Programmer en sécurité : Types, exception et test

Thibault PENNING

Lorsque l'on programme, il arrive souvent de faire des erreurs. Un bon programmeur se doit d'essayer de les prévenir aux maximums. Lorsqu'elle arrive en production ces dernières peuvent avoir des conséquences catastrophiques. L'industrie, chercheur et programmeur ont donc mis au point certaines techniques afin de se prémunir au plus possible de possibles bugs.

Les types en Python

Le type c'est quoi ?

Définition 0.1 (Type)**.** Un type (de données) en programmation désigne la nature d'une donnée stockée.

Ainsi le type donne sens à une donnée (écrit en binaire): il définit son sens, sa représentation humaine, les différentes manipulations possibles, …

Il est important de comprendre qu'une même donnée stockée peut être vu différemment en fonction de son type.

Ainsi l'octet `0b1100001` peut être vu comme 97 en entier, ou le caractère `a` en fonction de son type. De même manière, faire une addition avec `0b1100010` donnera $97 + 98 = 195$ si les données sont entières ou `"ab"` si on utilise des caractères.

On remarque ainsi que le type de donnée à une importance capitale au bon fonctionnement d'un programme, et que l'ordinateur, manipulant les bits, doit croire sur parole le programmeur. Si ce dernier se trompe, un bug existe, bug que l'ordinateur seul ne peut résoudre.

A contrario de nombreux langage, Python utilise un système de typage dynamique. Ainsi, c'est lors de l'exécution du programme qu'un type sera assigner à une valeur, et chaque variable n'est pas assignée à un type en particulier. Cela permet de ne pas se soucier du type de variable lors de l'écriture du code et ainsi de gagner en temps et en flexibilité. Mais les type en Python

sont toujours présents. Il est d'ailleurs possible d'utilisé la fonction `type` pour obtenir le type d'une valeur. Exemple :

Exemple 0.1.

```
>>> type(1000)
<class 'int'>
>>> a = "Hello"
>>> type(a)
<class 'str'>
```

Ici les deux types sont `int` pour un entier et `str` pour une chaine de caractère (un string en anglais) (ne vous concentrez pas sur le début avec le mot clef `class`, nous verrons cela en programmation orienté objet).

Les type intégré

Il existe plus de type Python. Une liste non exhaustive inclus :

Table 1: Liste partielle des types Python

Exemple	Type	Description
`100`	int	Entier (signé)
`1.0`	float	Flottant (double précision de la norme IEEE 754)
`1.0j`	complex	Nombre complexe (très peu utilisé)
`True`	bool	Valeur booléen (`True` ou `False`)
`"Salut !`	str	Chaine de caractère
None	NoneType	Type de None
`[1,2]`	list	Tableau (attention ce n'est pas une liste !)
`{1,2}`	set	Ensemble (pas d'index, pas de répétition)
`{a:1, b:2}`	dict	Dictionaire
(1, 2)	tuple	Un tuple (index, séquentiel mais immuable)
`b'\xf0\xf1'`	bytes	Tableau d'octets (très peu utilisé, utile pour les chaines de caractères)

> **NoneType ?**
>
> `None` en Python désigne une valeur non définie. Il est souvent utilisé comme type de retour de fonction ne retournant rien, et peut servir comme marqueur, afin de préciser qu'une

variable n'est pas initialisé par exemple.
`None` est de type `NoneType`, et est la seule valeur possible de ce type. Il n'existe une seule instance de `None` en Python, c'est ce que l'on appelle un **Singleton**. L'effet est que tester si une valeur est à `None` ne se fait pas avec l'opérateur habituel `==` mais avec le mot clef `is` (plus d'info sur Section).

```
if var is None:
  # Si var à la valeur None
else:
  # Sinon
```

Comme le typage est celui-ci, un code comme celui-ci est totalement possible :

```
text = "Hello"  ①
text = 1        ②
text += 2
```

① Le type stocké dans `text` est `str`
② Le type stocké dans `text` est `int`

Cependant si nous essayons l'inverse :

```
text = 1
text = "Hello"
text += 2

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Le programme s'arrête et un message s'affiche. Nous expliquerons ce mécanisme après, mais remarquer l'erreur qui nous est indiquée : Python considère que `text` est un `str`, le `+` est donc celui de la concaténation. Mais concaténer un `str` et un `int` n'est pas définie.

> **!** Important
>
> Si possible éviter de mélanger les types. Si une variable possède un type, rester sur ce dernier tout le long du code.

Évidement dans certain cas nous sommes content du typage dynamique :

Exemple 0.2 (Cas où le typage dynamique est utile)**.**

```
a = 3
a /= 2
```

Mais cela doit rester le plus rare possible, car cela pourrait produire des comportements non prévus.

Typage des variables et des fonctions

Définition 0.2 (Prototype d'une fonction)**.** Le prototype d'une fonction est la définition du squelette d'une fonction, sans son contenu. Il est opposé à son implémentation.

Ainsi le prototype d'une fonction se compose du nom de la fonction, du nom des variables et de leur type, ainsi que le type de retour. Si une description existe, on considérera qu'elle fait partie du prototype.

Le prototype d'une fonction permet de voir la fonction comme une boite noire : "Peu importe ce qu'il y a dans la fonction, si je lui donne telle valeur j'obtiens ceci".

Ainsi le prototype d'une fonction est le seul contenu nécessaire pour utiliser une fonction. Lorsque vous utilisez `print` savez-vous quel code est dans la fonction ? Non, mais vous connaissez son prototype et cela vous suffit.

Le cas général

En Python, si le typage est dynamique, il est malgré tout possible d'utiliser des indications de type (ou "type hints" dans la lange de molière). Ces indications de type sont a priori purement cosmétique et CPython, l'implémentation standard de Python ne les utilise pas lors de l'interprétation.

> **Note**
>
> Le typage est apparu avec Python 3.5, et de nombreuse amélioration ont vu le jour avec Python 3.9, 3.10 et 3.12. Ce sujet est encore sujet a changement et il se peut que certaine information présentée ne fonctionne pas sur une version trop ancienne (ou récente) de Python.

Afin d'indiquer le type d'une variable on utilise le caractère `:` comme présenter ci-dessous

```
a:int
```

On indique ici explicitement notre volonté que `a` soit exclusivement un entier.

Bien évidement cette syntaxe fonctionne aussi lors d'affectation si besoin :

```
a:float = 1.0
```

Ici la syntaxe est simple, on utilise le nom de la variable, suivie du type séparé par un `:`.

Parmi les types possiblement mis après les `:`, ceux de la Table 1. D'autre possibilité sont offertes et nous les verrons juste après

Pour une fonction, la syntaxe des arguments est la même que celle des variables, auquel on rajoute une syntaxe pour le type de retour. Voici le prototype d'une fonction :

```
def addition(a:int, b:int) -> int:
  pass
```

Ici le type de retour est indiqué après la `->`.

Une fonction qui ne retourne rien, retourne en réalité `None`. Ainsi le prototype suivant est correct :

Exemple 0.3.

```
def salutation(nom:str)->None:
  print(f"Bonjour, {nom}!")
```

> **Note**
>
> Remarquer que si `None` a un type `NoneType` cette syntaxe reste juste car Python nous offre ce sucre syntaxique, dû au fait que `None` est un singleton.

Le cas des conteneurs

Bien évidement la syntaxe s'étend au tableau, tuple et autre conteneur.

Ainsi pour les tableau la syntaxe suivante est valide :

Exemple 0.4.

```
l1:list = [1, 2, 3]
l2:list = ["a", "b", "c"]
l3:list = [1, "b", None]
```

Cela s'étend au autre conteneur de Table 1.

Cependant si l'on regarde attentivement une erreur peu se produire. Lorsque l'on itérera sur `l3`, la variable d'itération changera de type, ce qui peu nous causer des erreurs.

C'est ici qu'intervient la librairie `typing` permettant de préciser les données internes des conteneurs.

Ainsi la syntaxe suivante est correcte :

```
from typing import List                                    ①

l1:List[int] = [1, 2, 3]                                   ②
l2:List[str] = ["a", "b", "c"]                             ③
```

① On importe `List` depuis `typing`. Ceci est obligatoire !
② On indique que `l1` est une liste composée exclusivement d'entier
③ On indique que `l2` est une liste composée exclusivement de `str`

Attention le `L` majuscule de `List` est obligatoire !

- `list` = type intégré dans Python
- `List` = indication de typage, qui permet une plus grande flexibilité qu'utilisé `List` seul.

En fait, `typing` définie tous ses types par une majuscule sur la première lettre.

Ici on utilise des `[]` où l'on indique a l'intérieur le type exclusif souhaité.

Pour `set` et `dict` la syntaxe est similaire :

```
form typing import Set, Dict
s:Set[int] = {1, 2, 3}                                     ①
d:Dict[str, int] = {"a":1 , "b":2}                         ②
```

① Les données contenues dans le `set` sont des entiers
② Les clefs du dictionnaire sont des `str` et les valeurs des `int`

Pour le dictionnaire la syntaxe est donc :

```
Dict[type_clef, type_valeur]
```

Pour les tuples la syntaxe est légèrement différente, car l'on indique pour chaque élément le type possible :

```python
from typing import Tuple

t1:Tuple[int, int, int] = (1, 2, 3)
t2:Tuple[int, str, None] = (1, "a", None)
```

Si besoin, pour dénoter un tuple de taille non définie et de type constant on utilise :

```python
t3:Tuple[int, ...]
```

Plus sur les types

Il est possible de créer un alias de type, afin de donner un nom plus significatif. Ainsi la syntaxe suivante est valide :

```python
index = int                                                    ①

i:index = 0                                                    ②

l = [1, 2, 3]
l[i] #Ici cela vaut 1


# A PARTIR DE PYTHON 3.12 :
type index = int                                               ③
```

① On indique a Python que le type `index` existe et qu'il est un alias de `int`
② On peut utiliser le type normalement comme tout autre type
③ Depuis Python 3.12 le mot clef `type` (non obligatoire) sert afin d'indiquer notre intention de créer un alias (pas une véritable affectation)

> **Astuce**
>
> On peut vite se sentir limité par cette syntaxe, pensant qu'il n'est pas possible de décrire une fonction qui retourne plusieurs éléments. Il n'en est rien car une fonction qui retourne plusieurs éléments retourne en vérité un tuple.
> Ainsi la syntaxe suivante est juste :
>
> ```python
> def min_max(l1:List[int], l2:List[int]) -> Tuple[int, int]:
>   m = min(l1, l2)
>   M = max(l1,l2)
>   return m, M
> ```

Il peut aussi être utile dans certains cas de laisser la possibilité aux arguments ou au retour d'être de plusieurs types possible. Dans ce cas, on peut définir des unions de type:

```
# Syntaxe avec |
def addition(a:float|int, b:float|int) -> float|int:
  pass


# Syntaxe avec Union
from typing import Union
def addition(a:Union[float, int], b:Union[float, int]) -> Union[float, int]:
  pass
```

Il est courant que l'union soit entre un type et `None`. `typing` à inventer un type nommé `Optional` pour cela :

```
from typing import Optional

def test() -> Optional[int]:
  pass

# Équivaut à

def test() -> int | None:
  pass
```

Si l'on veut explicitement prendre en compte tous les types possibles on utilise `Any` :

```
from typing import Any
a:Any

a = 1
a = "a"
# ...
```

D'autres types peuvent être utilisés, comme `Callable` désignant des fonctions, `Never` indiquant qu'une fonction ne s'arrêtera jamais, `Self` lors de la programmation objet, `Litteral` pour donner explicitement les valeurs possibles, `Final` indiquant l'immuabilité, ou encore le décorateur `@overide` indiquant qu'il est possible qu'une fonction ait plusieurs formes.

Aussi, pour les conteneurs, une liste existe dans la documentation permettant de sélectionner précisément les fonctions souhaitez, et être le moins sélectif possible. Bien évidement, tout cela est de loin hors programme !

Teste de type et mot clef `is`

Une autre façon de s'assurer du bon fonctionnement d'un programme en utilisant le type est de définir un autre comportement en fonction du type de donnée. Contrairement aux indications de typage, ceci est exécuté en Python, et auras donc un comportement bien défini.

Pour cela on utilise un test de typage consistant en : la fonction `type` suivi du mot clef `is` suivie enfin du type à comparer (l'opérateur `==` n'est pas bien définie pour les types).

Ainsi le code suivant permet de créer une fonction adaptée à chaque type, en toute sérénité :

Exemple 0.5.

```
def magnitude(a:float|complex) -> float:
  if type(a) if float:
    return abs(a)
  elif type(a) is complex:
    from math import sqrt
    return sqrt( (a.imag)**2 + (a.real)**2 )
  else:
    # Cas d'un type non reconnu. Nous verrons cela plus tard...
    raise TypeError
```

Ce type de distinction de cas est très efficace car il permet une grande flexibilité. Cependant il peut toujours survenir un cas non-préparer. Utilisé les indications de typage permettre à chacun de prendre conscience de ce que la fonction peut ou non accepter. Les deux sont ainsi complémentaires.

> **!** Encore un détour sur `None`
>
> `None` a été introduit comme dans d'autre langage (souvent appeler `null`) afin de décrire une donnée n'ayant pas de sens.
> Cela se produit lorsque par exemple la valeur n'est pas initialisée.
> Cependant l'utilisation de `None` est très controversé. S'il est courant de retourner `None` lors d'un appel à une fonction, il est fortement aujourd'hui découragé de le faire (en dehors d'une fonction qui ne retourne rien).
> Tony Hoare, l'un des plus grands informaticien, créateur du concept de `null` a appelé son invention "Mon erreur en milliard de dollars".
> Cela est dû au fait qu'en pratique, peu de gens vérifie si une valeur est à `None` et donc de nombreux bugs apparait, certain si complexe qu'il n'arrive qu'en production.
> Les langages fonctionnels comme OCaml ou Rust utilise le principe de valeur optionnel, mais force l'utilisateur à vérifier la valeur avant utilisation, empêchant ainsi les bugs !
> Ayez l'habitude si vous voyez un type optionnel :

```
a = ma_fonction()
if a is None:
  pass
else:
  pass
```

Outils autour du type

Comme préciser dans Section , les indications ne sont qu'optionnels et n'ont pas d'impact lors de l'interprétation. Mettre des indications peut donc sembler a priori inutile ou en tout cas substituable par des commentaires bien rédiger.

Cependant certains outils permettent de prendre en compte ces indications, les rendant ainsi très utiles.

L'un de ces outils est l'analyseur statique **MyPy**. La syntaxe Python est même basée sur cet analyseur. Ce dernier vérifie le type de variable et s'assure que toutes les opérations faites serons légales.

D'autres acteurs comme Facebook ont développés leur outils maison comme `pyre`.

Ainsi ce dernier s'insurgera du code suivant :

```
a:int
a = "c"
```

Car l'on n'affecte pas la bonne valeur. MyPy regarde aussi les méthodes, ainsi il nous préviendra de cette opération illégale (car `append` n'est pas une méthode de `int`) :

```
a:int = 1
a.append(2)
```

MyPy comme analyseur statique doit être exécuté durant le développement. Ce n'est pas un interpréteur. De nombreuses intégrations avec votre éditeur existe, de telle sorte qu'il soit exécuter automatiquement et que les erreurs vous soit afficher de manière familière.

D'autre outil est souvent intégré votre IDE, il s'agit des analyseurs statique type 'linter'. Ce sont eux qui sont responsables de la complétion automatique, etc… Pylance est un bon exemple. Lors de la complétion automatique ce dernier ne vous proposera seulement les méthodes possibles.

> **Note**
>
> Python n'est pas le seul langage avec un typage dynamique. Un autre langage très utilisé (et au programme) est le JavaScript. C'est LE langage de programmation du web.
> Cependant de plus en plus de personne tend à passer vers TypeScript, une surcouche de JavaScript permettant des indications de type, et devant être retransformé en JavaScript pour être exécuté Comme sur Python les indication ne sont pas pris en charge, mais sa création par Microsoft est dû au besoin qu'ont les projets, petit comme gros, de posséder un système de typage, permettant de détecter certaines erreurs avant l'exécution.

Au delà du typage, documenter son code

Les indications de typage sont une formidable aide au développement. Coupler avec un choix de nom variable et de fonction, cela fait partie des bonnes pratiques quasi indispensable d'un projet sur du long terme. Mais une dernière brique s'ajoute : la documentation.

En effet il est très utile de commenter un programme pour expliquer certaine partie plus complexe, qui ne s'explique pas lors d'une première lecture. Savoir quoi et comment commenter s'apprend au fil du temps

Pour commenter une ligne on utilise la syntaxe suivante :

```
# Ceci est une ligne de commentaire

if cond: # Le reste de la ligne est un commentaire
  pass
```

> **Astuce**
>
> De manière générale on utilise des commentaires, comme si l'on devait parler à une personne extérieure à projet, même lorsque nous sommes le seul à pouvoir lire ces commentaires.
> Une manière simple de savoir si l'on doit mettre ou pas un commentaire consiste à se poser la question suivante : "est-ce que mon moi de 1ère comprendrait cette ligne de code facilement ?(au-delà de la syntaxe)". Si la question est non, cela mérite surement une petite explication.
> Pas assez de commentaire empêche la bonne compréhension mais trop de commentaire empêche la lisibilité !

La documentation de fonction est aussi très utile. Il permet d'expliquer le fonctionnement et le but d'une fonction sans avoir à regarder son code. Pour cela nous plaçons en Python un commentaire juste après sa définition :

```python
def ma_fonction():
  # Information importante


def ma_fonction2():
  """Une documentation de ma fonction,
  mais sur plusieurs ligne !"""
```

Ce commentaire est utilisé lors de l'écriture de votre programme dans votre IDE pour vous fournir une description, ou lors de l'appel de la fonction `help`. C'est ainsi la manière officielle de présenter au utilisateur d'une fonction, comment cette dernière peut être utilisée !

De plus un module nommer `pydoc` permet de générer une documentation sous forme de page web en utilisant en entrée du code source. Ce dernier utilise les documentations des fonctions, les indications de typage, etc… afin de générer un document permettant d'utiliser facilement vos magnifiques fonctions par un utilisateur externe !

Au-delà du typage, contrat et immuabilité

Le typage correspond à une forme de contrat passé avec l'utilisateur ! Il garantit que si l'on donne le bon type de donnée indiqué, le programme fonctionnera a priori (sauf contre-indication dans la documentation) comme prévu.

D'autre type de contrat existe, l'un des plus importants est l'immuabilité. Cette notion disque la chose suivante : ce qui retourner ne peux être modifié. Cela est très utile dans certains cas. Par exemple si vous avez une structure de donnée dont l'ordre et le nombre est important, alors utiliser un conteneur comme un tuple permet de s'assurer que cette structure sera garder et que donc il est possible de passer une fonction à une autre sans souci.

Erreur et programmation défensive

Les exceptions

Définition 0.1. Événement lié à une erreur lors de l'exécution d'un programme.

Une exception est donc souvent le fait d'un souci majeur dans l'exécution, qui demande un intérêt particulier. Cela arrive fréquemment lorsqu'un programme ne connait pas la marche à suivre lorsqu'il est dans cet état. Penser par exemple à l'ouverture d'un fichier qui n'existe pas, a l'utilisation d'une fonction inconnu, a une division par 0, un caractère Unicode inconnu, …

Définition 0.2. Une exception est dite fatal lorsqu'elle mène à l'arrêt du programme.

La plupart des exceptions en Python sont fatales. Une exception est fatale quand l'événement mène a une situation ou le programme ne peut continuer car il n'existe pas de bon moyen de passer outre. Comment faire lorsqu'une variable est inconnue ? Python décide donc de fermer le programme pour ne pas avoir un dégât qui se propage plus.

Maintenant que l'on sait à quoi nous pouvons faire face, voyons comment cela se passe en pratique…

Essayons de créer une erreur fatale :

Exemple 0.1 (Exemple d'une erreur fatale)**.**

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>                                    ②
ZeroDivisionError: division by zero                                      ①
```

① Nom de l'erreur et description. La description n'est pas obligatoire.
② Où se trouve l'erreur. Ici est indiqué le fichier (`<stdin>` indique ici que le "fichier" est simplement un programme dans la console), puis la ligne concernée (ici la première) et enfin le module s'il s'agirait d'une librairie que l'on a importée.

On remarque que Python nous informe de la localisation de l'erreur nous permettant de savoir ainsi plus précisément où se situe l'erreur. Dans les versions plus récentes de Python, la ligne suivie de chevron peut même nous afficher où dans cette ligne l'erreur apparaît. Le nom de l'erreur nous donne une information sur sa nature : division par 0. Souvent la description décris d'une manière plus compréhensible pour un humain, ce qu'il s'est passé. Il peut même préciser les valeurs qui ont pu mener au souci (même si cela n'est pas toujours possible, et c'est ici que le debugger intervient):

Exemple 0.2 (Exemple d'une erreur détaillé)**.**

```
>>> "a" + 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

La liste des exceptions native peut-être trouvée ici.

Lors de code plus compliqué, comme des bugs survenant dans les appels de fonction, Python affiche en plus ce que l'on appelle de `traceback` qui est constituées des dernières fonctions appelées, et encore en cours d'appel, permettant de savoir quel *flot d'exécution* a causé le souci.

Exemple 0.3 (Traceback avec fonction imbriquée)**.**

```
def add(a:int|float, b:int|float)->int|float:
  # Retourne a+b
  return a + b

def div(a:int|float, b:int|float)->float:
  # Retourne a/b
  return a / b

def mon_calcule(a:float|int, b:float|int)->float:
  # Retourne a/(a+b)
  return div(a, add(a, b))

a = 2
b = -2

print(f"Mon résultat : ", mon_calcule(a, b))


Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in mon_calcule
  File "<stdin>", line 3, in div
ZeroDivisionError: division by zero
```

Le `traceback` se lit ainsi : la dernière ligne est la fonction responsable de l'erreur, et toutes les lignes avant sont celle qui ont mené a cet appel. Ici `div` est responsable de l'erreur, `div` appelé par `mon_calcule` lui-même appelé dans le programme actuel. Chaque ligne d'appel est aussi documentée, permettant une véritable investigation des causes.

Gérer les exceptions

Les exceptions fatales ont de base le comportement de fermer le programme. Mais il se peut que l'on sache gérer le cas problématique. Si le fichier n'existe pas, on peut simplement le créer, ou demander à l'utilisateur de re-rentrer le nom de fichier.

Pour cela Python dispose d'un système permettant d'exécuter du code, tester si une exception est levée, et exécuter du code en fonction.

La syntaxe est la suivante :

```
try:
  # Mon code à tester
except Exception:
  # Code a executer en cas d'execption
```

Nous allons détailler cette procédure :

On commence par un bloc `try`. Ce dernier contient le code à tester. Ce dernier sera exécuter normalement, mais si une exception est levée, tout le restant du bloc ne seras pas exécuté. Si un bloc `except` existe et qu'il correspond a l'erreur, le contenu du bloc `except` seras exécuter, permettant ainsi de corriger cette dernière, l'erreur s'arrêtant net, le programme ne se coupera pas. Si aucun bloc n'est trouvé, l'erreur est propagée, jusqu'à l'arrêt (prématurée) du programme.

Le filtre du bloc `except` est dévié par l'exception définie juste après le bloc. Dans l'exemple, il s'agit de `Exception`, qui permet la prise en charge de toutes les exceptions (sauf `KeyboardInterrupt` et d'autre exception pour des raisons de praticité). Mais si nous voulions seulement prendre en charge les vision par 0, nous aurions pu utiliser `ZeroDivisionError`. De manière générale, `Exception` ne doit pas être utilisé ainsi (nous verrons après une utilité possible), on doit toujours essayer d'utiliser un filtre.

Ce mécanisme de filtre permet ainsi de prendre en compte le type d'erreur et d'adapté la résolution. On peut enchainer les filtre pour un même bloc `try`, permettant ainsi la prise en charge le plus grande possible.

```
try:
  pass
except ZeroDivisionError:
  pass
except ValueError:
  pass
```

Il est aussi possible d'utiliser un seul bloc `except` pour plusieurs erreurs (utile par exemple si plusieurs erreurs arithmétique peuvent arriver) :

```
try:
  a/b
except (ZeroDivisionError, FloatingPointError, OverflowError, ValueError):
  print("a et b ne sont pas divisibles entre eux !")
  exit()
```

> **Note**
>
> Comme le reste du bloc `try` n'est pas exécuté en cas d'exception, il convient d'y placer la quantité de code la plus faible possible, ayant un lien entre elle.
> Si besoin, vous pouvez utiliser la clause `else` d'un bloc `try`, qui est exécuté lorsque aucune erreur n'est levée, permettant ainsi de décharger le code de l'exception.

```
try:
  pass
except:
  pass
else:
  pass
```

Aussi il est tout à fait possible de faire des `try` dans des `try` (chacun avec leur propre `except`) mais surtout des `try` dans des `except` et cela permet de s'assurer que le code

Il peut être aussi très utile de sauvegarder l'erreur dans une variable. En effet, certaines erreurs peuvent contenir des informations, comme les valeurs mises en cause par exemple. Cela peut permettre de résoudre au mieux l'erreur.

Pour cela on utilise la syntaxe suivante :

```
try:
  # Mon code
except Exception as e:
  # e contient maintenant les informations de mon exception.
  e.args # Contient les informations qui ont mené à l'erreur.
         # Bien souvent malheureusement il ne s'agit que d'une chaine de caractère.
```

Certaines autres exceptions peuvent contenir plus de variable. Leur documentation donne plus d'information à leur sujet.

Il est aussi possible de comparer le type d'exception. Ce mécanisme est assez complexe et nécessite des prérequis de programmation objet, nous n'irons donc pas plus loin que le fragment de code donné plus loin. Pour cela nous utiliserons la fonction `type`.

```
try:
  pass
except Exception as e:
  if type(e) is ValueError:
    pass
```

Cela permet entre autre d'attraper un grand nombre d'erreurs (voir toutes) pour exécuter un même code permettant à l'arrêt correct du programme. Un des cas le plus utile est pour fermer un fichier ou un port ouvert !

Exemple 0.4 (Fermeture d'un fichier lors d'une exception)**.**

```
f = open("file.txt", "w")                                    ①
try:
  for i in range(10,-1,-1):
    f.write(str(10/i))                                       ②
except Exception as e:
  f.close()
  raise e                                                    ③
```

① L'ouverture d'un fichier doit toujours être fermé afin d'éviter de nombreux problèmes comme la corruption des fichiers, ou l'impossibilité d'y re-accéder.
② On sait que l'on va finir par diviser par 0
③ Ici on renvoie l'erreur. Nous verrons `raise` plus tard. Retenez que nous stoppons ici l'erreur seulement temporairement, dans le but de fermer le fichier, puis nous la relevons afin d'en informer l'utilisateur.

Il peut aussi dans certains cas être utile d'utiliser la close `finally` qui est exécuté peu importe s'il y a eu erreur ou non, permettant aussi ce genre d'opération :

Exemple 0.5 (Fermeture d'un fichier avec `finally`)**.**

```
f = open("file", "w")
try:
  for i in range(10,-1,-1):
    f.write(str(10/i))
except Exception
  print("Oulala")
finally:
  f.close()
```

Ici on ferme le fichier, il n'est donc pas utilisable après la clause `try`, où dans l'exemple précédent, le fichier est encore ouvert s'il n'y a pas eu d'exception.

Ce type de code étant très courant, Python à donner une syntaxe très utile remplaçant en partie ce code (et même plus !), appelé *Context Manager*. La syntaxe est la suivante :

Exemple 0.6 (Ouverture d'un fichier avec un gestionnaire de contexte)**.**

```
with open("file", "w") as f:
  f.write("Salut !")
```

Ouvrir un fichier avec `with` devrait maintenant être de l'ordre du réflexe !

Indiquer a l'utilisateur qu'une erreur s'est produite

Nous avons vu comment gérer les erreurs qui nous sont levées, mais nous n'avons pas encore vu comment informer l'utilisateur d'une erreur.

C'est ici le but de la programmation défensive :

Définition 0.3 (Programmation défensive)**.** C'est un principe qui consiste à considérer qu'une erreur arrivera, et donc de prévenir d'une possible erreur avant qu'elle n'arrive.

Ainsi, il est utile lors de la conception de code de se poser la question de cas posant un problème, de tester la situation et de lever une exception.

Cette philosophie évite une erreur, en la prenant en considération avant qu'elle ne puisse.

Pour lever une erreur on utilise le mot clef `raise` suivie du nom de l'erreur.

Exemple 0.7.

```
raise ValueError
```

Dans le cas général l'erreur est considéré comme une fonction, et ce afin de donner plus d'information sur l'erreur en argument :

Exemple 0.8.

```
raise NameError('a est inconnue')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: a est inconnue
```

Il est souvent compliqué de savoir quelle exception lever. La liste des exceptions native peut nous aider à choisir laquelle utilisée. De manière générale, les plus communs sont `ValueError` et `TypeError`.

Une bonne méthode consiste à reproduire une erreur similaire (mais pas exact) dans une fonction déjà développer et de noter quelle erreur et quelles informations sont passées en argument.

> **Note**
>
> Il est aussi possible de définir ses propres exceptions. Pour cela il faut créer une nouvelle classe qui hérite de la classe `Exception`.
> Si cela ressemble à du charabia, c'est normal, ceci est de la programmation objet, qui sera vue dans un prochain chapitre !

Une dernière erreur très commune est `NotImplementedError`. Cette erreur design une fonction non-implémenter. Elle a pour but d'être remplis dans un cas dont le code n'a pas encore été écris, mais l'implémentation est prévue. Vous l'aurez compris cette erreur sert au développeur de marquer cette partie comme *À faire*. C'est donc une erreur très commune a utilisé, mais cette dernière n'est pas censée se trouver en production, si une chose n'est pas encore implémenter, alors ce cas ne doit pas apparaitre (car l'erreur n'apparaitra que lors de l'exécution, et non lors de l'écriture, un développeur consciencieux peut donc se retrouver fautif sans le savoir). Une autre utilité est pour marquer une méthode comme abstraite (nous verrons cela en orienté objet !).

Exemple 0.9.

```
def ma_super_fonction():
  # TODO
  raise NotImplementedError
```

Une méthode simple de détection d'erreur : les `assert`

Comme préciser dans la Section , il est important d'indiquer à l'utilisateur lorsqu'une erreur s'est produite. Le souci avec `raise`, c'est qu'il nécessite de choisir l'erreur appropriée, remplir les arguments etc, et ceci décorréler des tests, ce qui complique la vie et il arrive souvent qu'aucune gestion d'erreur ne soit mis en place, et que cette dernière ne voit jamais le jour car ont finis pas oublier de le faire…

Pour cela Python nous donne accès à une fonction `assert` qui prend en entrée un booléen (donner par une condition) et qui lève une erreur si celle-ci est fausse.

Exemple 0.10.

```
def max(tbl:List[int]):
  assert(len(tbl)>0)                                    ①
  m = tbl[0]                                            ②
  for elm in tbl:
    if elm > m:
      m = elm
  return m
```

① Ici si le tableau est vide, on lève une exception
② On comprend pourquoi nous avons mis un `assert` : cette ligne aurais failli sinon avec un `tbl` vide

> **Avertissement**
>
> `assert` ne remplace pas une bonne gestion des erreurs. En effet dans un code en production le rattrapage d'erreurs sera compliqué, car le type d'erreur renvoyé est simplement `AssertionError`.

Cela permet dans un premier temps de garantir le bon fonctionnement du code, et ce, dès les premières étapes de conception.

On écrit en général des `assert` (et on lève des erreurs de manière générale) dans 2 grands cas :

Test des valeurs d'entrée

On vérifie que l'utilisateur a bien respecté la spécification du programme. Surtout lorsque ce dernier est implicite. Par exemple il est logique de ne pas donner un tableau vide dans une fonction `max` mais lorsque le programmeur oublie cela, il est utile de lui signaler.

Rappeler vous que la plupart du temps, on utilise des fonctions en cascade : on utilise le résultat d'une fonction en argument d'une autre (directement ou au travers de variable). Même lorsque l'on est consciencieux, il peut nous arriver que l'on oublie qu'une fonction peut retourner des valeurs que n'accepterait pas la suivante. Avec des `assert` on s'assure rapidement que cela n'est pas le cas et informe le plus rapidement possible le développeur !

On teste en général : les types, les valeurs limite, la contenue d'un conteneur, …

Test d'un invariant

Il arrive souvent dans un algorithme que l'on possède ce que l'on appelle un invariant, qui est pour faire simple une condition qui est toujours vraie lorsque l'on exécute l'algorithme et qui permet aux prochaines itération d'être vrai (pensez à la récurrence mathématique). Cela arrive fréquemment dans les algorithmes gloutons.

Par exemple, dans le cas de la dichotomie, on s'attend à ce que le tableau soit trié. Une des conséquences (facilement testable) et que chaque intervalle que l'on considère, la valeur du plus petit index est inférieurs ou égale au valeur de l'index du milieu qui est inférieur ou égal à la valeur d'indice supérieur. Ceci est un test simple (mais pas suffisant) permettant de savoir si notre résultat auras du sens. Si notre tableau n'est pas trié, ou que notre fonction est fausse, il arrivera un moment où nous serons peut-être capables de détecter cette erreur. Ce genre

d'invariant est appelé **invariant de boucle** car il est vrai à chaque itération d'un ensemble d'instruction.

Il peut aussi arriver que l'on structure nos données d'une façon particulière, de telle sorte a ce que l'on garde une propriété. Ceci s'appelle un **invariant de structure**. Nous en verrons plus tard, ne vous inquiétez pas. Pensez par exemple a un tableau que l'on veut garder trié. Sur ces données, nous pouvons appliquer des opérations (par exemple insérer un élément). Cela se révèle en général d'une très bonnes idées que chacune de ses fonctions teste que la structure nouvellement modifiée, garde vrai cet invariant. Pour cela nous utiliserons une exception pour nous indiquer que notre fonction à fauter, et donc arrêter le programme plutôt que risquer que cette structure soit par la suite mal utilisé !

Il est un peu dur de se représenter ces invariants car cela relève d'une définition générique. Mais retenez cela : ne vous cantonner pas à lever des exceptions seulement en début de programme, car lors de l'exécution, nous attendons un certains nombres de choses, et nous devons nous assurer que cela se déroule comme prévu, afin d'éviter les erreurs.

Des tests

Définition 0.1 (Testes)**.** Code permettant de vérifier le bon fonctionnement d'un programme en exécutant une ou toute partie de ce dernier, avec une entrée dont le comportement est connu et peut donc être vérifié.

Lorsque nous écrivons un programme il est souvent bon de tester ce dernier. Nous verrons ici différente méthode utile au test.

Tout d'abord il convient de donner un triste vérité : la validité d'un programme est indécidable. Cela veut dire qu'il est impossible de se doter d'un programme ou plus généralement d'une méthode permettant de vérifier a coups sûre du bon fonctionnement d'un programme. Bien sûr, il est possible dans certains cas de vérifier sa validité, mathématiquement (certain outil comme *Coq* peut nous aider), mais certains programmes (majoritairement complexe) sont impossibles à vérifier. Mais nous pouvons nous *"rassurer"* en vérifiant méthodiquement qu'aucun événement que nous avons détecté comme erreur (toute la science est de trouver ces événements) n'arrive jamais. Plus nous testons d'événement, plus nous serons confiants

Sans compter les erreurs que nous faisons en écrivant un programme cela fait donc 2 bonnes raisons de créer des tests, car ceci sont inévitables. D'autant que plus vite l'erreur est détectée, plus vite elle sera comprise et peut-être résolu !

> **i** Note
>
> Remarquer que la véracité de nos tests n'est pas abordé. Dans le monde réel on s'attelle à utiliser du code simple lors de tests afin d'être le moins source d'erreur. Car il se peut

> qu'un code pourtant juste soit détecté comme défectueux, car le teste en lui-même l'est...

Prototypes et implémentation

On distingue 2 type de tests :

Définition 0.2 (Tests en boite noire)**.** Les tests en boites noires considèrent les parties du programme sans connaissance de son implémentation

Définition 0.3 (Tests en boite blanche)**.** Les tests en boite blanche sont réalisés en connaissance de l'implémentation exacte ou partiel de la partie à tester

Les tests en boite noire sont utiles afin de réaliser une première évaluation, dans le cadre d'un développeur lambda. Ces derniers prennent en compte uniquement le prototype de la fonction dans le but de tester si le contrat est respecté. Il permet aussi de tester ce qui peut paraître évident à première vue mais qui a pu être oublié lors de l'implémentation.

Les tests en boite blanche permettent de prendre en compte certain spécificité du code afin d'imaginer où ce dernier pourrait s'arrêter de fonctionner correctement, et vérifie donc que des garde-fous évitent les situations dangereuses.

Ces deux types de tests sont complémentaires. Ils permettent en général la détection de problème différent.

Il est utile d'écrire les testes en boite noire par une personne extérieure ou préalablement à l'implémentation, afin d'éviter tout biais du a la connaissance de l'implémentation.

Intégration

Dans la Section nous avons vu l'interface avec laquelle nous créons nos tests. Nous allons désormais voir la manière dont nous pouvons utiliser nos tests.

Définition 0.4 (Test unitaire)**.** Test fait sur une partie précise et bien définie d'un programme. En général il s'agit d'une fonction ou d'une procédure.

Un test unitaire a pour but de tester une fonctionnalité précise d'un code. Cela a l'avantage d'être en général facile à implémenter : il suffit de lire la spécification du code (en général de la fonction) et d'appliquer des entrées (en général qui fonction, il est aussi possible de tester des erreurs), et d'observer si la sortie correspond.

C'est en général le testes que l'on fait intuitivement. Il est utile dans ce genre de teste de garder la portée du code tester la plus petite que possible.

Il est courant de mettre les tests unitaires dans le fichier dans lequel le code est définie, dans sa propre fonction (afin qu'il ne soit pas appeler en production). Le fait de mettre ces tests dans le même fichier permet au développeur de cette partie d'utiliser ces tests comme une documentation, et cela permet aussi une portabilité, où une fonction peut être réutilisé (avec son code de teste) dans un autre projet.

Exemple 0.1.

```
def ma_fonction(a:int, b:str)->float:
  """Ma super description !
  """
  pass


def test_ma_fonction():
  assert(ma_fonction(1, "a") == 1.0)
  assert(ma_fonction(2, "b") == 4.0)
  assert(ma_fonction(3, "c") != 1.0)
  #...
```

Définition 0.5 (Testes d'intégration)**.** Un test d'intégration est un test qui utilise tout ou une partie d'un code, ensemble dans une même exécution. Le test fait ainsi intervenir plusieurs unités du code.

Ce testes a pour but de tester la synergie des unités entre eux. Précédemment nous avons testé leur justesse, nous testons ici si l'on peut enchainer les unités entre elles.

Les tests d'intégration sont souvent utilisés dans deux cas :

- Lorsque la sortie d'une fonction est en général donnée à l'entrée d'une autre. On teste alors les fonctions impliquées dans une chaine "typique"
- Pour tester un programme complet.

Les tests d'intégration sont souvent très utiles afin de détecter facilement ce que l'on appelle des **régression** où l'ajout d'une nouvelle fonctionnalité, à travers une nouvelle fonction ou en modifiant une autre, provoque un effet en cascade, et finis par rendre tout ou une partie d'un faux programme, alors même que l'on a touché qu'à une faible partie de ce dernier. Cela arrive trop fréquemment en équipe.

En général les tests d'intégration sont réalisés dans un fichier à part, ou chaque ensemble de fonction à tester se voit attribuer sa propre fonction de teste. En plus d'être plus claire cela permet aussi d'utiliser chaque fonction avec des `import` et autres, et donc se placer dans le même cas qu'un utilisateur.

En général, les testes unitaires sont exécuté en premier puis viens-les testes d'intégration.

D'autres testes peuvent subvenir, comme la qualité du code ou le respect de certaines spécifications, mais nous n'aborderons pas ces derniers ici.

Tests *assisté par ordinateur*

Nous avons vu dans la section précédente comment écrire du code de teste. Cependant il est possible de laisser la machine nous assister dans certaines taches.

Fuzz Testing

Définition 0.6 (Tests à données aléatoires)**.** Les testes à données aléatoires ou *Fuzz Testing* est un test dans lequel les données d'entrée sont aléatoires.

Les tests par fuzzing sont très utilisés dans deux cas :

- Afin de tester qu'un programme crash ou autre lorsque les données d'entrée sont invalides. Cela permet de vérifier qu'un programme gère de la manière attendue les cas non-prévu.
- Lorsqu'il existe une méthode simple pour vérifier qu'un programme se comporte comme prévu mais qu'il est plus difficile de générer un ensemble de donnée suffisamment différent. Cela est le cas avec le tri : une fonction `ascendant` est très simple comparé à une fonction de tri, il est donc possible de tester des cas aléatoires, et il existe de nombreux cas limite dans certain algorithme de tri.

> ❗ Important
>
> Le fuzz testing n'est pas censé remplacer des tests ordinaire, écrit à la main. Au contraire. Mais il peut servir à étoffer le corpus de tests lorsqu'il est compliqué de trouver de nouveau cas

Ainsi un fuzzer est composé de 2 parties : un générateur de données qui suit une loi, et une fonction permettant de tester le comportement.

Un bon fuzzer gardera en mémoire les cas ayant posé un problème précédemment, afin de tester plus tard dans le corpus de tests, afin de ne pas oublier ces cas "plus compliqués" et éviter les régressions.

Testes de couverture

Définition 0.7 (Testes de couverture)**.** Le teste de couverture (ou *coverage testing*) est une pratique permettant de s'assurer que les testes couvre l'intégralité des instructions du code sources.

D'une manière générale cela permet de s'assurer que l'ensemble de tests et au moins suffisant pour avoir chaque cas traité. Il permet de s'assure que l'on a bien traité les différents cas d'un `if/else`, etc.

S'il existe encore une portion de code non exploré, cela traduit soit d'un jeu de tests insuffisant, soit d'un cas qui n'existe pas (ou a déjà été traité).

Le coverage testing s'inscrit donc en complément des tests précédent. Il existe aussi des méthodes mêlant fuzzing et couverture, afin de générer des cas de façon "quasi" automatique.

Tests de performance

Définition 0.8 (Testes de performance temporelle)**.** Testes permettant de mesurer le temps d'exécution d'une partie ou de l'ensemble du code.

Il peut être utile de tester les performances (notamment temporel) des applications. Cela permet en général le choix des implémentations à utiliser ou d'améliorer l'expérience utilisateur. De manière générale cela n'as pas d'impact sur la justesse du résultat.

Il existe en revanche certain cas, comme des programmes embarqués, ou de la gestion de serveur, où une exécution trop lente peut impacter la justesse des résultats. Ce genre de cas est en général très rare, mais il est facile d'oublier que ce genre de cas peut arriver.

De plus, la taille de l'entrée peut grandement influencer le temps d'exécution. C'est ce que mesure la complexité temporelle. Il peut être dans certain cas utile de connaitre cette complexité de manière empirique, afin de placer des garde-fous, évitant à un programme de devenir trop gourmand, au risque que le système d'exploitation l'arrête (Linux notamment à des solutions, radical, pour parler peu).

En Python deux moyens existe :

- Utiliser le module `time` ou la fonction `timeit`.

Exemple 0.2.

```
from time import time

t = time()
ma_fonction()
temps_execution = time() - t
```

Exemple 0.3.

```
from timeit import timeit

timeit("ma_fonction()", number=100)                                    (1)
```

1. Ici le code à exécuter est donné en tant que `str`. De plus `number` représente le nombre de fois que le code va être exécuté, afin d'obtenir des temps plus précis.

- Utilisation d'un profileur, qui calcule lors de l'exécution d'un code, le temps pris par chaque fonction. Cela permet entre autre de cibler plus précisément où l'optimisation doit se faire.

Comment écrire des tests

Des modules à la rescousse !

Il n'existe pas de bonne méthode mais je vais essayer ici d'explorer quelques pistes.

Il est souvent important d'écrire des tests unitaires pour chaque (ou quasi chaque) fonction, et d'écrire des teste d'intégration utilisant a minima tous les cas possible du programme principal.

En général on utilise des `assert` afin de tester son code. Si un test ne passe pas, alors le souci nous est de suite envoyer sous forme d'une erreur

Des librairies existent aussi afin de simplifier la création de tests.

2 librairies intégrées sont utilisables même en TP :

- `unittest` permettant l'écriture de test unitaire. Ceux module simple à utiliser nécessite l'utilisation de programmation objet (simple mais laisser pour plus tard). La documentation est ici
- `doctest` qui permet d'utiliser la documentation comme code de teste. Ce dernier regardera l'ensemble des fonctions à la recherche de ce qui ressemble des exemples dans la documentation de la fonction, exécuter ce code, et regarder si le résultat attendu est présent. Ce module a l'avantage de profiter d'une bonne documentation afin de générer automatiquement du code

Exemple 0.4. Exemple tirer de la documentation de `doctest`.

```
def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0

    Factorials of floats are OK, but the float must be an exact integer:
    >>> factorial(30.1)
    Traceback (most recent call last):
        ...
    ValueError: n must be exact integer
    >>> factorial(30.0)
    265252859812191058636308480000000

    It must also not be ridiculously large:
    >>> factorial(1e100)
    Traceback (most recent call last):
        ...
    OverflowError: n too large
    """
```

On appelle `doctest` ainsi (il est aussi possible de l'appeler au travers de code) :

```
python -m doctest -v example.py
```

Enfin d'autre module externe existe. On peut citer `pytest` par exemple. En général ces librairies apporte plus de fonctionnalités et une écriture simplifiée des teste, en utilisant directement des `assert` et en détectant les fonctions commençant par `test` comme des programmes de tests.

L'avantage de toutes ces librairies est qu'elles permettent l'exécution de code de teste et de rapport permettant en un clin d'œil de savoir les soucis rencontrer ou non.

Une philosophie

Un certain nombre de philosophies de développement se base en partie sur les tests ou expliquent comment ces derniers doivent être utilisé. De manière générale la philosophie dit *Agile* utilise les testes en son cœur.

Je citerai comme principe intéressant :

- Le *Test-driven-developpement* mettant le teste au centre du développement. On écrit les testes avant d'écrire une fonctionnalité. Chaque fonctionnalité doit répondre à un (plus rarement plusieurs) testes. Cela pousse à l'extrême ce concept : il ne peut pas y avoir de nouvelles fonctionnalités tant qu'un teste à montrer un souci, ce qui correspond à un besoin non répondu. Ainsi on écrit le cahier des charges à travers des objectifs à valider, qui se retranscrit par des tests.
- Le *pair programming* ou "développement par binôme" qui consiste à développer à deux. Ce concept va au-delà des tests (en utilisant un codeur et un superviseur), mais l'idée générale dans le cadre du cours est la suivante : chaque fonctionnalité est développée par 2 personnes. Une écrit le code, l'autre le testes. Les deux personnes n'ont pas accès à la production de l'autre. Au bout d'un temps donner (1 jours) les testes sont exécutés et les rôles sont inversés. Bien évidement le développeur fait aussi ses propres tests et le "testeur" peut aussi aider au cahier des charges. Cela permet d'écrire facilement du code en boite noire/boite blanche et d'avoir un regard neuf qui écrit du code afin d'éviter les situations de "nez dans le guidon" (manque de recul).
- Le *rubber ducking* ou méthode du canard en plastique consiste à prendre du recul en explicitant ce que l'on compte faire et la manière dont l'on compte le faire. Le canard en plastique aide dans ce processus : on explique ses pensées de manière simple et concise comme si l'on devait l'expliquer a quelqu'un d'extérieur, et ce de manière simple. Ce canard peut être au besoin remplacer par un être de chaire bien sûr ! Le but est que se forcer à expliquer, force son cerveau à clarifier ses propres penser, puisque cette étape est nécessaire à sa retranscription. Le rubber ducking est une méthode générale de debugging mais ce dernier peut être utilisé afin de trouver les possibles failles et donc trouver les testes à réaliser !