quelconque d'arguments numériques non complexes, et qui retourne la racine carrée de la somme des carrés des arguments.

Plus précisément : *distance* $(x_1, \ldots, x_n) = \sqrt{\sum x_i^2}$

Par convention on fixe que *distance* () = 0

```
In [ ]: # des exemples
        exo_distance.example()
```

```
In [ ]: # ATTENTION vous devez aussi définir les arguments de la fonction
        def distance(votre, signature):
            return "votre code"
```

```
In [ ]: # la correction
        exo_distance.correction(distance)
```

4.21.2 Exercice - niveau intermédiaire

```
In [ ]: # Pour charger l'exercice
        from corrections.exo_numbers import exo_numbers
```

On vous demande d'écrire une fonction `numbers`

- qui prend en argument un nombre quelconque d'entiers,
- et qui retourne un tuple contenant
  - la somme
  - le minimum
  - le maximum de ses arguments.

Si aucun argument n'est passé, `numbers` doit renvoyer un tuple contenant 3 entiers `0`.

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

En guise d'indice, je vous invite à regarder les fonctions *built-in* `sum`, `min` et `max`.

```
In [ ]: # vous devez définir votre propre signature
        def numbers(votre, signature):
            "<votre_code>"
```

```
In [ ]: # pour vérifier votre code
        exo_numbers.correction(numbers)
```