LU2IN013 Groupe 3

Projet Robotique

Nicolas Baskiotis
nicolas.baskiotis@lip6.fr

Laboratoire d'Informatique de Paris 6 (LIP6)
Sorbonne Université

S2 (2020-2021)

Plan

Description de l'UE

Objectifs du cours

Apprendre :

- à faire un projet;
- à appréhender un nouvel environnement (Python);
- à gérer un projet (Agile/Scrum, github, trello)
- à travailler en gros et petits groupes
- faire un rapport et une soutenance.

Ce n'est pas :

- un cours approfondi de python,
- que du codage.

Pré-requis

- notions d'algorithmique et de structure,
- de la curiosité,
- de la motivation !

Déroulement de l'UE

En Théorie . . .

- 1h45 de cours/TD le lundi 10h45-12h30;
- 3h30 de TME le mercredi 8h45-12h30;

Ressources

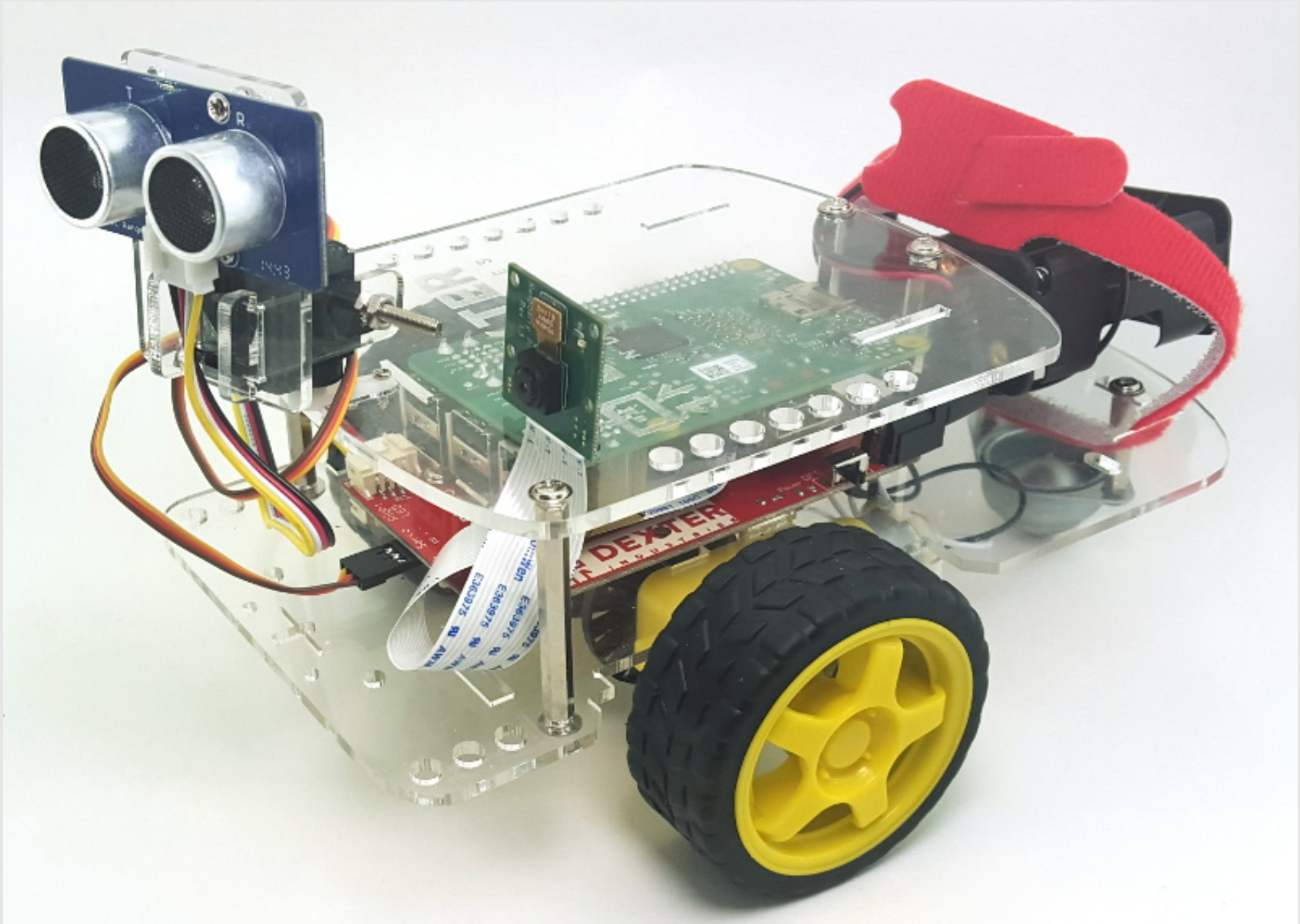
- slides et code sur http://github.com/baskiotisn/2IN013robot2020
- email : (mettre dans le titre `[2IN013]`)
  `nicolas.baskiotis@lip6.fr, olivier.schwander@lip6.fr`
- Discord de la L2, sur `LU2IN013/proj3`

Évaluation

- Projet et soutenance : 50%
- TME noté : 30%
- Code et rapport : 20%

Plan

Robot dexter



Composition du robot

- Un Raspberry Pi
- Une carte contrôleur (arduino)
- Deux moteurs encodeurs pour le contrôle des roues
- 3 senseurs :
  - une caméra
  - un capteur de distance
  - un accéléromètre

Organisation du projet

Première partie commune : effectuer des tâches simples avec le robot

Trois tâches :

- tracer un carré
- s'approcher le plus vite possible et le plus près d'un mur sans le toucher
- suivre une balise

Seconde partie différenciée :

À vous de proposer des tâches

- chasse au trésor
- jeu du chat et de la souris
- détection d'intrusion/patrouille
- ...

Pré-requis : Construire un environnement de simulation pour le robot.

Simulation et IRL

Pourquoi la simulation ?

- Les essais réels coûtent cher !
  - en temps (la simulation est beaucoup plus rapide)
  - en ressources (risque de casser le matériel, impossibilité de tester dans différents environnements, . . .)
- On est confiné (ou presque)
- Flexibilité des situations et des configurations du robot

Problème(s) de la simulation ? Vous découvrirez :)

Code fourni : Rien !

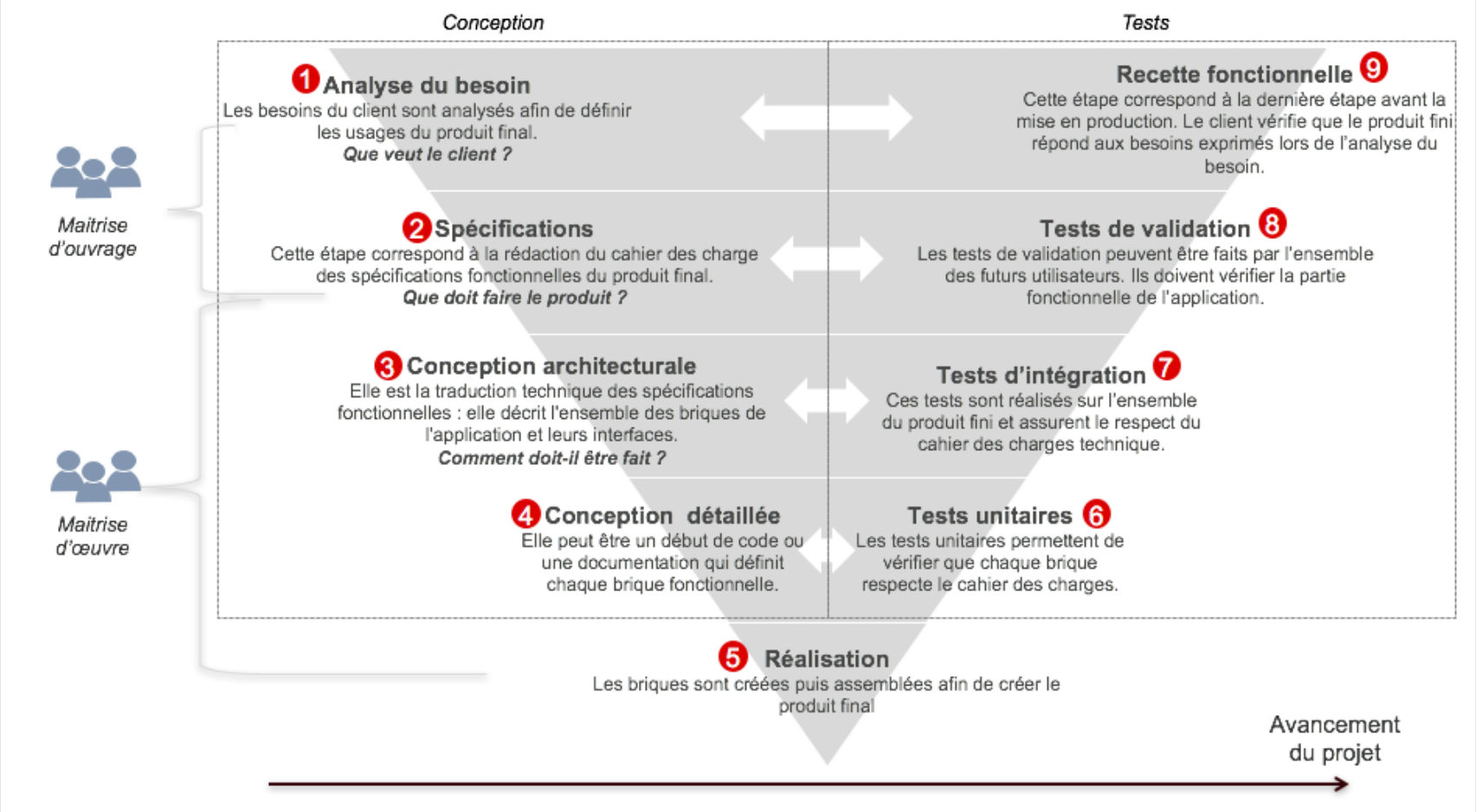
A vous de tout développer . . .

Code demandé : tout

- Première partie : le code de la simulation documenté, avec exemples etc

Plan

La gestion de projet en V



Problèmes récurrents

Inhérent au cycle de gestion

- Plus un projet est grand, moins les exigences sont stables
- Plus un projet est long, moins il y a de chances de succès
- Peu d'interactions entre l'équipe de dev et le client
- 80% de l'usage concerne uniquement 20% des fonctionnalités
- Souvent des surprises lors du déploiement.

Conclusions

- Manque de souplesse : pour terminer une étape, tout en amont doit être fini
- Manque de communication : pas d'interactions entre les différentes équipes
- Péremption du produit : le temps que le projet soit fini, des nouveaux usages ont vu le jour
- Manque de retour client : le client ne voit le produit qu'une fois fini.

Les principes de l'Agile

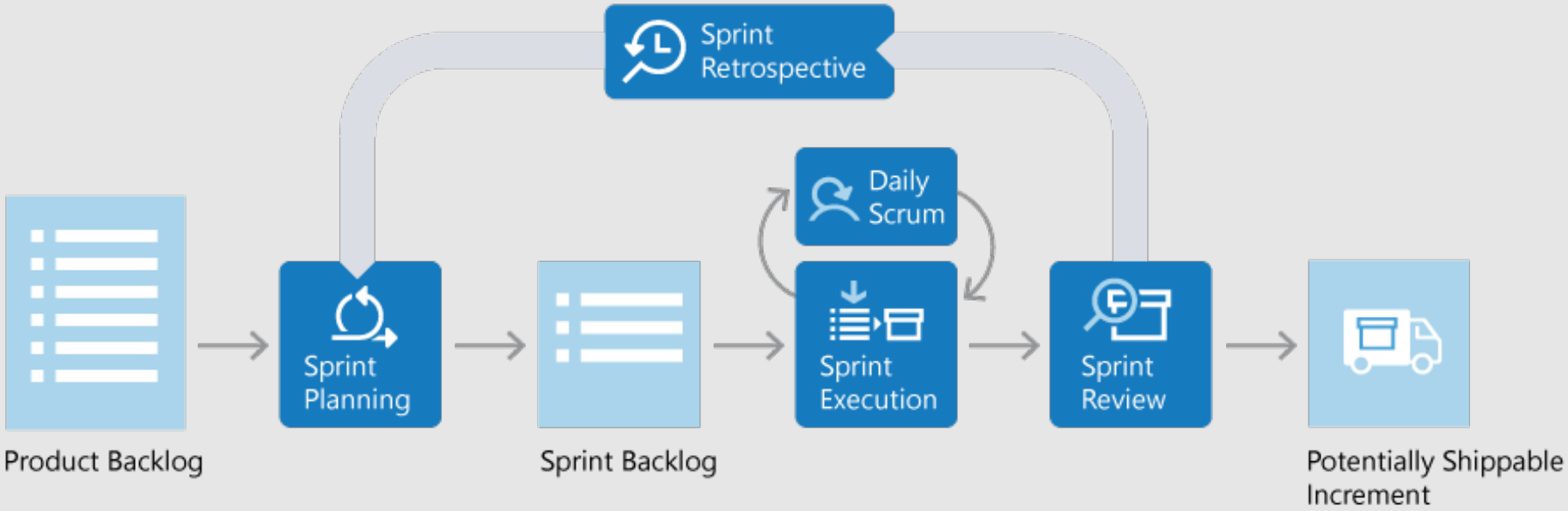
1. Notre plus haute priorité est de satisfaire le client en livrant rapidement et régulièrement des fonctionnalités à grande valeur ajoutée.
2. Accueillir positivement les changements de besoins, même tard dans le projet. Exploiter le changement pour donner un avantage compétitif au client.
3. Livrer fréquemment un logiciel opérationnel avec des cycles de quelques semaines à quelques mois et une préférence pour les plus courts.
4. Les utilisateurs ou leurs représentants et les développeurs doivent travailler ensemble quotidiennement tout au long du projet.
5. Réaliser les projets avec des personnes motivées. Fournissez-leur l'environnement et le soutien dont ils ont besoin et faites-leur confiance pour atteindre les objectifs fixés.
6. La méthode la plus simple/efficace pour transmettre de l'information à l'équipe de développement et à l'intérieur de celle-ci est le dialogue en face à face.
7. Un logiciel opérationnel est la principale mesure d'avancement.
8. Les processus agiles encouragent un rythme de développement soutenable. Ensemble, les commanditaires, les développeurs et les utilisateurs devraient être capables de maintenir indéfiniment un rythme constant.
9. Une attention continue à l'excellence technique et la conception renforce l'agilité.
10. La simplicité - i.e. l'art de minimiser la quantité de travail inutile – est essentielle.
11. Les meilleures architectures/spéc. émergent d'équipes auto-organisées.
12. À intervalles réguliers, l'équipe réfléchit aux moyens de devenir plus efficace, puis règle et modifie son comportement en conséquence.

Les différents rôles et composants de Scrum

Les rôles

- Le Product Owner : qui porte la vision du produit à réaliser (représentant généralement le client).
- Le Scrum Master : garant de l'application de la méthodologie Scrum.
- L'équipe de développement : qui réalise le produit.

Le framework Scrum



Principes

- Les individus et leurs interactions plus que les processus et les outils
- Des logiciels opérationnels plus qu'une documentation exhaustive
- La collaboration avec les clients plus que la négociation contractuel
- L'adaptation au changement plus que le suivi d'un plan

Les composants Scrum

Les composants

- Product Backlog : exigences du produit, listes des features.
- Planification du Sprint : sélection des éléments prioritaires du Product Backlog qu'elle pense pouvoir réaliser au cours du sprint.
- Revue de Sprint : à la fin du sprint, présentation des fonctionnalités terminées au cours du sprint et recueille les feedbacks du Product Owner et des utilisateurs finaux.
- Rétrospective de Sprint : après la revue de sprint, comment s'améliorer (productivité, qualité, efficacité, conditions de travail) en fonction du sprint écoulé (principe d'amélioration continue).
- Daily Meeting : réunion de synchronisation de l'équipe de développement qui se fait debout (elle est aussi appelée "stand up meeting") en 15 minutes maximum au cours de laquelle chacun répond principalement à 3 questions : ≪ Qu'est ce que j'ai terminé depuis la dernière mêlée ? Qu'est ce que j'aurai terminé d'ici la prochaine mêlée ? Quels obstacles me retardent ? ≫

Epic et User stories

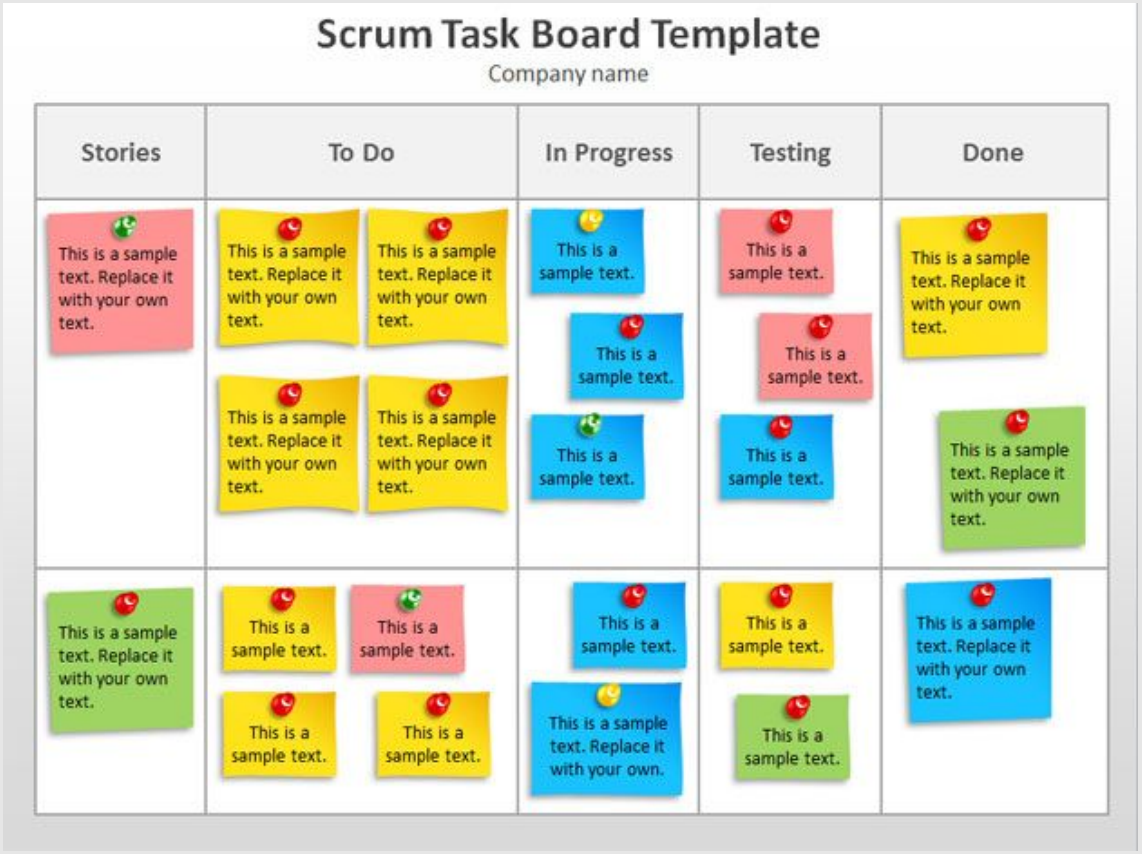
User story

- Petite feature élémentaire à développer/réflechir.
- La plupart du temps, sous format : En tant que [type d'utilisateur], je [veux/peux/être capable de/ai besoin/. . .] de . . . afin de . . . .
- C'est le seul élément sur lequel va travailler un individu/binôme.
- Un temps arbitraire est attribué à chaque story (méthode Poker ou autre).

Epic

- Feature plus générique que les User stories.
- Regroupe en général plusieurs User Stories.
- Tout une facette du produit.

Visualisation des tâches (Board)



Pour naviguer entre les cases :

- Nécessite des critères d'acceptances
- Nécessite un DoR : Definition of Ready
- Nécessite un DoD : Definition of Done

Ce qu'on attend de vous !

DevOps : les outils à votre disposition

- Github (1er TME)
- `trello.com` (Board)
- Discord
- Documentation succinte de la plateforme
- Internet
- Nous (les profs).

A vous de vous organiser mais . . .

Chaque semaine :

- un rapport/mini compte-rendu de la séance, des objectifs, de ce qui a été fait . . .
- une démo
- Vérification du **trello** et planification des nouveaux sprints
- et on avisera ! (sprint retrospective)

Plan

Python : un langage interprété

Peut être exécuté

- en console : interaction directe avec l'interpréteur
- par exécution de l'interpréteur sur un fichier script : `python fichier.py`

Opération élémentaire

```
# Affectation d'une variable
a = 3
# operations usuelles
(1 + 2. - 3.5), (3 * 4 /2 ),  4**2
# Attention ! reels et entiers
1/2, 1./2
# Operations logiques
True and False or (not False) == 2>1
# chaines de caracteres
s = "abcde"
s = s + s # concatenation
# afficher un resultat
print(1+1-2,s+s)
```

Structures : N-uplets et ensembles

Liste d'éléments ordonnes, de longueur fixe, non mutable : aucun élément ne peut être change après la création du n-uplet

```
c = (1,2,3) # creation d'un n-uplet
c[0],c[1]  # acces aux elements d'un couple,
c + c  # concatenation de deux n-uplet
len(c) # nombre d'element du n-uplet
a, b, c = 1, 2, 3 # affectation d'un n-uplet de variables

s = set()  # creation d'un ensemble
s = {1 ,2 ,1}
print(len(s)) #taille d'un ensemble
s.add('s')   # ajout d'un element
s.remove('s') # enlever un element
s.intersection({1,2,3,4})
s.union({1,2,3,4})
```

Structures : Listes

Structure très importante en python.
Il n'y a pas de tableau, que des listes (et des dictionnaires)

```
l = list() # creation liste vide
l1 = [ 1, 2 ,3 ] # creation d'une liste avec elements
l = l + [4, 5] #concatenation
zip(l1,l2) : liste des couples
len(l) #longueur
l.append(6)      # ajout d'un element
l[3]      #acces au 4-eme element
l[1:4]    # sous-liste des elements 1,2,3
l[-1],l[-2] # dernier element, avant-dernier element
sum(l)      # somme des elements d'une liste
sorted(l)      #trier la liste
l = [1, "deux", 3] # une liste compposee
sub_list1 = [ x for x in l1 if x < 2] # liste comprehension
sub_list2 = [ x + 1 for x in l1 ] # liste comprehension 2
sub_list3 = [x+y for x,y in zip(l1,l1)] # liste comprehension 3
```

Structures : Dictionnaires

Dictionnaires : listes indexées par des objets (hashmap), très utilisées également. Ils permettent de stocker des couples (clé,valeur), et d'accéder aux valeurs a partir des clés.

```
d = dict() # creation d'un dictionnaire
d['a']=1   # presque tout type d'objet peut etre
d['b']=2   # utilise comme cle, tout type d'objet
d[2]= 'c'  # comme valeur
d.keys()   # liste des cles du dictionnaire
d.values() # liste des valeurs contenues dans le dictionnaire
d.items() # liste des couples (cle,valeur)
len(d)     #nombre d'elements d'un dictionnaire
d = dict([ ('a',1), ('b',2), (2, 'c')])  # autre methode pour c
d = { 'a':1, 'b':2, 2:'c'}  # ou bien...
d = dict( zip(['a','b',2],[1,2,'c']))  #et egalement...
d.update({'d':4,'e':5}) # "concatenation" de deux dictionnaires
```

Boucles, conditions

Attention, en python toute la syntaxe est dans l'indentation : un bloc est formé d'un ensemble d'instructions ayant la même indentation (même nombre d'espaces précédent le premier caractère).

```
i=0
s=0
while i<10:   # boucle while
    i+=1      #indentation pour marquer ce qui fait parti de la
    s+=i
s=0
for i in [1, 2, 3]: #boucle for
    j = 0           # indentation pour le for
    while j<i:       # boucle while
        j+=1         # deuxieme indentation pour le bloc while
        s = i + j
    s = s + s # retour a la premiere indentation, instruction c
```

Fonctions

```
def increment(x):    # definition d'une fonction par le mot-cle
    return x+1       # retour de la fonction

y=increment(5)       # appel de la fonction

def somme_soustraction(x,y=2):
    # possibilite de donner une valeur par defaut aux parametre
    return x+y,x-y                  #  possibilite de retourner
                    #un n-uplet de valeurs,
                                    # equivalent a (x+y,x-y)
xsom,xsub = somme_soustraction(10,5) #ou
res = somme_soustraction(10,5)
xsom == res[0],res[1]
```

Fichiers

```
##Lire
f=open("/dev/null","r")
print(f.readline())
f.close()

#ou plus simplement
with open("/dev/null","r") as f :
    for l in f:
        print l

## Ecrire
f=open("/dev/null","w")
f.write("toto\n")
f.close()

#ou
with open("/dev/null","w") as f:
    for i in range(10):
        f.write(str(i))
```

Modules

- Un module groupe des objets pouvant être réutilisés
  - module `math` : `cos, sin, tan, log, exp, ...`
  - module `string` : manipulation de chaîne de caractères
  - module `numpy` : librairie scientifique
  - modules `sys`, `os` : manipulation de fichiers et du système
  - module `pdb`, `cProfile` : debuggage, profiling
- importer un module : `import module [as surnom]` et accès au module par `module.fonction` (ou `surnom.fonction`)
- importer un sous-module ou une fonction : `from module import sousmodule`
- tout répertoire dans le chemin d'accès qui comporte un fichier `__init__.py` est considéré comme un module !
- tout fichier python dans le répertoire courant est considéré comme module : `import fichier` si le fichier est `fichier.py` (ou plus souvent `from fichier import *`)

Les objets : très grossièrement

- c'est une structure : contient des variables stockant des informations
- contient des *méthodes* (fonctions) qui agissent sur ses variables,
- contient *un constructeur*, fonction spécifique qui sert à l'initialiser.
- le `.` sert à indiquer l'appartenance d'un objet/fonction à un autre objet : `obj.fun` est l'appel de la fonction `fun` de l'objet `obj`
- *self* indique l'objet lui-même

Un objet `Agent` pourrait être ainsi le suivant :

```
class Agent(object):
    def __init__(self,nom):
        self.nom = nom
        self.x = 0
        self.y = 0
    def agir(self,etat):
        action = None
        return action
    def get_position(self):
        return self.x, self.y
    def safficher(self):
        print(f"Je suis {self.nom}
            en ({self.x},{self.y})")
```

```
a = Agent("John") # creation
a.x, a.y = 1, 1 # deplacement
a.safficher() #equivalent a
Agent.safficher(a)
a.mavar = 4 #ajout d'une variab
```