Séq. 3- Mise au point des programmes et gestion des bugs

Objectifs

- Mettre au point des programmes (utiliser des jeux de tests – assert, Doctest, ...)
- Reconnaître et savoir répondre aux problèmes liés au typage
- Reconnaître et savoir répondre aux effets de bord non désirés, débordements dans les tableaux
- Reconnaître et savoir répondre aux instruction conditionnelle non exhaustive
- Reconnaître et savoir répondre aux choix des inégalités, comparaisons et calculs entre flottants
- Reconnaître et savoir répondre aux mauvais nommage des variables

Ce cours est inspiré des ressources suivantes :

- https://www.lecluse.fr/nsi/NSI_T/langages/mise_au_point/

1 Introduction

A faire vous même 1.

Regardez vidéo : https://www.youtube.com/watch?v=qE0QFxNiQLI

2 Le typage dynamique de python

Guido Von Rossum, l' inventeur du langage informatique python, a fait des choix forts dès le début. Il a toujours voulu laisser un maximum de liberté technique aux développeurs. Il a aussi favorisé l' écriture de « bonnes pratiques » en laissant le choix à chaque développeur de les suivre.

C' est donc tout naturellement que python est un langage à typage dynamique. C' est-à-dire qu' une variable peut changer de type suivant qu' on y stocke un entier, un décimal, une chaîne de caractères, une liste, un dictionnaire, …

Ceci à l' avantage d' en faire un langage souple, facile d' apprentissage et d' utilisation. Par contre, cela peut permettre à des utilisateurs malins et/ou malveillants d' utiliser des programmes python de manière non prévue. Le programme peut alors donner des valeurs inattendues, des valeurs fausses ou peut s' arrêter suite à un problème de fonctionnement.

Le développeur python doit donc veiller à vérifier que les données utilisées sont du type prévu. Pour ce faire, il est recommandé de pratiquer l' annotation de type.

Exemple :

```
def div_entiere(a:int, b:int)-> int:
    ...
```

Il est aussi recommandé de bien documenter la docstring en y précisant le type attendu pour chaque paramètre et le type de la donnée retournée par la fonction.

Exemple :

```
def maFonction(a, b):
    """
    Rôle : calcule a/b
    Paramètres : a le dividende
    b le diviseur non nul
    Retourne : le quotient a/b
    """
```

```
        return(a/b)
```

Il est enfin recommandé de pratiquer la programmation défensive que nous verrons plus loin dans ce cours.

3 Mise au point des programmes

A faire vous même 2.

1. Visionnez la vidéo suivante : https://youtu.be/cCFuNC7L85w de 0’ à 5’52
2. Notez ci-dessous les éléments importants à retenir de cette vidéo .

- Pour éviter les bugs :
  - commencer par corriger les bugs évidents (syntaxe invalide, NameError, TypeError, ZeroDivisionError)
  - prévoir toutes les saisies possibles par les utilisateurs (imaginer le pire)
  - faire des séries de test (tester toutes les entrées, sorties, les cas limites et distinct)
  - faire attention à l’indentation (ne produit pas d’erreur mais des résultats erronés)
  - éviter les effets de bord (modification possiblement non désirées de variable(s) globale(s) dans une fonction)

3.1 Gérer les erreurs

A faire vous même 3.

1. Visionnez la vidéo suivante : https://youtu.be/cCFuNC7L85w de 5’52 à 11’55
2. Quelle structure python permet d’ intercepter les erreurs ?

   try … except … else … finally ...

3. Quel est l’ intérêt d’ intercepter les erreurs?

   Intercepter les erreurs permet de mieux maîtriser le comportement du programme. On évite que celui-ci s’ arrête de manière imprévue et on anticipe le plus de cas possible.

4. Recopiez l’ exemple final donné dans la vidéo :

A faire vous même 4.

Testez les commandes python suivantes et notez les erreurs déclenchées :

```
>>> >>> print(['a','b'))

>>> >>> for i in range(10):
        print(i)

>>> >>> 10/'2'

>>> >>> 10/0

>>> >>> toto= ['a', 'b']
>>> toto[2]
```

```
>>>
>>> >>> toto = {'un':'one', 'deux':'two'}
>>> toto['three']

>>> >>> print(non_existing_var)

>>> >>> mon_fic = open('ghost.txt')

>>> >>> import toto
```

A noter : Certaines erreurs sont déclenchées dès l' exécution du programme, celle-ci ne peuvent être interceptées.

Toutes les erreurs python : https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Il faut désormais savoir lire les messages d' erreur. Comprendre de quel type d' erreur il s' agit. Savoir à quelle ligne elle s' est produite et, bien sur, la corriger.

3.2 Principes pour des tests performants

A faire vous même 5.

1. Revenez au début de la vidéo à 2'45.
2. Quels sont les 3 conseils pour concevoir des tests ?
   - Changer les entrées, examiner les sorties
   - Pour les structures conditionnelles, chaque cas distinct doit être tester au moins une fois
   - Tester un maximum de cas limites

A faire vous même 6.

1. Visionnez la vidéo suivante : https://youtu.be/cCFuNC7L85w de 11'55 à 16'24
2. Quelle structure python permet d' effectuer un test ?

```
>>> assert test_booléen, 'message'
```

3. Qu' est ce qu' une programmation défensive ?

   Dans une fonction, on prévoit une série de tests afin de vérifier que les préconditions sont effectivement satisfaites.

A faire vous même 7.

Reprenez vos cours de 1ère NSI sur les doctests.

Exemple d' utilisation :

```
import doctest
def carre(c) :
    """
    >>> carre(3)
    9
    >>> carre(0)
    0
"""
doctest.testmod()
```

3.3 Risque lié aux boucles non bornées

A faire vous même 8.

1. Visionnez la vidéo suivante : https://youtu.be/cCFuNC7L85w de 16’24 à 18’30
2. Quelle est le risque lié aux boucles non bornées ?

   Une boucle non bornée est une boucle qui s’ interrompe quand la condition de la boucle devient fausse. Si la condition n’ est jamais fausse alors le programme est dans une boucle infinie et ne s’ arrête jamais.

3.4 Risque lié aux flottants

A faire vous même 9.

1. Visionnez la vidéo suivante : https://youtu.be/cCFuNC7L85w de 18’30 à 20’00
2. Expliquez les limites de la gestion des nombres à virgule flottante.

   La traduction des nombres décimaux en nombres binaires n’ est pas parfaite.
   ```
   >>> 0.1+0.2
   ```
   Nous donne 0.3000000000000004 au lieu de 0.3

1. En quoi cette limite peut être un risque pour le bon déroulement d’ un programme ?

   Si nous utilisons des flottants pour définir une condition sur une boucle non bornée, il y a la possibilité pour que l’ égalité ne soit jamais réalisée et que la boucle soit infinie

2. Comment pallier à ce risque ?

   Transformer l’ égalité entre 2 nombres en une soustraction des 2 nombres et en vérifiant que la valeur absolue du résultat soit inférieure à une valeur très petite.

A proscrire :

```
while valeur1 != valeur2 :
    ...
```

A préférer :

```
while abs(valeur1-valeur2) > 1e-3 :
```

4 Les effets de bord

Principe : A l’ intérieur d’ une fonction, des modifications d’ objets mutables (qui ne devraient pas avoir d’ impact sur le monde extérieur à la fonction) ont finalement une influence sur l’ ensemble du programme.

4.1 Passage par référence

A faire vous même 10.

Testez le code suivant :

```
1. def mystere():
2.   print("Dans mystere avant le sort : ", L)
3.   L.sort()
4.   print("Dans mystere après le sort : ", L)
5.   for i in L:
6.     return i == 1
7.
8. def autre_fonction():
9.   print("Dans autre_fonction : ", L)
```

```
10.       assert L == [3,2,1]
11.
12.     L = [3,2,1]
13.     mystere()
14.     autre_fonction()
```

Pourquoi cela est-il étonnant ?

4.2 Variable utilisée avec global

A faire vous même 11.

Testez le code suivant :

```
1. def fonction():
2.     global toto
3.     toto = toto+"-hacked"
4.
5. toto = "je ne dois pas être modifié"
6. fonction()
7. print(toto)
```

Cherchez ce que fait la ligne n° 2 :

Pourquoi cela est-il risqué ?

5 Activité Pratique – Sécuriser la saisie utilisateur

1. Écrivez une fonction demandant à l'utilisateur la saisie d'un entier.

La fonction saisie_entier(question) devra :

- ✓ Demander à l'utilisateur la saisie d'un entier n grâce au texte contenu dans la question.
- ✓ Contrôler que l'utilisateur a bien saisi un entier. Tant que l'utilisateur n'a pas saisi un entier, la fonction repose la même question et affiche le message d'erreur correspond au problème rencontré.
- ✓ Renvoyer l'entier n saisi en retour.

2. Améliorer votre fonction précédente, en ajoutant deux paramètres, début et fin.

La fonction devient alors saisie_entier_controle(question, début, fin) et doit :

- ✓ Contrôler que la saisie est bien un entier (comme dans la fonction précédente).
- ✓ Vérifier que cet entier est bien dans l'intervalle [début, fin].
- ✓ Si début prend la valeur None, alors cela signifie qu'il n'y a pas de borne inférieure : $n \in]-\infty ; fin[$.
- ✓ Si fin (paramètre optionnel) est omis ou prend la valeur None, cela signifie que l'intervalle n'a pas de borne positive: $n \in [début ; +\infty[$.

A noter : Vous pouvez tester et utiliser

```
>>> plus_infini = float('inf')
>>> moins_infini = float('-inf')
```

* Livre Prepabac – P. 20 ex 2

* Livre Prepabac – P. 20 ex 3

* Livre Prepabac – P. 21 ex 4

* Livre Prepabac – P. 21 ex 5

* Livre Prepabac – P. 22 ex 9

* Livre Prepabac – P. 22 ex 10

6 Fiabilisation des algorithmes de base

6.1 Algorithme de calcul de moyenne

```
def moyenne(ma_liste) :
    ma_moyenne = 0
    for i in ma_liste :
        ma_moyenne = ma_moyenne + i
    ma_moyenne = ma_moyenne/len(ma_liste)
    return ma_moyenne
```

1. Copiez-collez cette fonction
2. Testez-la
3. Ajoutez-les préconditions/postconditions dans la déclaration de fonction
4. Faites une programmation défensive

6.2 Algorithme de comptage

A faire vous même 12.

- Voici la fonction permettant de compter la présence d' un caractère dans une chaîne :

```
def occurence(chaine, caractere) :
    nb_occurence = 0
    for i in chaine :
        if i == caractere:
            nb_occurence = nb_occurence+1
    return nb_occurence
```

1. Copiez-collez cette fonction
2. Testez-la
3. Ajoutez-les préconditions/postconditions dans la déclaration de fonction
4. Faites une programmation défensive

6.3 Algorithme de recherche d'un extremum

A faire vous même 13.

- Reprenez cette fonction déjà vu lors d'une activité précédente.

```
def maximum(ma_liste) :
    valeur_max = ma_liste[0]
    for i in ma_liste[1:]:
        if i>valeur_max :
            valeur_max = i
    return valeur_max
```

1. Copiez-collez cette fonction
2. Testez-la
3. Ajoutez-les préconditions/postconditions dans la déclaration de fonction
4. Faites une programmation défensive

7 Compléments

Afin d' améliorer vos programmes et de les rendre de plus en plus fiables :

- Utilisez un outil de déclaration et de suivi des bugs. Ce n'est pas nécessairement quelque chose de compliqué : parfois un simple fichier texte, éventuellement partagé avec vos collaborateurs suffit.
- Utilisez un gestionnaire de versions et faites des sauvegardes (check-in) régulières de votre travail afin de pouvoir revenir en arrière en cas de soucis. Il se peut que votre programme fonctionne à un moment donné et qu'une amélioration conduise à un problème. Pouvoir revenir à une version antérieure qui marchait peut s'avérer utile…
- N'ignorez jamais un problème potentiel sous prétexte qu'il ne peut normalement pas se produire : rendez votre code robuste par ce qu'on appelle la programmation défensive.

8 Utilisation serveur Git

8.1 Introduction

Git (se prononce [ˈgɪt]) est un système de contrôle de version (*VCS : version control system*) décentralisé, gratuit et open source.

Conçu pour gérer tous les projets, des plus petits aux plus grands, avec rapidité et efficacité.

C'est l'outil indispensable pour faire du travail collaboratif sur des documents numériques :

- facilité d'échange de fichiers
- traçabilité des modification
- gestion des conflits (lorsque les mêmes lignes d'un même fichier ont été modifiées par deux collaborateurs) □

8.2 Concepts

8.2.1 Les dépôts (*repositories*)

Ce sont des dossiers, cachés, contenant l'état de tous les fichiers du projet. Ils sont nommés .git (rappel, le point devant les noms de fichier/dossier sous linux signifie qu'ils sont cachés).

- Le **dépôt distant** (*remote repository*) se trouve sur un serveur qui centralise le projet.
- Les **dépôts locaux** (*local repository*) se trouvent sur les ordinateurs de travail des membres du projet.

8.2.2 Les révisions (*commits*)

A chaque fois qu'un utilisateur « commet » un changement sur les fichiers du projet (modification, ajout, suppression, …) cette modification est enregistrée dans une **révision** (*commit*).

Remarque : git utilise un système de stockage très efficace qui permet de n' enregistrer que les modifications des fichiers et pas les fichiers entiers, ce qui économise considérablement l'espace de stockage sur le serveur.

ATTENTION : cette particularité le rend nettement plus efficace sur des fichiers texte que sur des

fichiers binaires !

8.2.3 Copie de travail (*workspace* ou *working copy*)

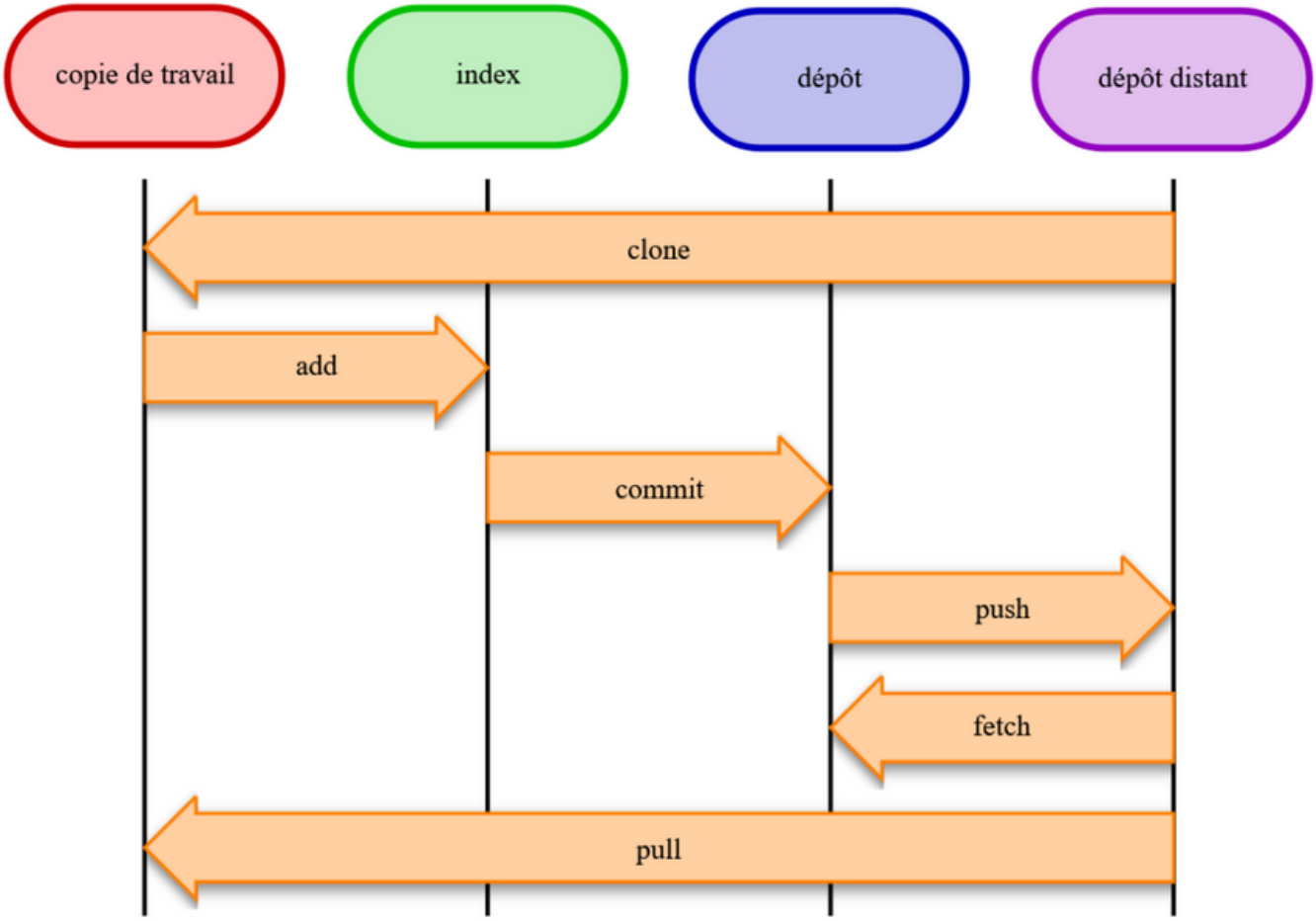
Il s'agit du dossier contenant les fichiers sur lesquels travaille l'utilisateur.

Leur état peut être différent de la dernière révision de l'historique, car d'autres membres du projet peuvent avoir publié des modifications …

8.2.4 L'index

L'index est un espace temporaire contenant les modifications prêtes à être « commitées ».

8.2.5 Les principales opérations git



Pour en savoir plus : https://perso.liris.cnrs.fr/pierre-antoine.champin/enseignement/intro-git/

8.3 Quelques commandes Git

Git dispose notamment des commandes suivantes :

- `git init` : crée un nouveau dépôt ;
- `git clone` : clone un dépôt distant ;
- `git add` : ajoute de nouveaux objets blobs dans la base des objets pour chaque fichier modifié depuis le dernier commit. Les objets précédents restent inchangés ;
- `git status` : Fait un point sur les actions
- `git diff` : Donne le détail des changements
- `git commit` : intègre la somme de contrôle SHA-1 d'un objet *tree* et les sommes de contrôle des objets commits parents pour créer un nouvel objet commit ;

- `git branch` : liste les branches ;
- `git merge` : fusionne une branche dans une autre ;
- `git rebase` : déplace les commits de la branche courante devant les nouveaux commits d'une autre branche ;
- `git log` : affiche la liste des commits effectués sur une branche ;
- `git push` : publie les nouvelles révisions sur le *remote*. (La commande prend différents paramètres) ;
- `git pull` : récupère les dernières modifications distantes du projet (depuis le *remote*) et les fusionne dans la branche courante ;
- `git stash` : stocke de côté un état non commité afin d'effectuer d'autres tâches ;
- `git checkout` : annule les modifications effectuées, déplacement sur une référence (branche, *hash*) ;
- `git switch` : changement de branche ;
- `git remote` : gestion des *remotes*.

8.4 Installation d'un client git GUI

Il existe des client Git dédiés, comme Github Desktop

Ou bien certains IDE intègrent directement leur propre client Git, comme Visual Studio Code par exemple.

94	SmartGit	FREE/PAID	Linux, Mac, Windows	Atlassian JIRA	Conflict Solver and C
87	Fork	$49.99	Windows, macOS	-	-
86	Visual Studio Code	FREE	Windows, macOS, Linux	-	-
80	GitKraken Client	FREE/PAID	Windows, macOS, Linux	-	-
80	Magit	FREE	Any supported by Emacs (Linu...	-	-
79	TortoiseGit	FREE	Windows	-	-
78	GitExtensions	FREE	Windows, Linux via Mono	-	-
--	tig	FREE	Windows, Linux, Mac	-	-
--	Sublime Merge	FREE/$99	Windows, Linux, Mac	-	-
--	gmaster	FREE	Windows	Yes	Full Diff and Merge T
--	GitAhead	FREE / PAID	Windows, Mac, Linux	-	-
65	SourceTree	FREE	Windows, macOS	-	-
62	GitHub Desktop	FREE	-	-	-

A faire vous même 1.

- Installez un client Git de votre choix (de préférence TortoiseGit)

8.5 Inscription sur Framagit

Il est possible d' installer et de maintenir son propre serveur Git mais il faut des compétences et du temps.

Il donc existe plusieurs sites web qui proposent de serveurs Git :

- GitHub
- GitLab
- ...

L' association Framasoft propose un ensemble de services libres et gratuits. Elle propose notamment un serveur Git. C' est celui-ci que nous avons choisi pour nos projets : https://framagit.org/

Pour en savoir plus : https://ohennebelle.frama.io/nsi/0_introduction/ref_framagit/

A faire vous même 2.

- Allez sur le site de framagit.org
- Inscrivez-vous
- Prévenez votre professeur qui vous donnera accès au projet appli_web_bus_dinan
- Connectez-vous au site web et vérifiez que vous avez bien accès au projet
- Avec votre client Git initialisez votre dépôt local Git :
  - Créez un répertoire dédié
  - `git config --global user.name "votre_identifiant_ici"`
  - `git config --global user.email "votre_email_ici"`
  - `git init`
  - `git clone https://framagit.org/termnsi/appli_web_bus_dinan.git`
- Vous devez avoir toutes les données téléchargées dans votre répertoire