Sensibilisation à la didactique de l'informatique (2)

Laurianne Foulquier d'après un travail de **Christophe Declercq**

Laurianne.foulquier@u-bordeaux.fr

I. Compétences informatiques

II. Retour sur les deux TP

Si vous deviez identifier des compétences majeures en informatique dans le contexte de la programmation, quelles seraient-elles et pourquoi ?

I. Compétences informatiques « Computational thinking »

II. Retour sur les deux TP

- L'expression "computational thinking" introduite par Jeannette Wing fait référence à des compétences et des habiletés humaines, pouvant être développées à l'occasion d'activité de programmation, et transférables à bien d'autres situations de type "résolution de problème".
- Il ne s'agit pas de penser comme une machine, mais de décrire les compétences cognitives en jeu pour résoudre - entre autres - un problème informatique - ou plutôt pour le faire résoudre par une machine.
- Il s'agit donc d'activité cognitive de haut niveau et donc bien d'une activité humaine. L'énumération des compétences en jeu fait débat. La traduction française aussi fait débat. "Thinking" aurait pu être traduit par "réflexion". Le terme "computationnel" n'existant pas, l'adjectif "calculatoire" est le plus proche mais semble réducteur.

I. Compétences informatiques

II. Retour sur les deux TP

- **Evaluer** : capacité à attribuer mentalement une valeur (résultat, type…) à un programme donné.
- **Anticiper** : capacité à se mettre dans la posture du programmeur qui doit décrire dans un algorithme l'enchaînement séquentiel/répétitif/conditionnel des instructions, avant même le début de son exécution.
- **Décomposer** : capacité à transformer un problème complexe en un ensemble de problèmes plus simples équivalents au problème initial.
- **Généraliser** : capacité à inférer un problème général à partir d'une instance de ce problème, et à repérer dans un problème particulier la répétition de traitements ou de données suivant un même schéma.
- **Abstraire** : capacité à "faire abstraction" des informations non pertinentes et à créer des solutions où la manière dont un problème est résolu peut être "abstraite" à l'aide d'une interface pertinente.

I. Compétences informatiques

II. Retour sur les deux TP

Extrait du préambule commun aux programmes de première et de terminale NSI

Il permet de développer des compétences :

- analyser et modéliser un problème en termes de flux et de traitement d'informations ;
- **décomposer** un problème en sous-problèmes, reconnaître des situations déjà analysées et réutiliser des solutions ;
- concevoir des solutions algorithmiques ;
- traduire un algorithme dans un langage de programmation, en spécifier les interfaces et les interactions, comprendre et réutiliser des codes sources existants, développer des processus de mise au point et de validation de programmes ;
- mobiliser les concepts et les technologies utiles pour assurer les fonctions d'acquisition, de mémorisation, de traitement et de diffusion des informations ;
- développer des capacités d'**abstraction** et de **généralisation**.

EVALUER

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

- Capacité à attribuer mentalement une valeur (résultat, type…) à un programme donné.

- L'évaluation d'un programme - lui donner une valeur - peut se faire de différentes manières dans différents domaines. Le plus simple est bien sûr d'évaluer le résultat que donne le programme à l'exécution. Mais il est aussi utile de savoir évaluer le **type** du résultat sans chercher nécessairement à connaître sa valeur.

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

Un exemple:

- Soit l'expression suivante prévue pour déterminer si une année est bissextile :

```
an % 4==0 and (not an % 100 == 0 or an
% 400 == 0)
```

On peut au préalable calculer le type du résultat de cette expression en fonction du type des arguments. On sait que `==` prend deux arguments de type quelconque et donne un résultat de type `bool`, que `or` et `and` donne des résultats de type `bool` donc l'expression complète donne un résultat de type `bool`.

On peut vérifier :

```
an = 2019
```

```
type(an % 4==0 and (not an % 100 == 0
or an % 400 == 0))
```

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

Evaluer c'est aussi donner la valeur du résultat de l'exécution - ce que fournit l'interprète.

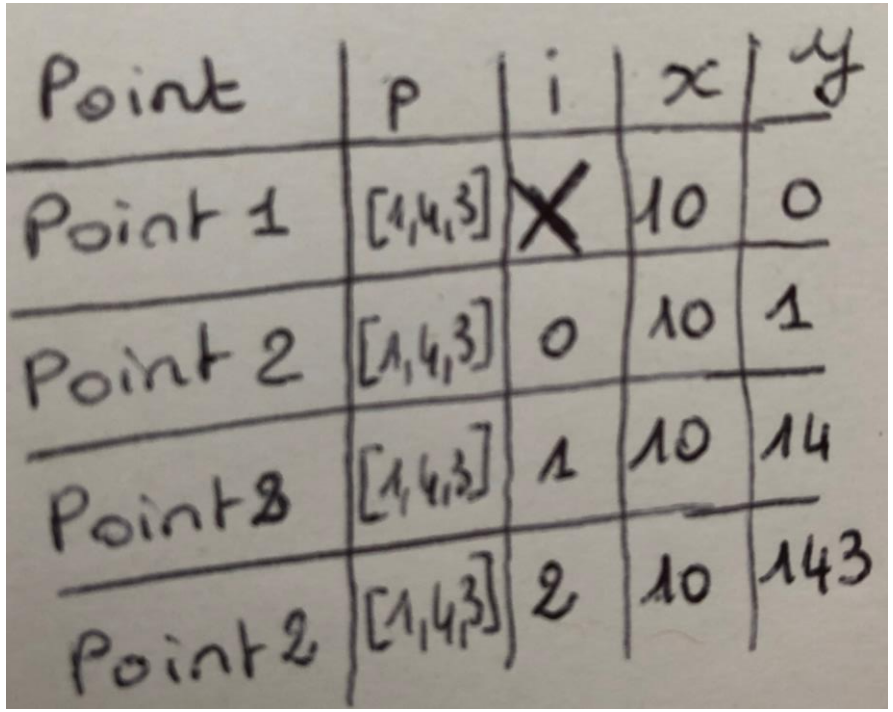
L'évaluation mentale peut être concrète. On peut construire à la main un tableau d'exécution du programme en notant sur chaque ligne les valeurs prises par les variables à certains points d'observation du programme :

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

Exemple : on évalue le résultat du programme suivant.

```
def EvaluerPolynome(p,x):
    y = 0 # Point d'observation 1
    for i in range (len(p)):
        y = y * x + p[i]  # Point d'observation 2
    return(y)
EvaluerPolynome([1,4,3], 10)
```



Point	p	i	x	y
Point 1	[1,4,3]	X	10	0
Point 2	[1,4,3]	0	10	1
Point 3	[1,4,3]	1	10	14
Point 2	[1,4,3]	2	10	143

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

- On peut aussi **évaluer** symboliquement. Dans ce cas on évalue *à la main* le résultat en nommant les valeurs des paramètres : `EvaluerPolynome([a2,a1,a0], x)` et en exécutant symboliquement chacune des affectations.

Point	p	x	y
Point 1	[a2,a1,a0]	x	0
Point 2	[a2,a1,a0]	x	a2
Point 2	[a2,a1,a0]	x	a2 * x + a1
Point 2	[a2,a1,a0]	x	a2 x**2 + a1 x + a0

Dans ce cas l'interprète Python ne peut vérifier le résultat, mais cela permet de raisonner sur le programme et d'obtenir une formulation qui correspond bien à ce que l'on souhaitait calculer.

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

On peut **évaluer** le temps d'exécution d'un programme ou le nombre d'affectations ou de tests exécutés.

```
Tab = [2, 8, 6, 12, 4, 9]
def tri (t):
    for i in range (len(t)-1):
        for j in range(i+1,len(t)):
            if t[j]<t[i]:
                t[i],t[j] = t[j],t[i]

tri(Tab)
```

on peut évaluer à 15 le nombre de tests et à 5 le nombre d'affectations.
Cette évaluation peut se faire pour une donnée concrète : on peut alors instrumenter le programme pour lui faire faire cette évaluation en y ajoutant des compteurs.

I. Compétences informatiques EVALUER

II. Retour sur les deux TP

- De manière générale, **évaluer** un programme consiste donc à regarder ce programme comme une donnée et à en calculer une valeur par une méthode particulière. Ce changement de plan du programmeur consiste à regarder son programme tel qu'il est - et non tel qu'il aurait voulu qu'il soit.
- La compétence *évaluer* est fondamentale pour mettre au point un programme.

ANTICIPER

I. Compétences informatiques ANTICIPER

II. Retour sur les deux TP

- Capacité à se mettre dans la posture du programmeur qui doit décrire dans un algorithme l'enchaînement séquentiel/répétitif/conditionnel des instructions, avant même le début de son exécution.
- C'est une compétence fondamentale pour la conception d'algorithmes. C'est aussi un des principaux obstacles didactiques rencontrés par les programmeurs débutants.

« Une propriété difficile à intégrer [...] est le caractère différé d'une exécution du programme » J. Rogalski, 1986

I. Compétences informatiques ANTICIPER

II. Retour sur les deux TP

Anticiper c'est imaginer

- C'est la part créative du travail du programmeur d'imaginer par quel chemin de calculs intermédiaires on peut passer, pour aller des informations disponibles aux informations que l'on souhaite calculer.
- Anticiper les étapes successives du traitement et les différents cas à envisager, c'est tout l'*art de programmer*. Référence de la formule : The Art of Computer Programming, Donald Knuth.

I. Compétences informatiques ANTICIPER

II. Retour sur les deux TP

Dans le contexte de l'enseignement de spécialité en classe de 1ere

- Même si cette compétence est abordée très tôt dans l'apprentissage de la programmation, et si l'obstacle associé est dépassé très vite, toutes les situations problèmes requérant une solution algorithmique font appel à la maîtrise de cette compétence.
- La difficulté d'une situation vient de la longueur de la chaine d'anticipation nécessaire et de la nécessité d'inventer des états intermédiaires pour anticiper le passage d'un état initial à un état final.
- Cette compétence est liée à la compétence **décomposer** car pour anticiper la résolution d'une situation complexe, il est souvent utile de profiter d'états intermédiaires identifiés pour séparer le problème.
- Le modèle d'exécution sous jacent et sa compréhension par l'élève ont une influence sur la difficulté à anticiper correctement.

I. Compétences informatiques ANTICIPER

II. Retour sur les deux TP

Dans le contexte lié à l'utilisation d'un langage de programmation

- L'ordre d'évaluation des expressions et d'exécution des instructions sont des caractéristiques à prendre en compte au moment d'anticiper l'écriture d'un programme.
- La compétence d'anticipation n'est donc pas seulement une capacité d'analyse *descendante* allant du problème vers sa solution algorithmique. Ce peut aussi être une démarche *ascendante* par composition d'instructions en anticipant leur effet pour aller des informations disponibles vers les informations à calculer.

DECOMPOSER

I. Compétences informatiques DECOMPOSER

II. Retour sur les deux TP

Capacité à transformer un problème complexe en un ensemble de problèmes plus simples équivalents au problème initial.

Cette compétence est fondamentale car c'est elle qui permet d'envisager le traitement de problèmes arbitrairement complexes par réduction à des problèmes plus simples ou déjà connus.

I. Compétences informatiques DECOMPOSER

II. Retour sur les deux TP

Décomposition par cas

- **Exemple** : le calcul du nombre de jours d'un mois peut se décomposer en 3 cas distincs pour traiter les mois de 31 jours, les mois de 30 jours et le cas particulier du mois de février.
- **Exemple** : le problème de la résolution d'une équation du second degré peut se décomposer en 3 sous-problèmes distincts selon la valeur du discriminant négatif, nul ou strictement positif.
- **Exemple** : le problème de la compression d'une image peut être séparé en plusieurs sous-problèmes selon le format d'enregistrement de l'image à compresser et selon le souhait de l'utilisateur de compresser avec ou sans perte d'information.
- On peut dire en analysant ces exemples que la décomposition peut se pratiquer à tous les niveaux de la conception d'un programme : du niveau le plus fin pour séparer quelques instructions au niveau le plus global pour séparer des traitements demandant chacun plusieurs centaines ou milliers de lignes de code.

I. Compétences informatiques DECOMPOSER

II. Retour sur les deux TP

Décomposition séquentielle

- **Exemple** : programmer la soustraction de deux nombres en utilisant le codage en complément à deux, peut se décomposer en 6 étapes consistant à coder chacun des nombres a et b, puis à calculer le code de -b en inversant les bits et en ajoutant 1 au code de b, à additionner les codes obtenus puis à décoder le résultat.

```
def soustraction(a,b):
    codea = coder(a)
    codeb = coder(b)
    codeinvb = inverserbits(codeb)
    codemoinsb = ajouter1 (codeinvb)
    codeamoinsb = additionnercodes(codea,
codemoinsb)
    amoinsb = decoder(codeamoinsb)
    return(amoinsb)
```

- Cette mise en œuvre conserve la décomposition suivie. Des variables intermédiaires ont été imaginées et nommées pour contenir les résultats intermédiaires de la décomposition choisie.
- La solution ne sera complète et exécutable que lorsque toutes les fonctions intermédiaires utilisées auront bien été définies.

I. Compétences informatiques DECOMPOSER

II. Retour sur les deux TP

Mise en oeuvre

- Le résultat d'une démarche de conception utilisant la décomposition d'un problème en problèmes plus simples peut être montré explicitement en utilisant une notion de sous-programme / fonction / procédure pour conserver dans le programme écrit la démarche de décomposition suivie.
- La démarche peut aussi être cachée en recollant ensemble dans un seul et même traitement toutes les parties élémentaires issues de la décomposition. Ceci est le plus souvent déconseillé car cela donne des fragments de programmes plus complexes et donc moins simples à comprendre.

I. Compétences informatiques DECOMPOSER

II. Retour sur les deux TP

Conclusion

Savoir décomposer est une compétence permettant d'appréhender des problèmes complexes. Sa mise en œuvre passe par l'utilisation systématique de primitives de programmation permettant la structuration des programmes - fonctions et procédures mais aussi objets et méthodes. Ces mécanismes permettent aussi, par ailleurs, de mettre en œuvre des démarches de généralisation et d'abstraction.

GENERALISER

I. Compétences informatiques GENERALISER

II. Retour sur les deux TP

Capacité à inférer un problème général à partir d'une instance de ce problème, et à repérer dans un problème particulier la répétition de traitements ou de données suivant un même schéma.

C'est la notion de fonction **avec paramètres** qui permet de mettre une œuvre un programme qui peut résoudre un problème général, dont les instances particulières correspondent aux application de cette fonction avec des valeurs particulières des paramètres.

I. Compétences informatiques GENERALISER

II. Retour sur les deux TP

Exemple: On veut calculer la durée de voyage d'un voyageur ayant pris successivement un long courrier pendant exactement 18h20mn30s suivi d'un vol moyen courrier de durée 6h45mn50s. Ecrire un programme Python permettant de résoudre ce problème.

```
h1 = 18
mn1 = 20
s1 = 30
total1 = 18 * 3600 + 20 * 60 + 30
h2 = 6
mn2 = 45
s2 = 50
total2 = 6 * 3600 + 45 * 60 + 50
total = total1 + total2
mn = total // 60
s = total - mn * 60
h = mn // 60
mn = mn - h * 60
print(h, "h", mn , "mn", s , "s")
```

```
h1 = 18
mn1 = 20
s1 = 30
total1 = 18 * 3600 + 20 * 60 + 30
h2 = 6
mn2 = 45
s2 = 50
total2 = 6 * 3600 + 45 * 60 + 50
total = total1 + total2
mn = total // 60
s = total - mn * 60
h = mn // 60
mn = mn - h * 60
print(h, "h", mn , "mn", s , "s")
```

```
def temps(h, mn, s):
    return (h * 3600 + mn * 60 + s)

total1 = temps(h1, mn1, s1)
total2 = temps(h2, mn2, s2)
```

```
def additionner_et_afficher(h1, mn1, s1, h2,
mn2, s2):
    total1 = temps(h1, mn1, s1)
    total2 = temps(h2, mn2, s2)
    total = total1 + total2
    mn = total // 60
    s = total % 60
    h = mn // 60
    mn = mn % 60
    print(h, "h",mn , "mn",s , "s")

additionner_et_afficher(18, 20, 30, 6, 45,
50)
```

I. Compétences informatiques GENERALISER

II. Retour sur les deux TP

Généralisation par une fonction en paramètre

Dans le cas où l'on repère, entre deux traitements, un même schéma d'algorithme mais cependant quelques différences dans les calculs effectués on peut utiliser la notion de fonction en paramètre.

```
def somme(liste):
    resultat = 0
    for elt in liste:
        resultat = resultat + elt
    return(resultat)

somme([1, 2, 3, 4, 5])
```

```
def produit (liste):
    resultat = 1
    for elt in liste:
        resultat = resultat * elt
    return(resultat)

produit([1, 2, 3, 4, 5])
```

On peut **généraliser** ces deux fonctions en définissant une fonction accumuler qui reçoit comme paramètre une « fonction » opération supposée recevoir deux paramètres.

```
def somme(liste):
    resultat = 0
    for elt in liste:
        resultat = resultat + elt
    return(resultat)

somme([1, 2, 3, 4, 5])
```

```
def produit (liste):
    resultat = 1
    for elt in liste:
        resultat = resultat * elt
    return(resultat)

produit([1, 2, 3, 4, 5])
```

```
def accumuler(operation, neutre, liste):
    resultat = neutre
    for elt in liste:
        resultat =operation(resultat, elt)
    return(resultat)
```

```
def addition(x,y):
    return(x + y)

accumuler(addition, 0, [1,2,3,4,5])
```

```
def multiplication(x,y):
    return(x * y)

accumuler(multiplication, 1,
[1,2,3,4,5])
```

I. Compétences informatiques GENERALISER

II. Retour sur les deux TP

```
def accumuler(operation, neutre, liste):
    resultat = neutre
    for elt in liste:
        resultat =operation(resultat, elt)
    return(resultat)
```

```
def addition(x,y):
    return(x + y)

accumuler(addition, 0, [1,2,3,4,5])
```

```
def multiplication(x,y):
    return(x * y)

accumuler(multiplication, 1, [1,2,3,4,5])
```

On a ainsi, à partir du même schéma, décliné plusieurs instances d'une solution à un problème plus général

I. Compétences informatiques GENERALISER

II. Retour sur les deux TP

Conclusion

- Généraliser est une compétence de haut niveau qui permet au programmeur de résoudre des problèmes plus généraux et ensuite de réutiliser pour des instances particulières des parties de programme déjà écrites.
- La notion de paramètre, utilisée pour instancier des valeurs particulières ou des fonctions particulières est le mécanisme permettant de mettre en oeuvre la généralisation.

ABSTRAIRE

I. Compétences informatiques ABSTRAIRE

II. Retour sur les deux TP

- Capacité à "faire abstraction" des informations non pertinentes et à créer des solutions où la manière dont un problème est résolu peut être "abstraite" à l'aide d'une interface pertinente.

I. Compétences informatiques ABSTRAIRE

II. Retour sur les deux TP

Abstraire avec les données

- Abstraire avec les données consiste à *encapsuler* un certain nombre d'informations en utilisant une structure de données composée qui peut éventuellement masquer le détail du codage des informations.
- **Exemple** : Le programme suivant peut effectuer un calcul de temps si les fonctions intermédiaires utilisées sont définies pour enregistrer les informations sous un format adapté et y accéder. La représentation choisie n'est pas visible à ce niveau. Elle est abstraite. Les informations fournies consistent en des nombres d'heures, minutes et secondes.

```
t1 = temps(18, 20, 30)
t2 = temps(6, 45, 50)
total = additionner_temps(t1, t2)
    afficher_temps (total)
```

I. Compétences informatiques ABSTRAIRE

II. Retour sur les deux TP

```
def temps(h,mn,s):
    return({'h':h, 'mn':mn,'s':s})

def additionner_temps(t1,t2):
    s = t1['s']+t2['s']
    mn = t1['mn']+t2['mn']
    h = t1['h']+t2['h']
    return({'h':h + (mn+s//60)//60,
'mn':(mn+s//60)%60,'s':s%60})

def afficher_temps(t):
    print(t['h'], "h", t['mn'], "mn", t['s'], "s")

t1 = temps(18, 20, 30)
t2 = temps(6, 45, 50)
total = additionner_temps(t1, t2)
afficher_temps (total)
25 h 6 mn 20 s
```

Une autre mise en œuvre peut choisir de tout convertir en secondes.

I. Compétences informatiques ABSTRAIRE

II. Retour sur les deux TP

```
def temps(h,mn,s):
    return((h*60 + mn)*60 + s)

def additionner_temps(t1,t2):
    return(t1 + t2)

def afficher_temps(t):
    print(t // 3600, "h",(t//60)%60 , "mn", t%60, "s")

t1 = temps(18, 20, 30)
t2 = temps(6, 45, 50)
total = additionner_temps(t1, t2)
afficher_temps (total)
25 h 6 mn 20 s
```

Pour l'utilisateur de cet ensemble de fonctions, le choix de la structure de données n'est pas nécessairement visible. Le résultat final est le même dans les deux mises en oeuvre.

I. Compétences informatiques ABSTRAIRE

II. Retour sur les deux TP

Abstraire avec les fonctions

- Les fonctions sont de manière générale un outil d'abstraction qui peut aussi masquer la méthode de calcul utilisée, y compris quand il n'y a pas de question de choix de structure de données.
- **Exemple** : les deux fonctions fact ci-dessous, calculent bien chacune la factorielle d'un entier positif. Elles sont interchangeables pour l'utilisateur.

```
def fact(n):
    f = 1
    for i in range(n):
        f = f * (i+1)
    return(f)

fact(10)
3628800
```

```
def fact(n):
    return(1 if n==0 else n*
fact(n-1))

fact(10)
3628800
```

I. Compétences informatiques ABSTRAIRE

II. Retour sur les deux TP

Conclusion

- Abstraire en cachant soit les données soit les algorithmes, par des fonctions, permet au programmeur de simplifier l'écriture de ses programmes.
- La combinaison des deux méthodes aboutit à la programmation orientée objet où données et méthodes sont encapsulées à l'intérieur d'une classe.

I. Compétences informatiques

II. Retour sur les deux TP

Retour sur des extraits des TP A et B

Exercice 4

Ecrire un programme qui demande à l'utilisateur d'entrer les coordonnées de deux points A et B dans un repère orthonormé puis qui calcule la distance AB.

Exercice 5

a) Ecrire un programme qui demande à l'utilisateur d'entrer 3 nombres puis qui les écrit dans l'ordre croissant (tri croissant).

b) Ecrire un programme qui demande à l'utilisateur d'entrer 3 nombres puis qui les écrit dans l'ordre décroissant (tri décroissant).

Extrait TP B

Concevoir un programme qui compare trois nombres (indication : On pourra appeler plusieurs fois la fonction max…)

BILAN

Bibliographie et sitographie

- C.Declercq, diaporama formation DIU de Nantes
- http://pythontutor.com/
- Dagiene, V., Sentance, S. et Stupuriene, G. (2017). Developing a Two-Dimensional Categorization System for Educational Tasks in Informatics. *Informatica*, 28(1), 23–44.