On découvre

- les notions de type et valeur en informatique
- le type `int` utilisé pour représenter les nombres entiers

1 Types de valeurs

L'informatique est d'un certain point de la vue la science de la manipulation d'informations.

Les informations sont représentées par des **valeurs**.

En Python, toutes les valeurs possèdent un **type**.

Le type d'une valeur définit :

- l'ensemble des valeurs qui peuvent être prises ;
- les **opérations** qu'il est possible de lui appliquer.

Les types de base sont :

- les entiers, appelés `int`
- les nombres à virgules flottantes (approximation des réels), appelés `float`
- les booléens, appelés `bool`
- les chaînes de caractères, appelées `str`.

Dans un premier temps nous présenterons les types numériques `int` et `float`, qui représentent des nombres, en commençant par le type `int`.

2 Le type `int`

Le type *entier* ou `int` de Python permet de représenter n'importe quel nombre relatif. L'ensemble des valeurs de type `int` sont donc tous les nombres relatifs, y compris les très grands :

```
>>> 12345432123456412345665431234565431234565234 56
12345432123456412345665431234565431234565234 56
```

2.1 Valeurs littérales de type `int`

En informatique, les **valeurs littérales**, ou **littéraux**, sont les manières d'écrire des valeurs.

Les littéraux entiers s'écrivent naturellement avec les dix chiffres `0`, `1`, ..., `9` (et donc en base 10).
Par exemple le nombre 42 s'écrira avec la suite de caractères `42`.

```
>>> 42
42
```

Le caractère `-` est utilisé comme préfixe pour les nombres négatifs. Par exemple :

```
>>> -14
-14
```

2.2 Syntaxe des littéraux

La **syntaxe** d'un langage de programmation désigne les règles d'écriture à respecter, un peu comme l'orthographe et la grammaire des langues comme le français.

La syntaxe Python des littéraux de type `int` impose quelques règles à respecter :

1. Il ne faut pas utiliser les séparateurs de milliers qu'on utiliserait "à la main (ni espace, ni point) : le nombre 10000000 s'écrit en Python `10000000` et non ~~`10 000 000`~~ ou ~~`10.000.000`~~. Par contre, on peut utiliser le séparateur `_` et écrire `10_000_000`
2. Il ne faut pas faire précéder le premier chiffre d'un 0, sauf pour le nombre 0. On doit écrire `7` et non ~~`007`~~.

3. D'autres règles existent[1]. Elles ne sont pas indispensables pour pouvoir écrire en Python n'importe quel nombre entier.

D'autres écritures (en binaire, octal ou hexadécimal) existent, cf cours de Codage.

2.3 Opérations

Le type `int` possède un ensemble d'opérations applicables aux valeurs de ce type.

2.3.1 Opérations d'addition, soustraction, multiplication, élévation à la puissance

On trouve naturellement les opérations arithmétiques habituelles:

- l'addition `3 + 14`,
- la soustraction `3 - 14`,
- la multiplication `3 * 14`, et
- l'élévation à la puissance `3 ** 14`.

Ces opérations produisent une valeur du type `int`.

```
>>> 3 + 14
17
>>> 3 - 14
-11
>>> 3 * 14
42
>>> 2**3
8
```

Il est possible de mettre aucune, une ou plusieurs espaces entre un opérande et l'opérateur : on peut écrire `1 + 3` ou `1+3`. Utilisez ce qui vous semble le plus lisible.

Notons la syntaxe de la multiplication : l'opérateur de multiplication est le `*` et non le $\times$ ou le point, ce qui diffère des notations mathématiques que vous avez l'habitude d'utiliser. La syntaxe de l'élévation à la puissance bouscule aussi vos habitudes d'écriture.

2.3.2 Opérations liées à la division euclidienne

La division euclidienne, que vous avez vue en cours de maths, est également disponible. Les opérateurs associés sont moins intuitifs pour les débutants.

Prenons par exemple la division euclidienne de 51 par 10. On note $51 = 10 \times 5 + 1$ où 5 est appelé le quotient et 1 le reste.
Ce reste est également nommé **modulo**.
Le schéma ci-dessous illustre cette division posée comme cela fait pour la résoudre à la main.

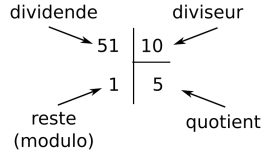

Figure 1: Exemple d'une division euclidienne posée

- le quotient est obtenu par `a // b` (notez bien l'opérateur `//` avec deux barres obliques)
- le reste ou modulo est obtenu par `a % b`

[1] Voir la syntaxe des littéraux de type entier dans la documentation officielle de Python : `https://docs.python.org/3/reference/lexical_analysis.html#integer-literals` (en anglais).

```
>>> 51 // 10
5
>>> 51 % 10
1
```

3 Représentation du type en Python

Le langage Python permet de trouver le type d'une valeur en utilisant la fonction `type` (nous verrons les fonctions un peu plus loin).

```
>>> type(3)
<class 'int'>
>>> type(51 // 10)
<class 'int'>
>>> type(51 % 10)
<class 'int'>
```

Pour le moment nous n'expliquerons pas le mot-clé `class`, vous le verrez au second semestre.

4 Erreurs

4.1 Erreurs de syntaxe

Que se passe-t-il si on ne respecte pas la syntaxe de Python ? Il se produit une erreur de syntaxe.

Par exemple :

```
>>> 3 +
  File "<stdin>", line 1
    3 +
      ^
SyntaxError: invalid syntax
```

L'erreur a un nom (`SyntaxError`) et vient avec un **message d'erreur** plus ou moins détaillé, ici `invalid syntax`.

Dans l'exemple suivant le message d'erreur est plus détaillé :

```
>>> 007
  File "<stdin>", line 1
    007
    ^^
SyntaxError: leading zeros in decimal integer literals are not permitted; use an 0o prefix for octal integ
```

Quand une erreur de syntaxe se produit, le programmeur ou la programmeuse doit lire attentivement le message d'erreur pour comprendre quelle portion du code pose problème et la corriger.

La connaissance de la syntaxe vient avec une pratique régulière de la programmation Python, au fil des interactions avec l'interpréteur Python et l'environnement de programmation.

4.2 Erreurs à l'exécution

Parfois la syntaxe est correcte, mais une autre erreur se produit, cette fois car la logique de Python n'est pas respectée.

Par exemple il est impossible d'effectuer une division euclidienne dont le diviseur est `0`:

```
>>> 100 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```


Dans ce cas le nom de l'erreur est `ZeroDivisionError`. Ce n'est pas une erreur de syntaxe : `100//0` est syntaxiquement correct.

Lire des messages d'erreur et **savoir agir pour réparer le code** fait partie du quotidien des programmeur · euses ! C'est une compétence à acquérir.

5 Memento

- un type définit l'ensemble des valeurs valides pour ce type
- le type d'une valeur définit l'ensemble des opérations possibles sur ces valeurs
- une valeur littérale est une manière d'écrire des valeurs
- les valeurs du type `int` représentent les nombres entiers
- on écrit les entiers (en base dix) en respectant quelques règles simples
- les opérations arithmétiques d'addition `+`, soustraction `-`, multiplication `*` et puissance `**` sont disponibles sur les entiers
- le quotient et le reste de la division euclidienne s'obtiennent avec les opérateurs `//` et `%`
- en Python, on peut connaître le type d'une valeur avec `type(…)`
- Le code écrit doit respecter les règles de la syntaxe de Python
- L'exécution d'un code Python peut provoquer une erreur, de syntaxe ou non.

On découvre

- les *flottants* et le type `float` utilisé pour représenter les nombres réels
- les opérations sur les *flottants*
- comment combiner nombres entiers et nombres flottants

1 Le type `float`

Le type *flottants* ou `float` de Python permet de représenter certains nombres réels. On dit aussi *nombre flottant* ou *nombre à virgule flottante*.

Nous pouvons considérer qu'un nombre flottant est la représentation d'une approximation d'un nombre réel, de la même façon que le décimal $3,1416$ est une approximation du réel π.

Notez bien que tous les nombres réels ne peuvent pas être représentés par une valeur de type `float`.

2 Littéraux flottants

Deux notations sont possibles pour les littéraux flottants :

- la notation simple,
- la notation scientifique.

2.1 Notation simple

Avec la notation simple, on écrit la partie entière et la partie décimale en les séparant par un point `.`. Le point est le séparateur décimal habituellement utilisé dans les systèmes anglo-saxons, alors que presque tous les autres systèmes utilisent la virgule.

Comme pour les nombres entiers, on peut préfixer un nombre à virgule flottante par un signe moins `-`.

```
>>> 3.1416
3.1416
>>> -1.25
-1.25
```

L'usage du séparateur décimal `.` est obligatoire afin de distinguer les flottants des entiers.

```
>>> 51.
51.0
>>> 51
51
>>> type(51.)
<class 'float'>
>>> type(51)
<class 'int'>
```

Il est possible d'omettre le zéro de la partie entière :

```
>>> .5
0.5
```

NB : si on utilise une virgule au lieu d'un point pour séparer la partie entière et la partie décimale, il ne se produit pas d'erreur de syntaxe (comme on pourrait s'y attendre). Mais les comportements ne sont pas non plus ceux attendus :

```
>>> 2,5
(2, 5)
>>> 2.5 + 2,5
(4.5, 5)
```

En effet, `2,5` est un objet de type `tuple` qu'on ne verra pas pour le moment :


```
>>> type((2,5))
<class 'tuple'>
```

2.2 Notation scientifique

La notation scientifique permet de représenter facilement des grands ou petits nombres.

Prenons l'exemple du *nombre d'Avogadro*, approximé à $6,02214076 \times 10^{23}$ mol^{-1}.
On appelle *mantisse* la partie à gauche du symbole $\times$, ici $6,02214076$. Il s'agit d'un nombre réel.
On appelle *exposant* la puissance de 10, ici 23. Il s'agit d'un nombre relatif.

Pour la notation scientifique, on écrit successivement :

- la mantisse en notation simple, ici `6.02214076`
- le caractère `e`
- l'exposant, ici `23`. On peut aussi écrire `+23`

Une notation simple du nombre d'Avogadro est `602214076000000000000000.0`, longue et peu lisible. On écrira donc plutôt :

```
>>> 6.02214076e+23
6.02214076e+23
>>> 6.02214076e23
6.02214076e+23
>>> 602214076000000000000000.0
6.02214076e+23
```

3 Opérations flottantes

Les opérations arithmétiques habituelles sont disponibles sur les valeurs du type `float` :

- l'addition `x + y`, la soustraction `x - y`, la multiplication `x * y`, et l'élévation à la puissance `x ** y`.

Ces opérations produisent une valeur du type `float`.

```
>>> 3.0 + 0.14
3.14
>>> 3.0 - 1.4
1.6
>>> 3.0 * 14.
42.0
>>> 9.0 ** 0.5
3.0
```

La division flottante est également disponible, notée de la simple barre oblique : `x / y`.
Elle produit une valeur flottante.

```
>>> 5.0 / 2.0
2.5
>>> 4.0 / 2.0
2.0
>>> 1.1 / 0.1
11.0
```

3.1 Approximation des nombres réels

Les flottants sont des approximations, certains résultats peuvent paraître surprenants.

```
>>> 0.1 + 0.2
0.30000000000000004
```

Le résultat du calcul précédent est précis à 10^{-17} près.

La commutativité des opérateurs n'est plus assurée :

```
>>> 0.1 + 0.5 - 0.45
0.14999999999999997
>>> 0.1 - 0.45 + 0.5
0.15000000000000002
```

Au contraire, les calculs du type `int` sont exacts.

Pour en savoir plus, vous pouvez consulter la documentation Python

4 Opérations entre entiers et flottants

L'ensemble des opérations arithmétiques s'appliquent à des nombres entiers et flottants.

Pour toutes les opérations sauf la division flottante, le type du résultat :

- est `int` si les deux opérandes sont des `int`
- est `float` si les deux opérandes sont des `float`
- est `float` si une opérande est un `int` et l'autre un `float`

```
>>> 3 * 14
42
>>> 3.0 * 14.0
42.0
>>> 3.0 * 14
42.0
>>> 3 * 14.0
42.0
```

La division flottante `/` produit toujours une valeur de type `float`, même si les deux opérandes sont de type `int` :

```
>>> 5 // 2
2
>>> 5 / 2
2.5
>>> 4 // 2
2
>>> 4 / 2
2.0
```

5 Memento

- les nombres flottants sont des représentations ou approximations des nombres réels
- le type `float` est utilisé pour représenter les nombres flottants
- la notation simple des flottants de la forme `123.45` utilise le point comme séparateur décimal
- la notation scientifique des flottants de la forme `12.34e+56` distingue mantisse et exposant
- les opérations arithmétiques d'addition `+`, soustraction `-`, produit `*`, division euclidienne `//` et `%`, et puissance `**` sont disponibles sur les flottants et les entiers
- la division flottante notée `/` produit toujours une valeur flottante

On découvre :

- les notions de type et valeur en informatique
- les chaînes de caractères

1 Chaîne de caractères

Un **caractère** est un symbole qui peut être écrit :

- une lettre (de l'alphabet latin) : `a`, `b`, etc., `X`, `Y`, `Z`
- une lettre accentuée : `à`, `û`, `É`, etc.
- un chiffre : `0`, `1`, etc.
- un symbole de ponctuation tel le point `.`, le point-virgule `;`, etc.
- une espace
- une lettre d'autres alphabets (non-latin) : α, Φ, $\breve{\cup}$, …

Une **chaîne de caractères** est une séquence de caractères, c'est–à–dire des caractères qui se suivent les uns derrières les autres.

En anglais, on utilise le terme *string*.

Le nombre de caractères d'une chaîne de caractères n'est pas contraint.

Une chaîne de caractères peut ne contenir aucun caractère : on l'appelle alors **chaîne vide**.

2 Valeur littérale d'une chaîne de caractères

En informatique, les **valeurs littérales** sont les manières d'écrire des valeurs.

Pour les chaînes de caractères, on écrit tout simplement la séquence des caractères entre deux délimiteurs.
On utilise habituellement l'apostrophe `'` comme délimiteur.

On écrit donc par exemple :

```
>>> 'Elle programme en Python'
'Elle programme en Python'
```

ou

```
>>> '314'
'314'
```

(ce littéral est bien de type chaîne, à ne pas confondre avec l'entier `314`).

La chaîne vide, qui ne contient aucun caractère, peut s'écrire :

```
>>> ''
''
```

On peut aussi utiliser les guillemets `"`comme délimiteurs, en particulier quand la chaîne de caractères comporte une apostrophe :

```
>>> "Elle s'amuse avec Python"
"Elle s'amuse avec Python"
```

3 Le type `str`

On peut interroger l'interpréteur Python sur le type d'une valeur :

```
>>> type('Elle programme en Python')
<class 'str'>
>>> type("Elle s'amuse avec Python")
<class 'str'>
```

Autre exemple avec la chaîne vide :

```
>>> type('')
<class 'str'>
```

ou pour une chaîne constituée exclusivement de chiffres :

```
>>> type('314')
<class 'str'>
```

4 Opérations sur les chaînes de caractères

Nous n'utiliserons que quelques opérations sur les chaînes de caractères dans l'UE. Nous en reprogrammerons certaines. L'ensemble peut être découvert par `dir(str)`.

4.1 Opération de concaténation

La **concaténation** de deux chaînes de caractères :

- produit une chaîne de caractères
- dont la séquence de caractères est la mise bout à bout des deux chaînes de caractères.

Cette opération est notée `+`.

On peut donc écrire :

```
>>> 'Elle ' + 'programme en Python'
'Elle programme en Python'
>>> 'Elle ' + "s'amuse avec Python"
"Elle s'amuse avec Python"
```

ou encore :

```
>>> '31' + '4'
'314'
```

NB : on ne confondra pas l'opérateur `+` des types numériques effectuant une addition et l'opérateur `+` du type chaîne effectuant une concaténation. Même si lexicalement (syntaxiquement) le caractère `+` est le même dans les 2 cas.

4.2 Répétition de chaînes de caractères

L'opérateur `*` appliqué à une valeur entière n et une chaîne de caractères s :

- produit une chaîne de caractères
- résultat de n concaténations de la chaîne s.

```
>>> 3 * 'bravo ! '
'bravo ! bravo ! bravo ! '
>>> 'bravo !' * 3
'bravo !bravo !bravo !'
>>> 'bravo !' * 0
''
```

NB: de même on ne confondra pas la multiplication des types numériques avec la répétition du type chaîne.

4.3 Conversions

Des fonctions prédéfinies du langage Python permettent de changer le type d'une valeur, lorsque c'est possible. Ce changement s'appelle une **conversion de type**.

La fonction prédéfinie `str()` produit une représentation sous forme d'une chaîne de caractères de la valeur passée en paramètre, par exemple d'une valeur numérique.


```
>>> str(12)
'12'
>>> str(42.14)
'42.14'
```

Réciproquement, la fonction prédéfinie `int()` produit un nombre entier à partir d'une chaîne de caractères, si et seulement si cette chaîne ne contient que des chiffres, éventuellement précédés du signe `-`. Si un autre caractère est présent, à part les espaces qui sont tolérés, une erreur est affichée.

```
>>> int('12')
12
>>> int('007')
7
>>> int('-12 ')
-12
>>> int('12.')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.'
>>> int('23.5')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '23.5'
>>> int('24°C')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '24°C'
```

Enfin, la fonction prédéfinie `float()` produit un nombre flottant à partir d'une chaîne de caractères, si et seulement si cette chaîne ne contient que des chiffres, éventuellement un `.` et éventuellement précédés du signe `-`. Si un autre caractère est présent, à part les espaces qui sont tolérés, une erreur est affichée.

```
>>> float('12')
12.0
>>> float('007')
7.0
>>> float('-12 ')
-12.0
>>> float('12.')
12.0
>>> float('23.5')
23.5
>>> float('24°C')
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ValueError: could not convert string to float: '24°C'
```

5 Memento

- une chaîne de caractères est une séquence de caractères
- on écrit les caractères d'une chaîne entre délimiteurs (apostrophes ou guillemets)
- l'opération de concaténation de deux chaînes de caractères est notée `+`
- l'opération de répétition d'une chaîne de caractères est notée `*`
- on peut convertir un type numérique en chaîne et caractères et, dans certains cas, des chaînes de caractères en type numérique.

On découvre comment combiner plusieurs opérations pour former des expressions.

1 Expressions

Il est possible de combiner plusieurs opérateurs pour former une *expression*.
On écrit par exemple `3 + 4 - 2`, ou `----2`.

En programmation, une expression est la notation d'un calcul à réaliser. Une expression peut-être une valeur littérale, ou la combinaison par un opérateur d'une ou plusieurs expressions. Par ex `2` est une expression, et `1 + 2*3` est une expression qui représente l'addition des expressions `1` et `2*3`.

Lors de l'exécution, l'expression sera toujours réduite à une valeur.
On dit qu'elle est **évaluée**. Par exemple, l'**évaluation** de `----2` produit la valeur `2`.

Quand on entre une expression près l'invite de l'interpréteur Python et qu'on presse la touche `Entrée`, Python évalue cette expression puis affiche la valeur obtenue:

```
>>> ----2
2
```

1.1 Priorité des opérateurs

Cette évaluation se fait en suivant un ordre de priorité qui définit l'ordre des opérations à effectuer.

Par exemple, l'évaluation de `3 + 4 - 2`

- calculera la somme de `3 + 4`
- puis fera la différence entre cette valeur et `2`

Le tableau ci dessous classe les opérateurs par ordre de priorité décroissant. Les opérateurs sur la même ligne ont la même priorité.

	`**`	`-` unaire (opposé)	`+` unaire
`*`	`/`	`//`	`%`
		`+` (addition)	`-` (soustraction)

On peut aussi décrire les niveaux de priorité ainsi, du moins prioritaire au plus prioritaire :

- `+` (addition), `-` (soustraction)
- `*`, `/`, `//`, `%`
- `**`, `-` unaire (opposé), `+` unaire

On aura par exemple :

```
>>> 2 * 3 + 5 * 1
11
>>> 2 ** 2 * 2
8
>>> - 2 - + 3
-5
```

Lorsque deux opérateurs ont la même priorité, ils sont évalués de gauche à droite sauf les `**` qui sont évalués de droite à gauche.

```
>>> 2 - 3 - 1
-2
>>>  5 // 2 * 2
4
>>> 2 ** 2 ** 3
256
```

Il est possible de contrôler l'ordre d'évaluation des opérations en utilisant des parenthèses `(` et `)` :


```
>>> 2 * (3 + 5) * 1
16
>>> 2 - (3 - 1)
0
>>> (2 ** 2) ** 3
64
```

Les priorités sont intuitives quand on les a déjà rencontrées en cours de maths. On verra quand on découvrira les opérateurs du type booléen qu'elles ne sont pas toujours faciles à retenir quand le nombre d'opérateurs augmente. D'une manière générale il vaut mieux utiliser les parenthèses pour lever toute ambiguité.

Remarquons que les expressions peuvent être formées d'opérateurs de toute sorte :

```
>>> ((1 + 1) * ('bravo,' + ' ')) + 'bravo !'
'bravo, bravo, bravo !'
```

1.2 Types et erreur

Chaque type vient avec un ensemble d'opérateurs.

Nous avons vu que certains opérateurs appartiennent à des types différents même si syntaxiquement ils sont représentés par le même caractère. Pour les opérateurs `*` et `+` les opérandes[1] soivent respecter les types suivants:

- `<type_numerique> + <type_numerique>` : addition
- `<type_numerique> * <type_numerique>` : multiplication
- `<type_str> * <type_int>` et `<type_int> * <type_str>` : répétition
- `<type_str> + <type_str>` : concaténation

Que se passe-t-il si on utilise des valeurs dont le type ne respecte pas les types prévus ? Dans ce cas une erreur de type de nom `TypeError` se produit :

```
>>> '3' * 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

L'expression est néanmoins syntaxiquement correcte.

2 Memento

- une expression combine une ou plusieurs opérations
- l'évaluation d'une expression la réduit à une valeur
- des règles de priorités fixent l'ordre d'application des opérations
- les parenthèses permettent de préciser l'ordre d'application des opérations

[1] Les opérandes sont les valeurs sur lesquelles s'appliquent les opérateurs

On découvre comment écrire des fonctions simples. On verra dans la suite du cours comment écrire des fonctions plus complexes.

Commentaires Python Ce support de cours utilise les commentaires de Python.

Un commentaire commence par le caractère dièse `#` et est suivi par ce qu'on veut. Tout ce qui est à droite d'un `#` est ignoré par Python. On peut donc y écrire des notes qui servent à expliciter des choses relatives au code.

```
>>> 1.1 + 2.2 # rappel : les calculs sur flottants sont approchés
3.3000000000000003
```

1 Pourquoi écrire des fonctions ?

Parce qu'on souhaite faire facilement des calculs sur des données !

Imaginons le problème suivant (qui n'est pas follement passionnant, mais on commence par des choses simples !). Dans le cadre de votre mission de bénévolat dans une association, vous avez à disposition des cartons de tailles diverses, et des objets à ranger dans les cartons. Étant donnés n objets et des cartons à m places, combien de cartons complets pourrez-vous remplir ? Vous devez écrire une fonction qui répond au problème.

Cette description informelle du travail devant être réalisé par une fonction est appelée **spécification**. Votre tâche de base durant le semestre sera d'écrire des fonctions qui répondent à leur spécification (souvent l'énoncé du TDM ou du TD).

Pour écrire un code Python qui répond au problème ci-dessus, les étudiant · es proposent souvent spontanément 2 solutions, par exemple pour `113` objets et des cartons à `8` places:

- utiliser la division approchée sur flottants de `113` par `8` puis se débrouiller pour garder la partie entière du résultat pour obtenir un nombre entier ;
- ou calculer le quotient de la division entière de `113` par `8`.

Nous choisissons la 2ème solution qui répond de manière exacte au problème :

```
>>> 113 // 8 # pour des cartons de 8
14
>>> 235 // 8
29
>>> 113 // 5 # pour des cartons de 5
22
```

En programmation écrire une fonction permet d'**encapsuler** le calcul en réalisant une **abstraction** : l'appel à la fonction identifie les données nécessaires au calcul mais masque la manière dont le calcul est réalisé.

Nous allons décrire une fonction `nb_cartons` qui pourra être utilisée ainsi :

```
>>> nb_cartons(235, 8) # la manière dont le calcul est fait
29
>>> nb_cartons(113, 5) # ne nous intéresse pas ici
22
```

Cette fonction pourra exprimer le calcul en fonction des données dont il dépend de la manière suivante: étant donné un nombre d'objets `nb_objets` et des cartons à `places_par_cartons` places, calculer la valeur de `nb_objets // places_par_cartons`.

Les fonctions seront les briques de base de la construction des programmes pour la durée du semestre, il est impératif que vous sachiez écrire des fonctions dès les 2 premières semaines.

2 Éléments d'une fonction, syntaxe

Il y a plusieurs manières de coder le problème précédent dans une fonction qui répond à la spécification. On va ici utiliser la fonction `nb_cartons` suivante :


```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
    """Renvoie le nombre de cartons complets qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
    return nb_objets // places_par_carton
```

Cet exemple respecte la syntaxe d'une définition de fonction telle que partiquée dans l'UE:

```
def <nom_fonction>(<liste_parametres>) -> <type_retour>:
    <docstring_de_documentation>
    <corps_de_la_fonction>
```

Ce qui se lit :

- Une définition de fonction commence par le mot-clé Python `def`
- ensuite, après au moins une espace, on trouve le **nom** de la fonction, dans notre cas `nb_cartons`. On a utilisé des chevrons et écrit `<nom_fonction>` pour signifier : "à remplacer par un nom de fonction".
- ensuite, entre parenthèses (caractère `(` pour la parenthèse ouvrante et `)` pour la parenthèse fermante), on trouve la liste des **paramètres** de la fonction (voir plus bas).
- *etc.*

La **docstring** de la fonction contient sa documentation, c'est une chaîne de caractères balisée par des triples guillemets.

```
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
```

Enfin le **corps** de la fonction contient le code qui sera exécuté lors de son appel :

```
    return nb_objets // places_par_carton
```

3 Signature de la fonction : première ligne

Cette première ligne indique la **signature** de la fonction, c'est à dire :

- son **nom**
- le **nom**, le **type** et l'**ordre** de ses **paramètres**
- le **type de retour** de la fonction.

```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
```

Les types dans la signature (**annotations de type** en Python) sont optionnels en Python[1] (mais obligatoires dans d'autres langages de programmation). Expliciter les types est une bonne pratique obligatoire dans l'UE.

Syntaxiquement, la définition d'une fonction est introduite par le mot-clé `def`. Comme Python est sensible à la casse (c'est -à-dire différencie les majuscules et les minuscules), il faut utiliser `def` et non `Def` ou `DEF`.py}.

À la suite de `def` on trouve :

- le **nom** de la fonction : `nb_cartons` dans notre exemple ;
- puis, entre parenthèses, la liste de ses **paramètres** : `nb_objets` et `places_par_carton` de type `int` dans notre exemple. Cette liste est de la forme : `<nom_paramètre>:<type>, ..., <nom_paramètre>:<type>`. Ce qui se lit : nom de paramètre puis caractère `:` puis type du paramètre, et s'il y a d'autres paramètres on sépare par le caractère `,` avant de continuer.
- puis le type de retour de la fonction, c'est à dire le type de la valeur calculée, précédé des caractères `->` : type `int` dans notre exemple:
- en terminant la ligne par le caractère `:`.

3.1 Syntaxe des identificateurs

Notre fonction utilise les noms `nb_cartons`, `nb_objets` et `places_par_carton`. Les **identificateurs** sont les noms choisis par le programmeur ou la programmeuse. Ils doivent respecter des règles définies par le langage Python.

En Python,

1. un identificateur ne peut contenir que des lettres, des chiffres ou le *blanc souligné* `_`[2]
2. un identificateur ne peut pas commencer par un chiffre (et doit donc commencer par une lettre ou par `_`)
3. un identificateur ne peut pas être un des *mots-clés* du langage Python. Ces mots-clés réservés sont les suivants:

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

Par exemple `annee_2526` est un identificateur correct, mais pas `25_26` qui commence par un chiffre.

On évitera les caractères spéciaux et les lettres accentuées.

Les identificateurs sont sensibles à la casse, ainsi `ide` et `Ide` sont deux identificateurs différents.

On adoptera un nommage cohérent au long du semestre, par exemple `nb_cartons` et non `NbCartons`.

3.2 Choix des identificateurs

On prendra soin de choisir des identificateurs **porteurs de sens**. L'objectif est que le rôle des objets nommés par les identificateurs soient immédiatement compréhensibles par le lecteur.

On aurait pu utiliser la fonction suivante:

```
def f(n:int, m:int) -> int:
    [...]
    return n//m
```

Cette fonction réalise exactement le même calcul que la fonction précédente mais est illisible. Que représentent `n` et `m` ? Doit-on écrire `f(113, 8)` ou `f(8, 113)` ?

Il ne faut pas hésiter à utiliser des identificateurs un peu longs : les éditeurs de code ont des mécanismes de suggestion de complétion qui vous permettront de ne pas taper chaque caractère.

[1] Techniquement les annotations de type font partie de la documentation, Python n'analyse pas ces anotations, mais d'autres outils construits au dessus de Python, par ex Thonny, le font.

[2] Dit aussi "souligné du 8" étant donné sa position sur le clavier azerty.

4 Bloc de code, indentation

Après cette première ligne de signature on trouve sa documentation et le corps de la fonction. Dans cet exemple il s'agit des lignes :

```
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
    return nb_objets // places_par_carton
```

Pour savoir où commence et où finit la définition d'une fonction, on utilise un **bloc de code**, signalé par les espaces qui le décalent vers la droite par rapport au mot-clé `def`.

Dans de nombreux langages de programmation on marque le début et la fin d'un bloc de code par des balises, par exemple `begin ... end` ou `{ ... }`. Python procède autrement : c'est le nombre d'espaces entre le début de la ligne (la première colonne) et le premier caractère du code qui délimite les blocs. On dit que le code est **indenté**.

On dit qu'on **indente** une ligne quand on la décale vers la droite. On dit qu'on **désindente** quand on la décale vers la gauche. Les niveaux d'indentation égaux délimitent des blocs.

Dans l'exemple de `nb_cartons` on voit que la ligne qui suit la ligne de signature est indentée au même niveau que toutes celles qui suivent. La fin du corps de la fonction n'apparaît pas, mais si on souhaitait définir une fonction `nb_objets_sans_carton` à la suite on désindenterait sa ligne de définition par rapport à la dernière ligne de `nb_objets_sans_carton`:

```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    [...]
    """
    return nb_objets // places_par_carton
    # cette ligne appartient à la fonction nb_cartons
# cette ligne n'appartient pas à la fonction nb_cartons
def nb_objets_sans_carton([...]
    [...]
```

Il est possible d'utiliser le nombre d'espaces que l'on veut pour indenter un bloc (souvent on prend 4 espaces au 1er niveau d'imbrication, 8 au second, etc). Par contre, ce nombre doit toujours être le même au sein d'un programme.

On n'utilisera pas la tabulation pour l'indentation.

En cas d'indentation intempestive, même si c'est un décalage d'une seule espace, une erreur dite d'indentation se produit:

```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
```

```
    $$$ nb_cartons(0, 5)
    0
    """
   return nb_objets // places_par_carton
```

Ici on a par erreur indenté à 3 espaces la dernière ligne, ce qui déclenche l'erreur de nom `IndentationError` suivante :

```
Traceback (most recent call last):
  File [...], line [...]
    return nb_objets // places_par_carton
                                        ^
IndentationError: unindent does not match any outer indentation level
```

Comme vous maîtrisez parfaitement l'anglais technique, inutile de traduire le mesage d'erreur…

Dans la suite de ce document, il est plus facile d'expliquer d'abord le corps de la fonction, puis la documentation.

5 Corps de la fonction, `return` et appel de fonction

On a écrit plus haut qu'on souhaitait effectuer des **appels** à la fonction `nb_cartons` pour exécuter le **calcul** codé par le **corps** de cette fonction.

Par exemple, pour `11` objets et des cartons à `5` places, on souhaite obtenir `2` comme résultat du calcul.

5.1 Appel de fonction

L'appel s'écrira `nb_cartons(11, 5)`.

La syntaxe générale d'un appel de fonction est :

`<nom_fonction>(<liste_expressions>)`

où `<liste_expression>` contient dans l'ordre de la signature de la fonction les valeurs qu'on souhaite fournir aux paramètres de la fonction. Les valeurs peuvent être des littéraux ou des expressions avec opérateurs.

On peut ainsi écrire l'appel `nb_cartons(8+3, 5)`.

Mais comment s'effectue le calcul qui produit la valeur `2` dans les deux cas précédents ?

5.2 Exécution du corps de la fonction

L'expression qui suit le mot-clé `return` est `nb_objets // places_par_carton`. Cette expression de type entier utilise comme opérandes de la division entière les paramètres de la fonction.

L'appel `nb_cartons(8+3, 5)` déclenche l'**évaluation** de `nb_objets // places_par_carton`, comme suit :

- chaque valeur passée au paramètre est évaluée : par exemple `8+3` est évalué en `11` et `5` est évalué en `5`;
- les valeurs sont associées aux paramètres dans ce qu'on appelle la **mémoire**, qui peut être représentée en première approche par le tableau suivant :

`nb_objets`	`11`
`places_par_carton`	`5`

- l'expression qui suit le mot-clé `return` (dans notre exemple `nb_objets // places_par_carton`) est évaluée en remplaçant chaque occurrence du paramètre[3] par la valeur qui lui est associée dans la mémoire. Dans notre exemple on évalue donc l'expression `11//5`, qui vaut `2`;
- l'exécution d'un `return` termine l'appel[4]: l'expression `nb_cartons(8+3)` vaut `2`.

[3] "Occurrence" de paramètre est un mot compliqué qui désigne simplement une **apparition** du paramètre. Par exemple dans `return nb_objets // places_par_carton` il y a une occurrence de `nb_objets` et une occurrence de `places_par_carton`. Dans `return nb_objets * nb_objets` il y a deux occurrences de `nb_objets`.

[4] On n'utilise pas le mot-clé `return` en dehors des fonctions.


```
>>> nb_cartons(11, 5)
2
```

L'expression associée au mot-clé `return` peut bien sûr utiliser des littéraux. Par exemple, si on arrive à caser 2 objets par place dans un carton, on écrira :

```
return (nb_objets * 2) // places_par_carton
```

Si on oublie le mot-clé `return`, ou si on commente la ligne contenant le `return` alors il n'y aura pas d'erreur (ni de syntaxe, si à l'exécution), mais la fonction ne renvoie aucune valeur :

```
>>> nb_cartons(11, 5) # sans return
>>>
```

NB : La fonction calcule un résultat pour toutes valeurs de `nb_objets` et `nb_cartons` qui ont du sens. Une erreur classique pour les débutants est de traiter uniquement les exemples donnés, par exemple avec un `return 11 // 5`. Dans ce cas la fonction se comporte bien pour les valeurs `11` et `5` et certaines autres valeurs mais n'a pas le comportement attendu pour l'immense majorité des autres valeurs. On écrit des fonctions pour traiter le cas général, pas juste des exemples.

6 Documentation : docstring

Juste sous la signature de `nb_cartons` on trouve une chaîne particulière balisée par des triples guillemets :

```
    """
    Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
```

Les triples guillemets permettent de définir des chaînes de caractères sur qui s'étendent sur plusieurs lignes:

```
>>> """Cette chaîne s'étend sur
plusieurs lignes"""
"Cette chaîne s'étend sur \nplusieurs lignes"
```

(Le caractère spécial `\n` représente un retour à la ligne).

Cette chaîne particulière qui suit immédiatement la signature de la fonction est appelée **docstring** et comme son nom l'indique ("chaîne de doc") contient la **documentation** de la fonction. Nous pouvons la consulter avec la fonction `help`[5] :

```
>>> help(nb_cartons)
Help on function nb_cartons:

nb_cartons(nb_objets: int, places_par_carton: int) -> int
    Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
```

[5] Même si les programmeur · euses ont tendance à aller chercher de l'information en utilisant un moteur de recherche sur Internet, la documentation textuelle de Python accessible via `help` directement dans l'outil de programmation est très pratique !

```
$$$ nb_cartons(10, 5)
2
$$$ nb_cartons(11, 5)
2
$$$ nb_cartons(0, 5)
0
```

La documentation n'a rien à voir avec les commentaires qui permettent d'écrire du texte libre à l'intérieur du code.

La documentation permet à fournir aux programmeur · euses qui voudront appeler la fonction dans leur propre code tout ce qui leur est utile de savoir **sans lire le code** (qui n'est pas toujours disponible, est parfois très long et complexe, et de toute manière on n'a pas envie de le lire).

Python n'interprète pas le contenu de cette chaîne, mais d'autres outils Python le font, et c'est très utile !

6.1 Description du calcul réalisé par la fonction

On commence classiquement une docstring par une courte phrase terminée par un point, décrivant le comportement de la fonction. C'est une convention Python.

Dans notre exemple :

```
Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
avec des cartons de `places_par_carton` places.
```

La suite de la docstring ne respecte par les conventions Python standard. L'équipe pédagogique a choisi d'écrire des docstrings adaptées à l'apprentissage de la programmation en mettant l'accent sur :

- la précondition de la fonction
- les exemples exécutables de la fonction, ou tests, détaillés dans le support qui suit.

6.2 Précondition de la fonction

"Pré" signifie "avant", "précondition" signifie donc : condition devant être vraie avant appel.

La **précondition** est l'éventuelle condition que les paramètres doivent respecter pour que l'appel à la fonction ait du sens. Dans le cadre de cette UE on exprimera toujours la précondition en fonction des **valeurs des paramètres**.

Dans l'exemple de `nb_cartons` il y a deux paramètres `nb_objets` et `places_par_carton`. Quelles sont les conditions sur les valeurs du nombre d'objets et du nombre de places par carton pour que calculer le nombre de cartons ait du sens ?

- comme ce sont des quantités d'objets, on s'attend à ce que les 2 valeurs soient positives;
- on peut admettre de ranger 0 objets (ce qui produira 0 cartons), par contre ça n'a aucun sens d'envisager des cartons à 0 places.

La documentation de la fonction indique donc :

```
Précondition : nb_objets >= 0 et places_par_cartons > 0
```

Appeler une fonction hors de sa précondition n'a aucun sens. Que se passe-t-il si on réalise tout de même un tel appel ?

- l'appel peut produire une erreur : `places_par_cartons(15, 0)` produira une erreur de nom `ZeroDivisionError`
- l'appel peut produire une valeur qui n'a aucun sens : l'appel `nb_cartons(-11, 2)` produit la valeur `-6`, qu'on se gardera bien d'utiliser où que ce soit (quel est le sens de `-6` cartons ?).

NB Dans l'UE il est obligatoire d'indiquer les préconditions des fonctions. S'il n'y en a pas (c'est à dire qu'appeler la fonction a du sens quelles que soient les valeurs des paramètres, ou si la fonction n'a pas de paramètre), on l'écrira ainsi :

```
Précondition : /
```

ou

```
Précondition : aucune
```


NB La précondition ne porte pas ~~sur le type~~ des paramètres. Ce type est indiqué dans les annotations de type de la signature. Elle ne porte pas non plus sur ~~la valeur renvoyée par la fonction~~ (ou son type) : c'est le rôle de la post-condition qu'on ne verra pas dans cette UE.

NB La précondition est un élément de documentation fondamental mais qui n'est pas utilisé par l'interpréteur Python. C'est un avertissement du programmeur ou de la programmeuse de la fonction sous la forme d'un contrat : *si tu m'appelles en respectant ma précondition, alors je me comporte comme indiqué dans la spécification* (sous-entendu : si la pré-condition n'est pas respectée il peut arriver n'importe quoi, rien n'est garanti car on est hors contrat).

6.3 Exemples exécutables d'utilisation

Voir le support qui suit !

7 Et après ?

On a vu la syntaxe et le principe d'exécution des fonctions.

Mais comment sait-on comment nommer une fonction, quels paramètres utiliser, quel code écrire ? Comment sait-on quelles fonctions définir ? C'est le plus difficile, bien plus difficile que de maîtriser la syntaxe des fonctions.

Décomposer un problème en sous-problèmes codés par des fonctions est une compétence centrale en programmation.

On travaillera cette compétence en amphi et elle viendra avec la pratique.

On découvre :

- comment compléter la spécification et la documentation des fonctions par des exemples et tests
- comment automatiser les tests de nos fonctions

On reprend l'exemple de `nb_cartons`:

```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
    return nb_objets // places_par_carton
```

1 Exemples d'utilisation et tests

La documentation telle que rédigée dans l'UE contient des exemples d'utilisation de la fonction. On reconnaît la syntaxe de l'interpréteur Python à l'exception de l'invite : on écrit

```
$$$ nb_cartons(10, 5)
2
```

et non

```
>>> nb_cartons(10, 5)
2
```

Ces exemples ont deux buts.

Le premier est d'enrichir la documentation : il est toujours précieux quand on cherche à comprendre le comportement d'une fonction d'en trouver une illustration associant un appel de fonction et son résultat attendu.

Le second est de **tester** automatiquement le comportement de la fonction en exécutant ces exemples et en comparant résultat obtenu et résultat atendu. On utilisera dans l'UE la **bibliothèque de test L1test**.

La docstring ci-dessus contient 3 tests matérialisés par les 3 invites `$$$`.

Considérons le premier test :

```
$$$ nb_cartons(10, 5)
2
```

Il peut se lire ainsi : "si j'exécute `nb_cartons(10, 5)`, le résultat attendu est 2".

2 Exécution des tests avec L1Test

Le principe est le suivant :

- L1Test exécute l'appel `nb_cartons(10, 5)`
- il compare la valeur obtenue (2) avec la valeur attendue (2 aussi) (ce mécanisme de comparaison est appelé **oracle**)
- il fournit un message (appelé **verdict**) indiquant l'issue du test. Dans ce cas, comme les valeurs obtenue et attendue sont égales, on dit que le test passe, ou que c'est un succès.

CC BY

Université de Lille
Test de fonction automatisé avec L1test
2025-2026
UE PROG, L1S1 MI

Tous les tests de la fonction sont exécutés.

La figure 1 montre comment L1test exécute les tests précédents et affiche les verdicts associés. Tout les tests sont au vert, couleur traditionnellement associée au succès dans le monde du test du logiciel.

L'écriture et l'exécution des tests sera détaillée dans le chapitre suivant.

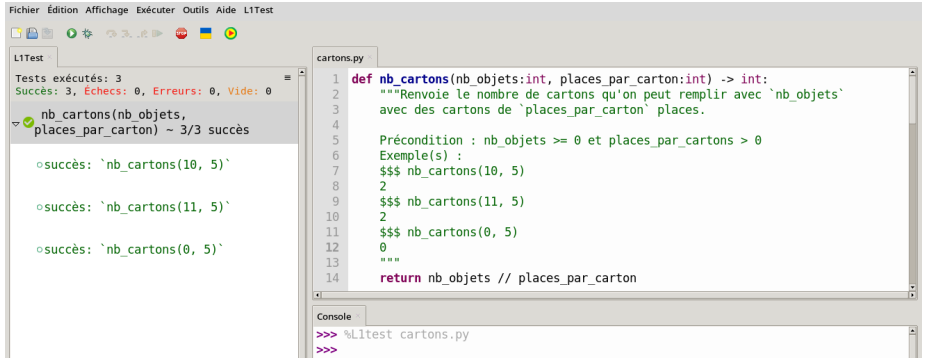

Figure 1: Exécution des tests avec L1test, avec un code correct

Supposons mainteant qu'on ait introduit une erreur dans le code ou dans les tests. La figure 2 montre cette fois des tests au rouge ! Signe qu'il faut retourner inspecter le code ou les tests, car quelque chose ne va pas !

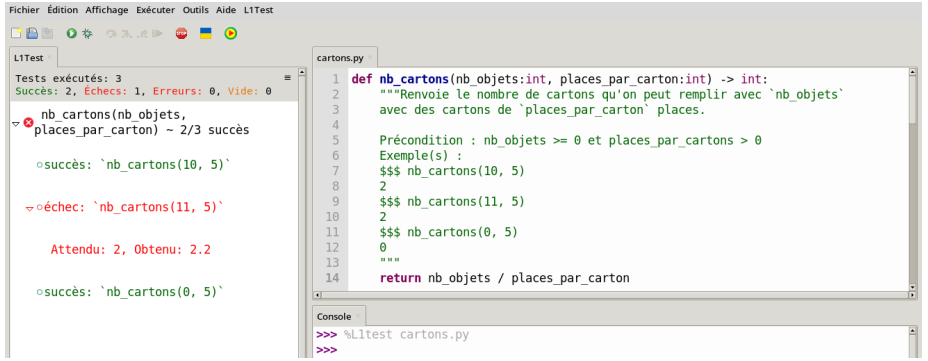

Figure 2: Exécution des tests avec L1test, avec un code ou des tests incorrects

NB : cliquer sur le bouton "Exécuter tous les tests" :

- exécute les tests
- rend disponibles dans la console les fonctions et autres objets définis dans l'éditeur.

3 Écrire des tests avant d'écrire le code

Notons que cette section `Exemples` de la documentation doit être écrite *avant* d'écrire le code de la fonction, en lisant sa spécification.

CC BY

Il ne s'agit pas d'écrire la fonction, d'exécuter des appels à la fonction dans l'interpréteur, et de recopier le résultat, car on risque alors de valider par des tests faux un code faux.

Écrire les tests avant d'écrire le code permet aussi de vérifier qu'on a bien compris la spécification / l'énoncé plutôt que de se lancer trop vite et d'écrire du code erratique.

4 Test automatisé

On parle de **test automatisé** car un simple clic sur le bouton situé à droite du drapeau ukrainien suffit à déclencher l'exécution des tests et l'affichage des verdicts. Quand on tape des appels de commandes dans l'interpréteur et qu'on compare dans sa tête la valeur obtenue et la valeur attendue, on fait au contraire au mieux du **test manuel** (si les tests sont écrits quelquepart), au pire de l'essai-erreur.

5 Memento

- la spécification des fonctions est complétée par des exemple d'utilisation, dans la section `Exemples`
- ces exemples, écrits avec la syntaxe de `L1test`, deviennent des tests automatisés
- exécuter un test consiste à exécuter l'expression testée et à la comparer avec le résultat attendu
- exécuter ces tests en 1 clic permet d'afficher, avec un code couleur vert/rouge, le verdict associé aux test

On découvre :

- comment écrire des tests unitaires pertinents
- comment `L1test` exécute les tests et produit un verdict

1 Écriture de tests unitaires avec L1Test

On commence par le cas général, puis on aborde le cas de calculs approchés.

1.1 Cas général

Un *L1test*, ou simplement *un test*, est écrit dans la docstring de la fonction testée[1]. Dans sa version la plus simple, il se compose de 2 lignes se suivant immédiatement :

```
$$$ <expr>
<expr_attendue>
```

NB : Syntaxiquement l'espace entre l'invite `$$$` et l'expression est obligatoire. Son oubli déclenche l'erreur suivante :

```
A space is missing after the prompt : $$$nb_cartons(10, 5)
```

On dit que le test est *unitaire* car il teste une unité de programmation indépendamment des autres, c'est à dire une fonction dans le cadre de ce cours.

On reprend la fonction précédente :

```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
    return nb_objets // places_par_carton
```

La documentation de la fonction `nb_cartons` contient donc 3 tests (on parle d'un *jeu de tests*).

Pour le premier test, l'expression à évaluer est `nb_cartons(10, 5)` et le résultat attendu est `2`. Les valeurs des paramètres (ici `10` et `5`) sont appelées des *données de test*.

L1test va évaluer ce test de la manière suivante :

- il évalue `<expr>` et obtient une valeur, appelons-là le résultat obtenu
- il évalue `<expr_attendue>` et obtient une autre valeur, appelons-là le résultat attendu
- il compare le résultat obtenu et le résultat attendu[2].

Il y a 3 *verdicts* possibles :

- Le résultat obtenu est égal au résultat attendu : le test est un *succès* (*success* en anglais). Il sera affiché en *vert*. La fonction renvoie le bon résultat pour cet exemple.
- Le résultat obtenu n'est pas égal au résultat attendu : le test est un/en *échec* (*failure* en anglais). Il sera affiché en *rouge*. La fonction ne renvoie pas le bon résultat pour cet exemple, ou alors on s'est trompé en écrivant le test.

[1] Cette syntaxe est directement inspirée du module Python `doctest`
[2] On verra plus tard comment se fait exactement cette comparaison en Python.

- l'évaluation de `<expr>` ou `<expr_attendue>` a déclenché une erreur[3] : le test est *en erreur* (*error* en anglais). Il sera affiché en rouge. Il va falloir trouver d'où vient cette erreur.

1.2 Cas des calculs approchés

On a vu que les calculs sur flottants sont approchés :

```
>>> 1.1 + 2.2 - 3.3
4.440892098500626e-16
```

Supposons la fonction suivante (à titre illustratif) :

```
def tiers(x:int) -> float:
    [...]
    return x / 3
```

On pourrait utiliser uniquement dans les exemples des données de test "qui tombent juste", par exemple :

```
>>> tiers(9)
3.0
```

Mais il semble compliqué d'écrire des tests pour les cas "qui ne tombent pas juste", par exemple :

```
>>> tiers(10)
3.3333333333333335
```

La valeur `3.3333333333333335` n'est guère informative. On préférerait par exemple indiquer que la fonction renvoie `3.33` à `1e-2` près.

L1test propose une construction qui permet d'écrire des tests basés sur une tolérance :

```
$$$ tiers(10)
approx(3.33, 1e-2)
```

Ce test passe, car $|3.3333333333333335 - 0.33| \leq 10^{-2}$.

On écrira donc :

- un code qui effectue le calcul sans tronquer le résultat
- des tests qui expriment la valeur attendue avec un précision donnée.

```
def tiers(x:int) -> float:
    """Renvoie le tiers de x.

    Précondition : /
    $$$ tiers(10)
    approx(3.33, 1e-2)
    """
    return x / 3
```

Dans le cas général un test de calcul approché s'écrit ainsi[4] :

```
$$$ <expr>
approx(<expr_attendue>, <tol>)
```

L1test va évaluer ce test de la manière suivante :

- il évalue `<expr>` et obtient une valeur, appelons-là le résultat obtenu
- il évalue `<expr_attendue>` et obtient une autre valeur, appelons-là le résultat attendu

Ensuite :

- si $|obtenu - attendu| \leq tol$ le test passe ;
- si $|obtenu - attendu| > tol$ le test échoue ;

[3] Le message indique `Test raises exception`, la levée d'exception étant le mécanisme par lequel Python gère les erreurs.
[4] Cette syntaxe est directement inspirée de l'outil pour Python `pytest`.

- le test est en erreur si l'évaluation de l'une ou l'autre expression déclenche une erreur.

On peut aussi utiliser la tolérance par défaut prévue par L1test, qui est `1e-6` : `approx(<expr>)` équivaut à `approx(<expr>, 1e-6)`.

Par exemple :

```
$$$ 1.1 + 2.2
approx(3.3)
```

passe car $4.440892098500626e - 16 < 10^{-6}$.

```
$$$ tiers(10)
approx(3.33)
```

échoue car $|3.3333333333333335 - 3.33| > 10^{-6}$.

NB : `approx` est utilisable tel quel uniquement dans les L1tests. Son utilisation dans l'interpréteur produira un `NameError`.

2 Exemples de verdicts

2.1 Avec un code réduit à `pass`

Si on reprend la fonction `nb_cartons` en commentant la ligne qui contient le `return`, la fonction ne renvoie plus de valeur. Dans ce cas, Python renvoie par convention la valeur spéciale `None`.

Pour les 3 tests le résultat obtenu est donc `None` et les 3 tests échouent avec un message du type:

```
Test failed for: nb_cartons(10, 5)
Expected: 2, Got: None
```

ce qu'on peut traduire par "la valeur attendue est `2` alors que la valeur obtenue est `None`".

2.2 Avec un code correct

On revient à un corps de fonction correct[5] pour `nb_cartons`.

Supposons que la docstring contienne le test suivant :

```
$$ nb_cartons(10, 5)
2
```

Oups ! Il manque un caractère `$` dans l'invite… ces 2 lignes sont donc considérées comme du texte libre dans une docstring et pas comme un L1test, qui n'est donc pas exécuté.

Supposons maintenant que la docstring contienne le test suivant :

```
$$$ nb_cartons(0, 5)
2
```

Cette fois on a peut-être fait un copier-coller[6] et oublié de modifier la valeur attendue, le test échoue avec un message en rouge du type :

```
Test failed for: `nb_cartons(0, 5)`
Expected: 2, Got: 0
```

Si par inadvertance nous écrivons un test *qui n'a pas de sens*, en utilisant une donnée de test qui viole la précondition de la fonction[7] avec la valeur `0` pour `places_par_carton` :

```
$$$ nb_cartons(5, 0)
0
```

[5] Un code est correct s'il est conforme à sa spécification, l'énoncé de l'exercice par exemple.
[6] Il est fortement déconseillé de construire des tests par copier-coller.
[7] Une fonction doit par convention être appelée avec des valeurs des paramètres qui satisfont sa précondition.

La division entière par `0` calculée dans le corps de la fonction produira une erreur et le test est en erreur avec un message en rouge du type :

```
Test raises exception for: `nb_cartons(5, 0)`
Zero division error: float division by 0
```

Les 3 tests précédemment écrits dans la docstring initiale de `nb_cartons` passent, avec un message en vert du type :

```
Test OK for: `nb_cartons(10, 5)`
```

2.3 Avec un code incorrect

Supposons maintenant qu'on écrive par inadvertance un code faux, par exemple en oubliant la 2ème barre oblique et en utilisant ainsi une division flottante.

```
def nb_cartons(nb_objets:int, places_par_carton:int) -> int:
    """Renvoie le nombre de cartons qu'on peut remplir avec `nb_objets`
    avec des cartons de `places_par_carton` places.

    Précondition : nb_objets >= 0 et places_par_cartons > 0
    Exemple(s) :
    $$$ nb_cartons(10, 5)
    2
    $$$ nb_cartons(11, 5)
    2
    $$$ nb_cartons(0, 5)
    0
    """
    return nb_objets // places_par_carton
```

Le premier test passe, car `2` et `2.` sont considérées par L1Test (et Python) comme des valeurs égales. Le troisième test passe de même car les valeurs `0` et `0.0` sont considérées comme égales.

Il est important de noter que ces 2 tests **ne détectent pas l'erreur**. Donc il est possible que les tests passent mais que le code soit néanmoins incorrect. **Des tests au vert ne garantissent pas un code correct.**

Le deuxième test échoue et permet de détecter l'erreur:

```
Test failed for: `nb_cartons(11, 5)`
Expected: 2.2, Got: 2
```

3 Questions fréquentes

3.1 Pourquoi écrire les tests avant le code ?

- pour vérifier qu'on a bien compris le travail que doit effectuer la fonction à coder, d'après sa spécification ;
- parce que si on écrit les tests après avoir écrit le code, on risque de choisir des valeurs attendues en regardant le code, et non la spécification de la fonction. On risque donc d'écrire des tests faux qui valident un code faux ! (et ça n'arrive pas qu'aux débutants !) ;
- pour ces raisons, les méthodes actuelles de développement logiciel sont des méthodes "test first".

Pour ceux qui continueront en Licence Informatique, le cours de programmation orientée objet (POO) en L2 (Resp : Jean-Christophe Routier) repose aussi sur l'écriture obligatoire de tests avant d'écrire le code. La figure 1 montre un extrait du cours dédié aux tests en 2023, avec l'aimable autorisation de JC Routier.

3.2 Si un test échoue, c'est que le code de la fonction est faux ?

Soit le code de la fonction est faux, soit le test est mal écrit.

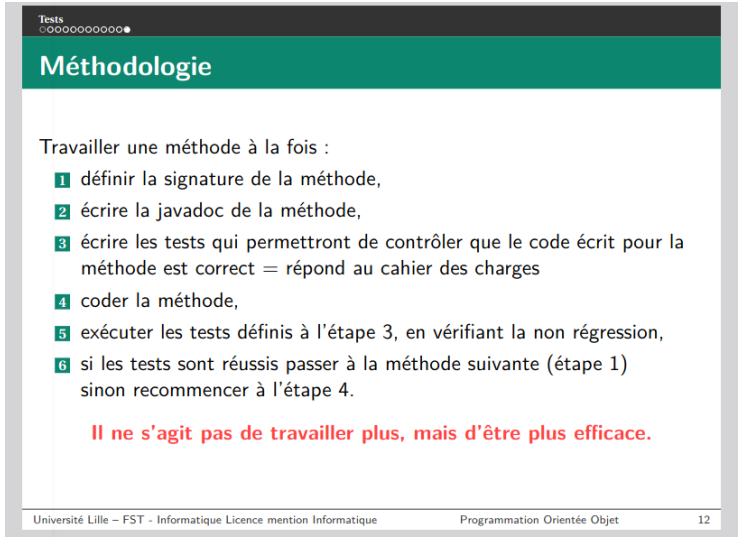

Figure 1: Méthodologie dans l'UO POO de L2 Informatique

3.3 Si tous les tests passent, le code est correct ?

Malheureusement non. Exécuter des tests peut mettre en évidence une erreur dans le code d'une fonction, mais ne peut pas prouver que le code est correct. En effet on peut très rarement tester toutes les valeurs que peuvent prendre les paramètres d'une fonction. Mais des tests au vert permettent d'avoir une plus grande confiance dans la correction du code, surtout si les jeux de tests sont bien choisis.

3.4 Comment bien choisir ses jeux de tests ?

C'est un art qui s'acquiert avec l'expérience.

On choisira classiquement des *valeurs représentatives* des différents comportements.

Pour le calcul du nombre de cartons on a choisi d'abord :

- un jeu de test `(nb_objets = 10, places_par_carton = 5)` qui permet de caser tous les objets dans les cartons;
- puis un jeu de test `(nb_objets = 11, places_par_carton = 5)` pour lequel il reste des objets hors cartons ;

On a choisi des valeurs de test produisant un résultat parlant pour le lecteur de la fonction qui peut faire le calcul de tête.

On a choisi aussi des valeurs dites "*aux limites*", c'est à dire qui illustrent les bornes du domaine de la fonction : le jeu de test `(nb_objets = 0, places_par_carton = 5)` montre que la valeur `0` est acceptable pour `nb_objets`.

Si l'énoncé stipule qu'un paramètre x doit être positif, on ne sait pas trop si la valeur `0` est acceptable. Si elle l'est, on choisira cette valeur comme donnée de test.

3.5 C'est vraiment utile de tester ?

C'est la méthode de validation de programme la plus utilisée, y compris dans le monde professionnel. Pour vous, c'est une excellente manière de savoir "si vous avez l'air d'avoir bon" sur ordinateur, l'autre manière étant la relecture de code attentive et la réflexion sur papier.

3.6 Mais alors, plus besoin de réfléchir ?

Disposer d'un jeu de test automatisé ne dispense pas de réfléchir ! Il est nécessaire de se creuser la tête pour écrire un code correct, trouver un algorithme, ou encore pour comprendre pourquoi certains tests ne passent pas.

Écrire un code non réfléchi puis essayer par tous les moyens de faire passer les tests au vert est une approche de la programmation qui ne produit pas de code correct !

Les tests servent de filet de sécurité, ils rassurent le · a programmeur · euse sur la qualité de son code. Ils permettent aussi de détecter de manière automatisée une régression dans le code, ce qui arrive quand une modification dans le code à un endroit déclenche une erreur à un tout autre endroit.

3.7 Peut-on toujours tester une fonction ?

Avec L1test, non. C'est un outil pour débutant. D'autres bibliothèques de test plus avancées existent, enseignées en Licence et Master mention Informatique.

4 Memento

- un jeu de tests est un ensemble de tests incluant appel de fonction sur des données de test et résultat attendu de l'exécution de l'appel
- un test est dit unitaire quand il teste une petite unité de code, dans notre cas une fonction
- la chaîne de documentation `dosctring` inclut un jeu de tests dans une section `Exemples`
- selon le résultat de son exécution, un test peut être un succès, un échec ou être en erreur
- les tests des fonctions qui renvoient une valeur de type `float` utilisent la fonction de L1Test `approx`
- des tests qui passent augmentent la confiance qu'on peut avoir dans son code, mais il peut y subsister des erreurs non détectées
- l'outil `L1test` permet d'automatiser et systématiser l'exécution des tests unitaires

CC BY