CORRECTION

[ ]:
```
while True:
    try:
        x = int(input("Saisir une date"))
        break
    except ValueError:
        print("Il faut saisir un nombre entier entre 1 et 365")
```

À retenir

Un grand programme est décomposé en plusieurs **modules**, dont chacun est dédié à la réalisation d'une **tâche précise**. L'**interface** d'un module décrit l'ensemble des fonctions offertes par ce module.
Avec le principe d'**encapsulation** :

1. il suffit de connaître l'interface pour utiliser convenablement un module,
2. le développeur du module possède un cadre pour modifier, corriger, améliorer son programme sans nuire aux autres programmes utilisant ce module.

On complète l'encapsulation d'un module en gérant explicitement à l'aide d'**exceptions** les utilisations non conformes de son interface.

Chap. 2 — Modularité

1.1 — Un exemple : le paradoxe des anniversaires

Exemple

[14]:
```
# Programme 1
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = set()
    for x in t:
        if x in s:
            return True
        s.add(x)
    return False
```

Exemple

[15]:
```
# Programme 2
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = []
    for x in t:
        if x in s:
            return True
        s.append(x)
    return False
```

Exemple

[16]:
```
# Programme 3
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = [False] * 366
    for x in t:
        if s[x]:
            return True
        s[x] = True
    return False
```

1.2 — Factorisation du code

Les trois programmes précédents se ressemblent beaucoup car ils font la même chose (mais avec des stratégies complètement différentes) et tous les trois ont

la même structure :

- `s` représente d'une manière ou d'une autre un ensemble de dates qu'il faut *créer*.
- Il faut vérifier si `s` *contient* l'élément `x`.
- Il faut être capable d'*ajouter* l'élément `x` à `s` si besoin.

Ce qui donne, en délégant ces trois aspects aux fonctions `cree()`, `contient()` et `ajoute()` :

```
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = cree()
    for x in t:
        if contient(s,x):
            return True
        ajoute(s,x)
    return False
```

REMARQUE

Cette factorisation du code a de nombreux avantages :

- pour **changer le mode de représentation** des dates, il ne faut plus changer `contient_doublon()`
- l'ensemble de dates peut être **réutilisés** dans d'autres programmes
- il y a **séparation** entre le programme qui utilise les dates et les programmes qui définissent comment sont programmées en interne ces dates.

Ces trois fonctions représentent l'**interface** entre le programme qui utilise l'ensemble de dates **et** les programmes qui définissent d'une façon ou d'une autre cet ensemble.

CORRECTION

La fonction `int()` lève une exception `ValueError`.

Pour rattraper une exception, on va utiliser les mots-clés `try` et `except`.

Exemple

Pour rattraper une exception `ValueError`, on va utiliser le mot clé `try` suivi du symbole : et d'un *premier bloc*. Ensuite, le mot-clé `except` suivi du nom de l'exception et du symbole : précède un *deuxième bloc* de code.

```
try:
    x = int(input("Entrer une date"))
except ValueError:
    print("Prière de saisir un entier valide")
```

Le premier bloc est le bloc *normal*. Si son exécution s'achève normalement (sans lever d'exception) le second bloc est ignoré.

Le second bloc est le bloc *alternatif*. Si une exception est levée dans le bloc normal, alors l'exception est comparée avec le nom précisé à la ligne `except`. Si les noms correspondent, l'exception est **rattrapée** et le bloc alternatif est exécuté **avant** de passer à la suite. Sinon, le programme s'interrompt (sauf si le tout est inclu dans une autre construction `try`/`except`).

ACTIVITÉ

Proposer un code demandant à l'utilisateur une date à l'utilisateur tant que la date saisie est invalide.

```
[35]: def ecrit(t,i,v):
          if i < 0:
              raise IndexError('Indice négatif')
          t[i] = v
```

ACTIVITÉ

Dans le module `date`, **modifier** la fonction `ajoute` afin de lever une exception si la date n'est pas dans l'intervalle 1..365.

Justifier le type d'exception choisi.

CORRECTION

```
[37]: def ajoute(s,x):
          if x < 1 or x > 365:
              raise ValueError("date", str(x), "invalide")
          s[x] = True
```

3.3 Rattraper une exception

Si des exceptions sont prévisibles et correspondent à des situations connues, il est parfois préférable de ne pas interrompre le programme.

Par exemple en demandant à un utilisateur une date, il est tout à fait possible qu'il ne saisisse pas un nombre entier.

ACTIVITÉ

Relever l'exception levée lorsque l'utilisateur ne saisit pas un nombre entier lors de l'exécution du code ci-dessous

```
x = int(input("Entrer une date"))
```


2.1 Modules

Une des clés du développement à grande échelle consiste à séparer proprement les différentes parties d'un programme.

Exemple

Par exemple on peut séparer la définition d'une structure de données (comme l'ensemble de dates) et son utilisation.

On peut aussi séparer la partie interface graphique d'une application de sa partie logique qui en constitue le cœur.

Chaque morceau de code peut être placé dans un fichier différent appelé **module**.
Pour importer les fonctions définies dans un module et les utiliser, il faut utiliser le mot clé `import`

Exemple

Par exemple, pour importer le module permettant de gérer les valeurs aléatoires, on importe `random` grâce à l'instruction `import random`. Auquel cas, pour utiliser la fonction `randint()` du module, il faut écrire `random.randint()`. Mais sous cette forme, cette façon d'importer est à éviter car ce sont *toutes* les fonctions du modules qui sont importées.

Il est préférable de n'importer que la ou les fonctions utiles. Par exemple `from random import randint` ne va importer que la fonction `randint(a,b)` qui permet de choisir aléatoirement un nombre entier compris entre `a` et `b` inclus.

Exemple

Par exemple, pour créer son propre module, il suffit de sauvegarder dans un fichier `monModule.py` les fonctions.

Pour importer les fonctionnalités d'un module, il faut que le fichier `monModule.py` soit dans le même répertoire puis alors il suffit d'utiliser le mot clé `import` en écrivant : `from monModule import ...`.

Exemple

Par exemple, les fonctions `cree()`, `ajoute()` et `contient()` peuvent être sauvegarder dans le fichier `dates.py`.

```
def cree():
    return [False] * 366

def contient(s,x):
    return s[x]

def ajoute(s,x):
    s[x] = True
```

Ensuite, si l'on souhaite utiliser notre ensemble de date dans un programme, il suffit d'écrire `from date import cree, ajoute, contient` en ayant préalablement placé le fichier `date.py` dans le même répertoire que le fichier de travail `anniversaire.py` :

```
from dates import cree, contient, ajoute
def contient_doublon(t):
    """le tableau t contient-il un doublon ?"""
    s = cree()
    for x in t:
        if contient(s,x):
```


ACTIVITÉ

Tester le code ci-dessous et indiquer le nome de l'**exception** levée.

```
t = [1,1,2,5,14,42,132]
print (t[12])
```

CORRECTION

Exception `IndexError` car l'indice 12 n'existe pas dans le tableau. Il y a 7 valeurs donc les indices appartiennent ici à 0..6.

3.2 Signaler un problème avec un exception

Il est possible de lever manuellement toutes ces exceptions en faisant suivre le mot clé `raise` du nom de l'exception, lui même suivi d'une chaîne de caractère donnant l'information sur l'erreur signalée.

```
raise IndexError('Indice trop grand')
```

ACTIVITÉ

Définir une fonction `ecrit(t,i,v)` qui affecte la valeur `v` à l'emplacement `t[i]` d'un tableau **et** qui lève une exception si l'indice est négatif.

Pourquoi n'y a-t-il pas besoin de l'instruction `else` ?

CORRECTION

3.1 Les exceptions

Selon l'implémentation d'un module, une mauvaise utilisation des fonctions de l'interface risque d'engendre des erreurs ou des effets qui ne peuvent pas être compris et anticipés facilement. Il faudrait alors que l'utilisateur étudie le code du module ce qui contredit le principe de l'encapsulation.

Une bonne pratique est de renvoyer à l'utilisateur des **erreurs explicites** et de pratiquer une **programmation defensive**.

> Lorsqu'un programme s'interrompt à cause d'une *erreur*, il affiche des messages variés. En programmation, ces *erreurs* sont appelées des **exceptions**.

Lorsqu'une exception est levée (c'est-à-dire détectée par l'interprète Python), l'exécution du programme s'interrompt **sauf si** une prise en charge spécifique a été prévue par le développeur.

Exemple

Par exemple, voici les exceptions classiques en Python :

exception	contexte
`NameError`	accès à une variable inexistante
`IndexError`	accès à un indice invalide dans un tableau
`KeyError`	accès à une clé inexistante d'un dictionnaire
`ZeroDivisionError`	division par zéro
`TypeError`	opération appliquée à des valeurs incompatibles


```
            return True
        ajoute(s,x)
    return False
```

ACTIVITÉ

Après avoir écrit votre module `date.py`, **écrire** un programme permettant de savoir combien d'élève il faut en moyenne dans une école pour qu'un anniversaire soit fêté chaque jour.

Pour cela, tirer au hasard des dates et les stocker dans un ensemble jusqu'à ce que toutes les dates aient été obtenues au moins une fois.

Répéter cette expérience 1000 fois et afficher une valeur moyenne.

CORRECTION

[29]:
```
from dates import cree, contient, ajoute
from random import randint

def fete_continue():
    compteur = 0
    nombre_dates = 0
    s = cree()
    while nombre_dates < 365:
        compteur += 1
        x = randint(1,365)
        if not contient(s,x):
            nombre_dates += 1
            ajoute(s,x)
    return compteur

n = 0
for _ in range(1000):
    n += fete_continue()

print("En moyenne", n/1000, "élèves")
```

```
En moyenne 2388.026 élèves
```

2.2 Interfaces

Pour chaque module, on distingue :

- son **implémentation** : c'est-à-dire le code lui même et
- son **interface**, consistant en une énumération des fonctions définies dans le module qui sont destinées à être utilisées dans la réalisation d'autres modules, appelés *clients*.

L'interface doit expliciter ce qu'un utilisateur a besoin de connaître des fonctions proposées : *comment* et *pourquoi* les utiliser.

L'objectif est que :

1. ces fonctions soient suffisantes pour permettre à un utilisateur de faire appel aux fonctionnalités du module et
2. que ces fonctions soient utilisées sans avoir besoin d'aller consulter le code du module.

Pour chaque fonction il faut :

- un nom
- la liste des paramètres
- sa spécification, c'est-à-dire les conditions auxquelles la fonction peut être appliquée et les résultats à attendre.

REMARQUE

La documentation de l'interface peut être vue comme un **contrat** entre l'auteur du module et ses utilisateurs.

C'est mieux si le nombre de choses à lire est limité, facile à comprendre et à mémoriser.

Exemple

Par exemple, voici l'interface de l'ensemble de dates.

fonction	description
`cree()`	crée et renvoie un ensemble de dates vide
`contient(s,x)`	renvoie `True` si et seulement si l'ensemble `s` contient la date `x`
`ajoute(s,x)`	ajoute la date `x` à l'ensemble `s`

2.3 Encapsulation

Comme l'auteur d'un module est libre de s'y prendre comme il le souhaite pour respecter l'interface, il peut donc utiliser toute une série de fonctions ou d'objets annexes. Ces éléments *internes* ne doivent pas être utilisés par les modules clients.

Ces éléments *hors interface* sont qualifiés de **privés** et on parle d'**encapsulation** pour dire qu'ils sont enfermés et que l'utilisateur n'a pas a connaître le contenu.

Exemple

En Python, pour indiquer que certains éléments (variables, fonctions) sont privés, on fait précéder leur nom par le symbole _. Cette écriture n'est une *convention* qu'il vaut mieux respecter. Mais rien n'empêche l'accès aux éléments privés d'un module.

D'autres langages mieux adaptés aux projets à grande échelle introduisent un contrôle stricte de l'encapsulation.