Modularité, mise au point de programmes et gestion de bugs

Cours de spécialité NSI de Terminale

D Pihoué

Lycée Camille Jullian Bordeaux

28 septembre 2023

Capacités attendues

1. Utiliser des API (*Application Programming Interface*) ou des bibliothèques.
2. Exploiter leur documentation.
3. Créer des modules simples et les documenter.
4. Savoir répondre aux causes typiques de bugs : typage, effet de bord non désiré, débordement, instruction conditionnelle non exhaustive, choix des inégalités, comparaisons et calculs entre flottants, mauvais nommage des variables, etc.

Par abstraction et généralisation, on dégage les traitements liés à la structure de données choisie des opérations spécifiques à la recherche d'un doublon.

Par abstraction et généralisation, on dégage les traitements liés à la structure de données choisie des opérations spécifiques à la recherche d'un doublon.

Abstraction et généralisation

Trois actions portant sur la structure de données se détachent :

1. création d'un objet ayant cette structure de données
2. test de présence d'une valeur dans cet objet
3. ajout d'une valeur dans cet objet.

On associe chacune de ces actions à une fonction pour implémenter la structure de données.

On associe chacune de ces actions à une fonction pour implémenter la structure de données.

Abstraction et généralisation

Trois fonctions pour ces trois actions portant sur la structure de données :

1. `creerEnsemble` pour créer un objet
2. `estPresent` pour tester si une valeur est présente dans l'objet
3. `ajouteElement` pour ajouter une valeur dans l'objet

On associe chacune de ces actions à une fonction pour implémenter la structure de données.

Abstraction et généralisation

Trois fonctions pour ces trois actions portant sur la structure de données :

1. `creerEnsemble` pour créer un objet
2. `estPresent` pour tester si une valeur est présente dans l'objet
3. `ajouteElement` pour ajouter une valeur dans l'objet

La fonction générique `contientDoublon` va alors s'écrire de la manière suivante.

Fonction générique

```
1  def contientDoublon(tab : list) -> bool:
2    """Cette fonction renvoie la valeur True
3        si le tableau passé en argument
4        contient au moins un doublon.
5       Elle renvoie False sinon.
6    """
7    dejaVus = creerEnsemble()  # ens vide
8    for elt in tab:
9        if estPresent(elt, dejaVus) :  # si elt ...
     ...est déjà dans l'ensemble
10           return True  # c'est un doublon
11       ajouteElement(elt, dejaVus)  # sinon il ...
     ...est ajouté
12   return False  # car pas eu interruption
```

Définition

Les deux entités ainsi séparées

- la structure de données et ses traitements
- la fonction générique qui les utilise

se nomment des **modules**.

Définition

Les deux entités ainsi séparées

- la structure de données et ses traitements
- la fonction générique qui les utilise

se nomment des **modules**.

Cette séparation conduit à deux observations importantes.

Définition

Les deux entités ainsi séparées

- la structure de données et ses traitements
- la fonction générique qui les utilise

se nomment des **modules**.

Cette séparation conduit à deux observations importantes.

1. La structure de données ou les implémentations des trois fonctions qui lui sont spécifiques peuvent être modifiées sans qu'il y ait nécessité d'intervenir sur le code de la fonction `contientDoublon`.

Définition

Les deux entités ainsi séparées

- la structure de données et ses traitements
- la fonction générique qui les utilise

se nomment des **modules**.

Cette séparation conduit à deux observations importantes.

1. La structure de données ou les implémentations des trois fonctions qui lui sont spécifiques peuvent être modifiées sans qu'il y ait nécessité d'intervenir sur le code de la fonction `contientDoublon`.
2. La structure de données et ses trois fonctions pourraient être réutilisées dans d'autres situations et même complétées sans que la fonction `contientDoublon` ne soit concernée.

Pour du développement de code à grande échelle, il est nécessaire de circonscrire et de séparer les différentes parties du programme envisagé.

Pour du développement de code à grande échelle, il est nécessaire de circonscrire et de séparer les différentes parties du programme envisagé.

Un grand programme est ainsi décomposé en plusieurs **modules** dont chacun est lié à la réalisation de tâches précises.

Pour du développement de code à grande échelle, il est nécessaire de circonscrire et de séparer les différentes parties du programme envisagé.

Un grand programme est ainsi décomposé en plusieurs **modules** dont chacun est lié à la réalisation de tâches précises.

Dans les projets importants, des modules peuvent être regroupés dans une **bibliothèque**.

Exemple

Deux exemples de modules et deux de bibliothèques connus.

Pour du développement de code à grande échelle, il est nécessaire de circonscrire et de séparer les différentes parties du programme envisagé.

Un grand programme est ainsi décomposé en plusieurs **modules** dont chacun est lié à la réalisation de tâches précises.

Dans les projets importants, des modules peuvent être regroupés dans une **bibliothèque**.

Exemple

Deux exemples de modules et deux de bibliothèques connus.

- `math`
- `turtle`
- `matplotlib` dont un module est `pyplot`
- `PIL` dont un module est `Image`

Quelques notions et leurs désignations

Quelques notions et leurs désignations

- **Dépendances**, il s'agit d'autres modules auxquels le module considéré fait référence pour son fonctionnement.

Quelques notions et leurs désignations

- **Dépendances**, il s'agit d'autres modules auxquels le module considéré fait référence pour son fonctionnement.
- **Interface**, il s'agit des fonctions offertes par le module à ses dépendances.

Quelques notions et leurs désignations

- **Dépendances**, il s'agit d'autres modules auxquels le module considéré fait référence pour son fonctionnement.
- **Interface**, il s'agit des fonctions offertes par le module à ses dépendances.
  Elle peut être considérée comme une **abstraction** du module, une description formelle des fonctions offertes.

Quelques notions et leurs désignations

- **Dépendances**, il s'agit d'autres modules auxquels le module considéré fait référence pour son fonctionnement.
- **Interface**, il s'agit des fonctions offertes par le module à ses dépendances.
  Elle peut être considérée comme une **abstraction** du module, une description formelle des fonctions offertes.
- **Réalisation** du module, il s'agit de son **implémentation**, soit son code.

Quelques notions et leurs désignations

- **Dépendances**, il s'agit d'autres modules auxquels le module considéré fait référence pour son fonctionnement.
- **Interface**, il s'agit des fonctions offertes par le module à ses dépendances.
  Elle peut être considérée comme une **abstraction** du module, une description formelle des fonctions offertes.
- **Réalisation** du module, il s'agit de son **implémentation**, soit son code.
- **Encapsulation**, un module peut comporter des fonctions ou des objets annexes qui ne figurent pas dans son interface. On les qualifie de **privés**.

Quelques notions et leurs désignations

- **Dépendances**, il s'agit d'autres modules auxquels le module considéré fait référence pour son fonctionnement.
- **Interface**, il s'agit des fonctions offertes par le module à ses dépendances.
  Elle peut être considérée comme une **abstraction** du module, une description formelle des fonctions offertes.
- **Réalisation** du module, il s'agit de son **implémentation**, soit son code.
- **Encapsulation**, un module peut comporter des fonctions ou des objets annexes qui ne figurent pas dans son interface. On les qualifie de **privés**.
- **Couplage**, entre deux modules, quand une modification sur l'un en implique sur l'autre.

Encapsulation en Python

Il est possible d'indiquer que certains éléments d'un module, variables globales ou fonctions, sont privés en faisant précéder leur nom par le symbole underscore _.

Tous les autres éléments de ce module sont alors réputés publics et donc appartiennent à l'interface. Ils doivent être décrits en ce sens.

L'utilisation d'éléments privés d'un module source dans un module client est très fortement déconseillée.

La documentation des éléments de chaque module est **impérative**.

La documentation des éléments de chaque module est **impérative**. Pour les fonctions, elle prend la forme de *docstrings* qui sont les textes écrits entre deux couples de trois guillemets après la signature.

La documentation des éléments de chaque module est **impérative**.
Pour les fonctions, elle prend la forme de *docstrings* qui sont les textes écrits entre deux couples de trois guillemets après la signature.
Ces textes doivent renseigner sur

- ce que fait la fonction,
- ses pré-conditions,
- éventuellement ses post-conditions et des exemples d'utilisation.

La documentation des éléments de chaque module est **impérative**.
Pour les fonctions, elle prend la forme de *docstrings* qui sont les textes écrits entre deux couples de trois guillemets après la signature.
Ces textes doivent renseigner sur

- ce que fait la fonction,
- ses pré-conditions,
- éventuellement ses post-conditions et des exemples d'utilisation.

Le module lui-même peut faire l'objet d'une introduction par un texte écrit en tête et avec la même syntaxe.

Le code doit être accompagné de **commentaires**, courts textes précédés d'une double espace, du symbole dièse # et encore d'une espace, pour le décrire ou préciser certaines actions non évidentes à comprendre par une lecture rapide.

Le code doit être accompagné de **commentaires**, courts textes précédés d'une double espace, du symbole dièse # et encore d'une espace, pour le décrire ou préciser certaines actions non évidentes à comprendre par une lecture rapide.
Le choix des noms des variables et des fonctions est en soi un commentaire.

Le code doit être accompagné de **commentaires**, courts textes précédés d'une double espace, du symbole dièse # et encore d'une espace, pour le décrire ou préciser certaines actions non évidentes à comprendre par une lecture rapide.
Le choix des noms des variables et des fonctions est en soi un commentaire.
L'ensemble de la documentation d'un module peut être affiché dans la console.

Le code doit être accompagné de **commentaires**, courts textes précédés d'une double espace, du symbole dièse # et encore d'une espace, pour le décrire ou préciser certaines actions non évidentes à comprendre par une lecture rapide.
Le choix des noms des variables et des fonctions est en soi un commentaire.
L'ensemble de la documentation d'un module peut être affiché dans la console.

Exemple

```
import module
help(module)
help(module.fonction)
```

Un module peut à la fois

- offrir des fonctions ou des variables en interface
- être exécutable avec un code principal.

Un module peut à la fois

- offrir des fonctions ou des variables en interface
- être exécutable avec un code principal.

Un attribut particulier

Tout module possède un attribut nommé `__name__`, celui-ci

- renvoie le nom du module s'il a été importé
- renvoie la chaîne de caractères `'__main__'` s'il est exécuté.

Un module peut à la fois

- offrir des fonctions ou des variables en interface
- être exécutable avec un code principal.

Un attribut particulier

Tout module possède un attribut nommé `__name__`, celui-ci

- renvoie le nom du module s'il a été importé
- renvoie la chaîne de caractères `'__main__'` s'il est exécuté.

Cette chaîne `'__main__'` est le nom de l'environnement d'exécution principal.

Exemple

1. Dans une console, la saisie de l'instruction `__name__` provoque le renvoi de la valeur `'__main__'`.
2. Dans la console, la saisie des deux instructions `import nsi_dates` puis `nsi_dates.__name__` provoque le renvoi de la valeur `'nsi_dates'`.

Exemple

1. Dans une console, la saisie de l'instruction `__name__` provoque le renvoi de la valeur `'__main__'`.
2. Dans la console, la saisie des deux instructions `import nsi_dates` puis `nsi_dates.__name__` provoque le renvoi de la valeur `'nsi_dates'`.

Dans le premier cas, désignation de l'environnement principal d'exécution et dans le second nommage du module importé.

On peut exploiter cet attribut pour ajouter à la fin du code du module, une très courte série d'instructions en cas d'exécution du module et non d'importation.

On peut exploiter cet attribut pour ajouter à la fin du code du module, une très courte série d'instructions en cas d'exécution du module et non d'importation.

Exemple

Insérons à la fin du code du module du paradoxe des anniversaires

```
# programme principal
if __name__ == "__main__":
    produitNuagePoints(frequenceCumuleeSucces(500))
    plt.show()
```

qui s'exécutera lors d'un interprétation du code ou par saisie dans une console shell

```
python3 nsi_moduleParadoxe.py
```

Définition

Une erreur correspondant à la détection d'un problème qui empêche la bonne exécution d'un programme est nommée une **exception**.

Définition

Une erreur correspondant à la détection d'un problème qui empêche la bonne exécution d'un programme est nommée une **exception**.

L'opération `raise` permet de déclencher directement une exception, on dit **lever une exception**, en donnant en argument du nom de l'exception levée, une chaîne de caractères qui renseigne sur celle-ci.

Une liste des exceptions les plus courantes avec des éléments de contexte qui aident à leur analyse.

Une liste des exceptions les plus courantes avec des éléments de contexte qui aident à leur analyse.

Une liste d'exceptions

Exception	Contexte
`SyntaxError`	Erreur de parenthèses ou une ponctuation est manquante.
`IndentationError`	Indentation oubliée ou ajoutée.
`NameError`	Accès à une variable inexistante.
`IndexError`	Accès à un indice invalide d'un tableau.

Une liste d'exceptions

Exception	Contexte
`KeyError`	Accès à une clef inexistante dans un dictionnaire.
`ZeroDivisionError`	Division par zéro.
`TypeError`	Opération appliquée à des valeurs qui sont de types incompatibles.
`ValueError`	Un paramètre effectif inadapté est donné à une fonction.

Un exemple pour lever une exception.

Exemple

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

Un exemple pour lever une exception.

Exemple

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

Avec cette opération et un peu de programmation défensive à l'aide d'instructions conditionnelles, il est alors possible d'informer la personne utilisant le module d'un usage inadapté par des messages en référence à l'interface.

Cette notion de type permet de classifier les objets en fonction de leur nature.

Cette notion de type permet de classifier les objets en fonction de leur nature.

Types des valeurs en Python

Valeurs	Type	Description
`1`	`int`	Nombres entiers
`3.14`	`float`	Nombres décimaux
`True`	`bool`	Booléens
`"abc"`	`str`	Chaînes de caractères
`None`	`NoneType`	Valeur indéfinie
`(1, 2)`	`tuple`	*n*-uplets
`[1, 2, 3]`	`list`	Tableaux ou listes
`{1, 2, 3}`	`set`	Ensembles
`{'a':1, 'b':2}`	`dict`	Dictionnaires

Cette notion de type permet de classifier les objets en fonction de leur nature.

Types des valeurs en Python

Valeurs	Type	Description
`1`	`int`	Nombres entiers
`3.14`	`float`	Nombres décimaux
`True`	`bool`	Booléens
`"abc"`	`str`	Chaînes de caractères
`None`	`NoneType`	Valeur indéfinie
`(1, 2)`	`tuple`	*n*-uplets
`[1, 2, 3]`	`list`	Tableaux ou listes
`{1, 2, 3}`	`set`	Ensembles
`{'a':1, 'b':2}`	`dict`	Dictionnaires

La fonction `type` permet d'obtenir le type de la valeur passée en paramètre.

Une bonne pratique est d'annoter les variables et les fonctions afin d'afficher clairement le type de données qui leur est associé.

Une bonne pratique est d'annoter les variables et les fonctions afin d'afficher clairement le type de données qui leur est associé.

Exemple

Un exemple pour une variable et deux autres pour des fonctions.

```
n : int = 2022

def maFonction(arg1 : int, arg2 : list) -> bool, ...
    ...float:

def maFonction(arg1 : tuple, arg2 : int = 2) -> dict...
    ...:
```

Une bonne pratique est d'annoter les variables et les fonctions afin d'afficher clairement le type de données qui leur est associé.

Exemple

Un exemple pour une variable et deux autres pour des fonctions.

```
n : int = 2022

def maFonction(arg1 : int, arg2 : list) -> bool, ...
    ...float:

def maFonction(arg1 : tuple, arg2 : int = 2) -> dict...
    ...:
```

En Python, ces annotations ont uniquement un rôle de documentation.

Ces types restent cependant assez superficiels mais il est possible d'être plus précis en exploitant des **types paramétrés**.

Ces types restent cependant assez superficiels mais il est possible d'être plus précis en exploitant des **types paramétrés**.

Exemple

Ces exemples de types paramétrés nécessitent d'importer le module `typing`.

Types paramétrés	Description
`Tuple[int, bool]`	Couple formé d'un entier et d'un booléen.
`List[int]`	Tableau ou liste d'entiers.
`Set[str]`	Ensemble de chaînes de caractères.
`Dict[str, int]`	Dictionnaire dont les clefs sont des chaînes de caractères et les valeurs des entiers.

Exemple

Un exemple d'utilisation

```
from typing import Tuple
t : Tuple[int, bool] = (54, True)
```

Exemple

Un exemple d'utilisation

```
from typing import Tuple
t : Tuple[int, bool] = (54, True)
```

Il est possible d'associer un *alias* à un type paramétré.

Exemple

Deux exemples d'*aliasing* pour désigner des structures de données.

```
from typing import Tuple
Ensemble = List[int]
Chose = List[List[float]]  # un type composé
```

La mise au point d'un programme consiste à corriger les erreurs identifiées par des tests.

La mise au point d'un programme consiste à corriger les erreurs identifiées par des tests.
La conception de tests demande une réflexion propre voire le développement de fonctions adaptées pour soumettre le programme testé à une série d'appels ou de contrôles bien ciblés.

La mise au point d'un programme consiste à corriger les erreurs identifiées par des tests.
La conception de tests demande une réflexion propre voire le développement de fonctions adaptées pour soumettre le programme testé à une série d'appels ou de contrôles bien ciblés.

Techniques

Pour identifier des erreurs au cours du développement du programme, on peut

- intégrer des traces dans le code avec des instructions `assert`,
- ou des `print`, pour afficher des valeurs intermédiaires par exemple,
- ou utiliser l'outil de *débogage* fourni avec l'éditeur Python utilisé pour analyser une exécution pas à pas du programme.