Modularité

Sommaire

Sommaire

Présentation

- La modularité est une notion intervenant en *génie logiciel* lors de la conception des grands programmes [1].
- Il s'agit de développer les différentes parties de sorte que
  - le développement de chacune puisse se faire indépendamment des autres [2],
  - les «briques» logicielles créées puissent être réutilisables, plus faciles à maintenir et à modifier.
- La conception correcte de modules [3] conduit, en plus du code lui-même, à la réalisation de son *interface*, à l'aide des outils de documentation et demande de respecter la notion d'*encapsulation* [4].
- Pour respecter ce principe d'encapsulation, on doit gérer les éventuelles erreurs lors d'appels incorrects des fonctions offertes par le module, ce qui passe par la *gestion des exceptions*.

1. À titre d'exemple, savez-vous quel est l'ordre de grandeur du nombre de lignes de code d'un programme tel qu'un système d'exploitation ?
2. ce qui est important quand plusieurs développeurs travaillent sur un même projet.
3. le terme de module est celui qui convient en Python, mais dans d'autres langages il existe une terminologie semblable.
4. notion qui sera revue bientôt quand on abordera la Programmation Orientée Objet.

Sommaire

Exemple

- Les API sont une illustration de la modularité, où plusieurs applications peuvent collaborer au sein d'un même programme.

Ci-dessous, on donne deux réalisations possibles en Python d'une même fonction. Pour cette fonction `mystere(t)`, la variable `t` est un tableau d'entiers, compris entre 1 et 366 (représentant un jour dans l'année).

```
def mystere1(t):
    s = []
    for x in t:
        if x in s:
            return True
        s.append(x)
    return False
```

```
def mystere2(t):
    s = [False] * 367
    for x in t:
        if s[x]:
            return True
        s[x] = True
    return False
```

Ex. 1

1. Que fait la fonction `mystere` ?
2. Comparer les deux implémentations proposées en terme d'efficacité (complexité en temps) et de taille mémoire (complexité en espace).

- D'autres implémentations de la même fonction sont encore possibles, comme celle proposée ci-dessous [5].

```
def mystere3(t):
    s = 0
    for x in t:
        if s & (1<<x) != 0:
            return True
        s = s | (1<<x)
    return False
```

- On remarque que le corps de la fonction réalisée est le même pour les 3 implémentations. Plutôt que de reprendre à chaque nouvelle implémentation l'intégralité du code, cela conduit à l'idée de rassembler dans un même endroit les parties qui sont les seules à changer, en assignant à chacune une fonction qui réalise la tâche voulue.

5. Basée sur la notion de tableau de bits. On rappelle en Annexe le fonctionnement des opérations bit à bit qui sont utilisées dans cette implémentation.

- On obtient alors les deux fichiers suivants :

```python
from dates import cree, contient, ajoute

def mystere(t):
    s = cree()
    for x in t:
        if contient(s, x):
            return True
        ajoute(s, x)
    return False
```

Code source 1 – Fichier `main.py`

```python
def cree():
    return [False] * 367

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

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

Code source 2 – Fichier `dates.py`

- Avec cette solution :
  - le choix de la structure de données réellement utilisée pour réaliser les fonctions du module devient *interne* au code de celle-ci et n'a pas à être connu des utilisateurs de cette fonction : on parle d'*encapsulation*,
  - le code proposé par le module est réutilisable par des programmes ayant des besoins similaires.

Sommaire

La modularité en Python

- En Python, le nom du module est simplement le nom du fichier de code sans l'extension `.py`
- Il est crucial, dans cette démarche de modularisation, de réaliser l'*interface* du module, c'est à dire de fournir à l'utilisateur, pour chaque fonction, les informations par lesquelles il sait :
  - ce que réalise précisément chaque fonction, en particulier si elle renvoie quelque chose et la nature de ce qui est renvoyé,
  - comment elle doit être appelée (valeurs correctes des arguments d'appels).

Ex. 2 Réaliser l'interface du module `dates` à l'aide des chaînes de documentation ou «doc-string». Vérifier, dans l'interpréteur, que celles-ci sont bien affichées à l'aide de la commande `help()`. Vérifier également, à l'aide de la commande `dir()` que le contenu du module est bien affiché.

- La bibliothèque standard de Python contient des modules que vous avez déjà utilisés : `random`, `math`, `turtle`, ...
- On rappelle qu'il existe plusieurs syntaxes pour importer des modules :

```
import math
import math as m
from math import sqrt, cos, pi
from math import *
```

Ex. 3 En supposant qu'on veuille affecter à `x` la valeur $\sqrt{2}$, comment faut-il coder cette affectation selon le type de syntaxe d'importation du module `math` qui a été choisie parmi les 4 proposées ci-dessus ? Discuter les avantages/inconvénients des différentes syntaxes.

- En Python, la notion de bibliothèque est «au-dessus» de celle de module, au sens où une bibliothèque fournit souvent plusieurs modules. Exemple : la bibliothèque `matplotlib` contient le module `pyplot` : `import matplotlib.pyplot as plt`

Sommaire

La gestion des exceptions

- si une erreur se produit à l'exécution d'une fonction fournie par un module, un message d'erreur du type ci-dessous est susceptible d'apparaître :

```
>>> import dates
>>> s = dates.cree()
>>> dates.ajoute(s, 400)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/fred/src/dates.py", line 8, in ajoute
    s[x] = True
IndexError: list assignment index out of range
```

- les conséquences sont :
  - l'utilisateur est informé que s est implémenté à l'aide d'un tableau, alors qu'il n'a pas à le savoir (principe d'encapsulation),
  - le message d'erreur n'est pas assez explicite pour lui et ne lui permet pas forcément de corriger l'erreur.
- Il faudrait mieux, lors de la conception du module, renvoyer des messages d'erreurs plus explicites, compréhensibles avec la seule connaissance fournie par l'interface du module.

- Les erreurs[6] sont de plusieurs types possibles. Les plus courants sont :

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

- Il est possible de déclencher (on dit plutôt *lever*) une exception, afin d'envoyer un message d'erreur explicite à l'utilisateur. On utilise pour cela la syntaxe `raise nom_de_l_exception("message")`.

6. on parle en fait d'*exceptions*, les deux mots étant synonymes.

- En appliquant cette idée, le code de `ajoute(s,x)` devient

```
def ajoute(s, x):
    if x<0 or x>366:
        raise IndexError("date " + str(x) + " invalide")
    s[x] = True
```

- On peut également souhaiter gérer les exceptions levées par un appel incorrect des fonctions du module et ne pas laisser le programme s'interrompre car les cas de figure susceptibles de provoquer une erreur sont prévisibles.
- Voici un exemple d'une fonction `saisie_date` chargée de saisir et renvoyer une date correspondant à un jour de l'année :

```
def saisie_date():
    while True:
        try:
            x = int(input("Entrer un numéro de jour dans l'année: "))
            if x<1 or x>366:
                raise ValueError
            return x
        except ValueError:
            print("Il faut rentrer un nombre entier entre 1 et 366")
```

Ex. 4 Le tableau ci-dessous rappelle 5 types d'erreurs possibles courantes pouvant être déclenchées par l'interpréteur Python.

	exception	contexte
①	`NameError`	accès à une variable inexistante
②	`IndexError`	accès à un indice invalide d'un tableau
③	`KeyError`	accès à une clef inexistante d'un dictionnaire
④	`ZeroDivisionError`	division par zéro
⑤	`TypeError`	opération appliquée à des valeurs incompatibles

On donne ci-dessous (page suivante) 5 codes-source Python de 5 programmes. Chacun d'entre eux déclenche une des 5 erreurs (et une seule), laquelle ?

```
l = [1, 2, 3]
l[3] = 4
```

Code source 3

```
def f(x):
    """calcule et renvoie l'image de x par une fonction f"""
    return 1-5/(x+2)

# remplissage d'un tableau tab de valeurs d'images de f
tab = [0]*20;   # initialisation avec 20 valeurs nulles
for i in range(-10, 10):    # calcul des images
    tab[i] = f(i)
```

Code source 4

```
nom = input("Donner votre nom SVP: ")
age = eval(input("Donner votre âge SVP: "))
print("Monsieur/Madame"+ nom + "vous avez " + age + " ans.")
```

Code source 5

```
d = {}
d["voiture"] = 4
d["velo"] = 2
d["tricycle"] = 3
print(d["monocycle"])
```

Code source 6

```
x = 3
y = y + 1
```

Code source 7

Sommaire

La notion d'API

- Une API, de l'anglais Application Progamming Interface, est une interface de programmation d'application, autrement dit un protocole permettant d'interagir avec une application (soit pour un humain, soit pour une autre application).
- La notion d'API est très générale : une url, une signature de fonction, un protocole réseau sont des exemples d'API
- Le bon usage d'une API nécessite de s'être familiarisé avec celle-ci par le biais de sa documentation, qui peut être volumineuse...
- Il existe aujourd'hui de nombreuses API Web[7], permettant de récupérer des données par le biais de requêtes HTTP. Les deux principaux formats de données échangées sont les formats XML et JSON.

7. certaines sont gratuites, d'autres payantes.

- Un exemple d'utilisation possible d'une telle API, est l'API Géo, qui permet de récupérer des données géographiques et administratives sur le territoire français. La description de l'API est à l'url https://api.gouv.fr/les-api/api-geo et sa documentation à l'url https://api.gouv.fr/documentation/api-geo.
- On peut, en Python, à l'aide de la bibliothèque `requests` récupérer les données fournies par cette API en récupérant les données renvoyées par une requête HTTP GET correctement formée. Ci-dessous, un exemple où on récupère la population d'une commune dont on connaît le nom.

```
>>> import requests
>>> url="""https://geo.api.gouv.fr/communes?nom=Bourgoin&fields=population&format=json"""
>>> rep = requests.get(url)
>>> rep.status_code
200
>>> rep.json()
[{'population': 27651, 'nom': 'Bourgoin-Jallieu', 'code': '38053', '_score':
↪  0.7071067811865477}]
```

Sommaire

Exercices

Ex. 5 Créer un module `stats` contenant deux fonctions `somme` et `moyenne`. Ces deux fonctions prennent une liste non vide de nombres. La première renvoie la somme des nombres, la deuxième leur moyenne. Tester le module en l'important dans un autre fichier où les deux fonctions sont utilisées. On veillera à tenir compte des points indiqués dans le cours : présence d'une interface et gestion des exceptions. Par la suite, le module pourra être enrichi avec d'autres fonctions calculant d'autres paramètres statistiques : `ecart_type`, `mediane`, `quartiles`,...

Ex. 6 En utilisant uniquement un interpréteur Python, rechercher dans un module nommé `statistics`, inclus dans la bibliothèque standard, s'il existe une fonction calculant la moyenne de plusieurs nombres et une fonction déterminant la médiane. Comment utiliser ces fonctions pour calculer la moyenne et la médiane de plusieurs nombres ?

Ex. 7 On considère un fichier `dessin.py` qui contient le code suivant :

```
from turtle import *
import figures as f

up()
goto(50, -60)
down()
f.rectangle(100, 75, 'red')

up()
goto(-80, -20)
down()
f.triangle(85, 'blue')

up()
goto(20, 60)
down()
f.cercle(60, 'green')
```

Il s'agit d'écrire le fichier `figures.py` qui est importé dans le programme. Lorsque le programme est exécuté, on obtient dans la fenêtre graphique du module `turtle` un rectangle, un triangle et un cercle avec des couleurs différentes. De plus le périmètre et l'aire de chaque figure sont affichés avec le dessin.

Ex. 8 Utilisation d'une API. Consulter le site à l'adresse http://api.open-notify.org/ qui concerne l'ISS, la Station Spatiale Internationale. Utiliser le module `requests` comme dans le cours pour écrire un code en Python.

1. Obtenir les personnes à bord de l'ISS.
2. Obtenir la position de l'ISS.
3. Obtenir les passages de l'ISS au-dessus d'un point.

Sommaire

Annexe

Annexe : opérations bit à bit

Ce paragraphe fait référence à la représentation binaire des nombres, telle qu'elle a été abordée dans le cours de Première et qu'il est conseillé de relire.

- L'opérateur `|` est celui du *ou bit à bit*, c'est à dire le ou qui agit sur la représentation binaire des nombres. Autrement-dit, si `a` et `b` sont des entiers positifs, `a | b` est l'entier dont chaque bit (de sa représentation binaire) est obtenu en faisant un *ou* logique entre les bits (de même rang) des représentations binaires de `a` et `b`. Prenons l'exemple suivant : si `a` $= 40$, sa représentation binaire est[8] `101000`. Si `b` $= 10$, alors il s'écrit `001010` en binaire. Le nombre `a | b` est donc le nombre qui s'écrit en binaire[9] `101010`, c'est à dire 42. On peut vérifier ceci en console avec un interpréteur Python, en tapant `40 | 10`.

8. On n'écrit que les bits de poids faible non nuls, sachant qu'a priori en Python un entier est codé sur 8 octets, soit 64 bits.
9. On rajoute en tête des bits nuls pour avoir le même nombre de bits que a.

`&` L'opérateur `&` est celui du *et bit à bit*, c'est à dire le et qui agit sur la représentation binaire des nombres. L'explication est identique à celle du ou bit à bit en remplaçant le ou logique par le *et* logique. Reprenons `a` $= 40$ et `b` $= 10$. Alors `a & b` s'écrit `001000`, c'est donc le nombre 8. On peut là aussi vérifier ces calculs en console et faire d'autres tests.

`<<` L'opérateur `<<` est celui du *décalage à gauche*. Autrement dit, le nombre `a << n` (où `n` est un entier positif) est obtenu en décalant tous les bits de `a` `n` fois à gauche. Par exemple, si `a` $= 10$, représenté par `1010`, `a << 2` est représenté par `101000`, donc `a << 10 = 40`. On remarque que cela revient à multiplier `a` par $2^{\texttt{n}}$. Donc `1 << n` est le nombre $2^{\texttt{n}}$.

Dans le programme `mystere3`, la variable `s` est un entier qui sert de *tableau de bits* : les bits de sa représentation binaire stockent une information. Plus précisément, le bit de rang `x` va être égal à 0 si la date `x` est présente dans le tableau `t`, et 0 sinon. Les opérations bit à bit interviennent dans :

`s & (1 << x)` Cette expression réalise un *et bit à bit* avec l'entier $2^{\texttt{x}}$, c'est à dire avec l'entier qui n'a que le bit de rang `x` égal à 1. Donc l'expression `s & (1 << x) != 0` est vraie si et seulement si le bit de rang `x` de `s` vaut 1. Cela signifie que la date `x` a déjà été rencontrée dans `t`.

`s = s | (1 << x)` Cette affectation va mettre à 1 le bit de rang `x` de `s`, de sorte qu'on saura que la date `x` a été rencontrée dans `t`.