Informatique 1
Introduction à la programmation

Synthèse de la matière

Réponses aux questions

Kim Mens **Charles Pecheur** **Siegfried Nijssen**

Modalités pratiques de l'examen

- Date : lundi 20 janvier 2025
- Durée : de 8h30 à 11h30 et de 11h45 à 14h15 (3 heures)
  - Soyez présent au moins 15 minutes avant l'heure indiquée
- Examen en présentiel sur INGInious
- Dans un environnement protégé (Safe Exam Browser)
  - Pas d'accès à Thonny / PyCharm / PythonTutor / …
- Dans les salles informatiques de l'université

Des instructions plus détaillées suivront par annonce :

- répartition des locaux
- carte des salles informatiques

Présence à l’examen

- On vous conseille vivement de participer activement à l’examen.
  - pas autorisé à quitter la salle après le démarrage de l’examen
- Néanmoins, si vous ne souhaitez pas passer l'examen
  - Les étudiant·es qui souhaitent peuvent demander une note de présence

Introduction

INGInious

Collapse context

La durée de l'examen est 3 heures.

Cette examen comporte 6 questions.

Chaque question est évaluée indépendamment des autres et possède son contexte.
Des tests vous sont fournis afin de vous aider à évaluer votre code. *Attention*, passer ceux-ci r
question. Veillez donc à ajouter vos propres tests.

Attention qu'ici seule la syntaxe de votre code est évaluée. Un pourcentage à 100% à une question indiquera donc juste un code
syntaxiquement correct. Il faudra en outre qu'il respecte les spécifications afin de passer nos

Dans aucune question il ne sera possible de faire des `import`.

Ressources

Pendant l'examen, vous pouvez consulter les ressources suivantes:

- Syllabus interactif
- Syllabus Format PDF
- Open a new window to INGInious

Zone de test

Voici une zone de test Python. Insérez du code de test Python dans la boîte ci-dessous. Vous pouvez utiliser `print` pour tester votre code. Ce code de test est exécuté en Python, en dehors de toute autre définition. Vous pouvez utiliser cette zone par exemple pour obtenir la documentation Python avec la fonction `help`. Exemple:

```
help(print)
```

Runner python

Il s'agit d'un runner Python libre que vous pouvez utilisez pour exécuter du code Python. L
apparaîtront dans le feedback de retour.

```
1
```

Pendant l'examen, est-ce qu'on a accès au syllabus interactif?

OUI, au syllabus **théorie** + en PDF

Est ce qu'on aura accès à nos codes des différentes missions sur INGInious?

NON, pas au syllabus **d'exercices** ni à vos soumissions sur INGInious

Avons nous le droit d'utiliser les slides du cours ou nos notes?

NON, que le syllabus théorie

Consignes de confidentialité

- Durant l'examen, vous pouvez consulter le **syllabus théorie du cours**
- Mais vous ne pouvez utiliser aucun autre document don't vous disposez ou disponible sur le web :
  - pas le **syllabus d'exercices**, ni vos **soumissions INGInious**, ni vos **missions**, ni les **diapositifs** du cours, ni les **capsules vidéo**, ni vos **notes**
- Vous vous engagez à faire votre examen de manière strictement individuelle, **sans assistance extérieure ni communication** avec d'autres étudiants.
- Un système de **détection de plagiat** sera appliqué à cet examen.
- Vous vous engager, sur votre honneur et votre intégrité académique, à respecter ces consignes de confidentialité.

[Q1] Manipulation de listes

Collapse context

Définissez une fonction `mix(l)` qui prend comme entrée une liste avec un nombre **pair** d'entiers, par exemple : [1,2,3,4, ..., ,97,98,99,100]. La fonction retourne une nouvelle liste de la même taille, par exemple : [1,100,2,99,3,98, ...,50,51], avec comme premier élément le premier élément de la liste originale, comme deuxième élément le dernier élément de la liste originale, comme troisième élément le deuxième élément de la liste originale, comme quatrième élément l'avant-dernier élément de la liste originale...

Plus précisément, pour chaque index *i* pair où $0 \leq i < n$ et *n* est la taille de la liste originale, les éléments aux positions *i* et *i+1* de la nouvelle liste s...

La fonction demandée est la suivante:

```
def mix(l):
    """
    @pre:    l est une liste d'entiers
             la taille n de cette liste est un nombre pair
    @post:   retourne une liste r d'entiers
             la liste retournée r a la même taille n
             pour chaque index 0 ≤ i < n où i est pair on a la
             correspondance suivante entre les deux listes :
                 r[i] = l[i//2]
                 r[i+1] = l[n-1-(i//2)]
    """
```

Si nécessaire vous pouvez aussi définir des fonctions auxiliaires.

Question 1: mix

```
1  def mix(l):    # Ne pas effacer cette ligne
2      """
3      @pre:    l est une liste d'entiers
4               la taille n de cette liste est un nombre pair
5      @post:   retourne une liste r d'entiers
6               la liste retournée r a la même taille n
7               pour chaque index 0 ≤ i < n où i est pair on a la
8               correspondance suivante entre les deux listes :
9                   r[i] = l[i//2]
10                  r[i+1] = l[n-1-(i//2)]
11     """
```

Question 2: Zone de test

Insérez votre code de test optionnel ci-dessous. Vous pouvez utiliser `print` pour tester votre programme. Votre code de test sera exécuté à la s... méthode `mix` et d'éventuelles fonctions auxiliaires ci-dessus (il ne faut pas les recopier ici).

Exemple:

```
1  print(mix([1, 2, 3, 4]))
2  # [1, 4, 2, 3]
3
4  # vos tests
```

Submit

Contexte d'une question...

Détails de la question...

Réponse à la question...

Zone test Python pour exécuter du code de test, tenant compte du code de votre solution...

Soumettre votre réponse et lancer votre code de test...

[Q2] Somme d'entiers

Masquer l'énoncé

Un mathématicien se pose la question de savoir combien de séries d'entiers *consécutifs* strictement positifs existent, dont la somme est égale à un nombe entier donné *n*. Donc on considère des séries *i,i+1,i+2,...,i+k* où *i>0* et *i + i+1 + i+2 + ... + i+k = n*.

Par exemple, pour *n = 100*, la somme de la série *18,19,20,21,22* est *18 + 19 + 20 + 21 + 22 = 100*. Mais combien de telles séries existent dont la somme est égale à *n* ? Pour *n = 100*, il y en a *3* :

- *9, 10, 11, 12, 13, 14, 15, 16*
- *18, 19, 20, 21, 22*
- *100*

Implémentez une fonction `combien(n)` qui, pour un nombre entier strictement positif `n`, retourne un entier représentant le nombre de séries d'entiers consécutifs strictement positifs dont la somme est égale à `n`. Par exemple : `combien(100)` retourne *3*.

```
def combien(n) :
    """
    @pre:  n est un nombre entier > 0
    @post: retourne le nombre de series d'entiers consecutifs
           strictement positifs dont la somme est egale a n
    """
    # à compléter
```

Si nécessaire vous pouvez aussi définir des fonctions auxiliaires.

Votre réponse a passé les tests ! Votre note est de 100.0%. [Soumission #5fcccc886779dd1786d63188]

Bravo, votre code compile.Voici la sortie de votre méthode de test:

```
Réponse à la question = 3
```

Question 1: combien

```
1  def combien(n):    # Ne pas effacer cette ligne
2      """
3      @pre:  n est un nombre entier > 0
4      @post: retourne le nombre de series d'entiers consecutifs
5             strictement positifs dont la somme est egale a n
6      """
7      res = 1
8      for i in range(0, n//2+2):
9          tmp = i
10         acc = 0
11         while tmp <= n:
12             if tmp == n: res += 1
13             acc += 1
14             tmp += i + acc
15     return res
```

Question 2: Zone de test

Insérez votre code de test optionnel ci-dessous. Vous pouvez utiliser `print` pour tester votre programme. Votre code de test sera exécuté à la suite de la définition de votre méthode `combien` et d'éventuelles fonctions auxiliaires ci-dessus (il ne faut pas les recopier ici).

Exemple:

```
1  print("Réponse à la question = " + str(combien(100)))
2  # 3
3  # vos tests
```

Soumettre

>_

Y aura-t-il des exemples dans l'examen du résultat du code demandé?

Attention: la zone de test est actif pendant l'examen mais sera ignoré lors de la correction de l'examen…

Evaluation de l'examen

- **Durant** l'examen
  - soumettre exécute uniquement ***vos*** tests, pas les notres
  - 100% vert = le code est exécutable ≠ score de 100%

    Your answer passed the tests! Your score is 100.0%.
  - 0% rouge = votre code ne fonctionne pas : erreurs de Python

    There are some errors in your answer. Your score is 0.0%. => score de 0 %
- **Après** l'examen : évaluation avec ***nos*** tests
  - correction automatisée avec INGInious
  - pas de correction manuelle
  - une solution exécutable et partiellement correcte peut obtenir des points
  - une solution qui n'exécute pas donnera un score de 0%
  - Outil de détection de plagiat

Est ce que notre code sera encore corrigé manuellement?

Evaluation de l'examen

- La correction est complètement ***automatisé***
  (nos tests d'évaluation)
  - Vérifie le input-output de votre programme
  - Vérifie les cas de bords
  - Peut vérifier la structure interne de votre programme
  - (et même l'efficacité d'un programme)
  - Ne lit pas vos commentaires
    et donc pas vos spécifications pre-post
  - Vérifie surtout le bon comportement de votre solution

Evaluation de l'examen

- Vous recevrez **des points pour chaque partie correctement implémentée**.
  - Par conséquent, si vous **rencontrez des problèmes** pour gérer des cas particuliers, assurez-vous que **votre code fonctionne correctement dans les autres cas.**
- Chaque question comporte une zone de test. Des exemples sont donnés pour vous aider à évaluer votre code.
  - **Tester votre code avec ces exemples ne garantit pas la note maximale à la question**. Veillez donc à bien lire l'énoncé et les spécifications et à tester votre code sur d'autres exemples également;
  - **Votre code est exécuté mais pas coté pendant l'examen;**
  - **Vous pouvez soumettre autant de réponses que vous voulez;**
  - Seule votre **dernière soumission** pour chaque question sera évaluée .

Evaluation de l'examen

Dois-je vérifier les pré-conditions ?

- Le principe d'une précondition est que vous pouvez supposer qu'elle sera satisfaite quand on appelle la fonction/méthode
- Donc pas besoin de la vérifier explicitement dans votre fonction/méthode
- Par contre, bien vérifier dans vos *tests* que la postcondition est satisfaite

Est-ce qu'on pourra utiliser des modules ?

- Chaque solution est réalisable avec ce que vous avez vu au cours
- tout le langage Python reste évidemment à votre disposition
- Néanmoins, *dans aucune question il ne sera possible de faire des import de modules vous-même.*

Donc pas le module unittest non plus

La note du cours

```
def note(interro, examen, bonus):  # /20, /20, /1
    if examen > interro :
        note = examen + bonus
    else:
        note = (1/3 * interro) + (2/3 * examen) + bonus
    if note > 20 :
        note = 20
    return round(note)

>>> note(15, 6, 0)
9
>>> note(10, 15, 1)
16
>>> note(20, 20, 1)
20
```

Pourriez-vous expliquez comment la note finale sera calculée?

Matière à connaître

- Il n'y aura pas de questions QCM sur la théorie
- Possibilité de faire des tests en Python
  - code de test en Python
  - fonction help() de Python
- Il ne faut ***pas*** connaître par cœur toutes les fonctions de Python
  - Le syllabus théorie sera disponible
    (PDF et lien via INGInious)
- Quelle matière à (ne pas) connaître?

Matière – Partie 1

À connaître	Pas à connaître
Syntaxe de Python	
Expressions et **instructions**	
Opérateurs arithmétiques et **ordre**	
Variables et **affectation**	
Boucles : **for**, **while**, **range**()	Turtle graphics
Types de données : **int**, **float**, **bool**, **str**	
Conditionnelles et valeurs **booléennes**	
Fonctions : définition, paramètres, arguments, résultats	
Portée et **durée de vie** des variables	**Définition de modules**
Spécifications pré/post	Modules particuliers : random, time, os, …

Matière – Partie 2

À connaître		Pas à connaître
Strings	notation [i], [:i], [i:j], [i:]	format()
	len(), +, *	
	strip(), split(), upper(), lower()	find(), join ()
	>, <, >=, <=, ==, !=	
	for … in …	if … in, not in
Listes	notation [i], [:i], [i:j], [i:]	listes en compréhension
	len(), +, *	enumerate()
	append()	sort(), sorted(), extend()
	l.remove(3) ≠ del l[3]	insert(), count(), index(), reverse()
	for … in …	if … in, not in
String → liste	list()	
… → String	str()	
Tests	assert	if__name__== '__main__'

Matière – Partie 2

À connaître		Pas à connaître
Tuples	notation () et [i]	
Dictionnaires	notation { : } et [ ]	keys(), values()
	notation [], get(), in	setdefault()
	for … in, items()	
Références		copy ≠ deepcopy
Structures de données imbriquées et hétérogènes		
Structures muables vs. immuables		
Fichiers	open, close, with ... as	
	read, readline, readlines	
	write, writelines	
	for ... in file	
Exceptions	try... except, try ... except as	les différents types d'exceptions : IndexError, FileNotFoundError, …
	raise	

Matière – Partie 3

À connaître	Pas à connaître
is versus ==	
classes et objets	
attributs et méthodes d'instance	
création d'un objet	
appel d'une méthode	
appel à self (self.attribut, self.method())	
références	copy, deepcopy
égalité entre objets	
__init__, __str__, __eq__	les autres méthodes magiques

Matière – Partie 3

À connaître	Pas à connaître
attributs privés	méthodes privées
méthodes accesseurs et mutateurs	
attributs et méthodes de classe	
composition de classes	classes imbriquées
héritage	héritage multiple
super()	
polymorphisme	tests unitaires avec unittest
la liaison dynamique de self	
portée et visibilité des variables	

Matière – Structures et algorithmes

À connaître	Pas à connaître
Principes de comment un algorithme de recherche dichotomique fonctionne	Connaître par cœur le code d'un algorithme de recherche dichotomique
Comment utiliser une structure chainée	Connaître par cœur l'implémentation de la classe LinkedList
Comment implémenter une liste chaînée	

Informatique 1
Introduction à la programmation

Synthèse de la matière

Réponses aux questions

Kim Mens **Charles Pecheur** **Siegfried Nijssen**

La différence entre « is » et « == »

- Utilisez « x is y » pour vérifier l'**identité** :
  - Cela signifie que x et y font référence au **même objet en mémoire**.
  - C'est particulièrement utile dans les **structures de données chaînées**, où l'on souhaite savoir si deux références pointent vers **un même nœud**.
- Utilisez « x == y » pour vérifier l'**égalité** :
  - Vérifie si x et y ont des **valeurs équivalentes**, selon la méthode __eq__.
  - C'est pertinent lorsque vous souhaitez comparer le **contenu** des objets, par exemple, pour savoir si leurs attributs ont les mêmes valeurs.
- Pas de différence entre is et == si aucune méthode __eq__ n'a été redéfinie.
- Par contre si __eq__ a été redéfinie pour les objets à comparer, alors == repose sur l'égalité implémenté par cette méthode (typiquement pour comparer si les attributs de deux objets ont des **valeurs équivalentes**)

La différence entre « is » et « == »

```
# Utilisation de “is” :

class Node:
  def __init__(self,value):
    self.value = value
    self.next = None



node1 = Node(10)
node2 = Node(20)
other = Node(20)
node1.next = node2

#Comparaison d'identité
node1.next is node2 # True
node1 is node2      # False
node1 is Node(10)   # False
Node(5) is Node(5)  # False
node2 is other      # False
```

```
# Utilisation de “==” :

class Node:

  def __init__(self,value):
    self.value = value
    self.next = None

  def __eq__(self,other):
    if not isinstance(other, Node):
      return False
    return self.value == other.value

node1 = Node(10)
node2 = Node(20)
other = Node(20)

node1.next = node2

#Comparaison d'identité
node1.next == node2 # True
node1 == node2      # False
node1 == Node(10)   # True
Node(5) == Node(5)  # True
node2 == other      # True
```

Quelle solution à préférer?

```
##1##
if self.__first is self.__last and current.value() is cargo:
    …

##2##
if self.__first == self.__last and current.value() is cargo:
    …

##3##
if self.__first is self.__last and current.value() == cargo:
    …

##4##
if self.__first == self.__last and current.value() == cargo:
    …
```

La différence entre une méthode et une fonction

- Une **fonction** est un bloc de code indépendant qui peut être appelé n'importe où dans le programme.
  - Une fonction **n'appartient pas** à une classe.
  - Elle prend des paramètres explicites et n'a pas de référence implicite à un objet ou une classe (comme self ou cls).
- Une **méthode** est une fonction qui est **définie à l'intérieur d'une classe** et qui est donc associée à une classe ou à une instance d'une classe.
- **Méthodes d'instance** (les plus courantes) :
  - Prennent `self` comme premier paramètre pour accéder à l'instance actuelle.
- **Méthodes de classe** (`@classmethod`) :
  - Prennent `cls` comme premier paramètre pour accéder à la classe.

La différence entre (l'appel d') une méthode et une fonction

```
# Fonction :
def ajouter(x, y):
    return x + y


# Appel de la fonction,
# n'importe où dans le programme
print(ajouter(2, 3)) # Affiche 5
```

```
# Méthode d'instance
class Calculatrice:

    def ajouter(self, x, y):
        return x + y

# Appel de la méthode
calc = Calculatrice()
print(calc.ajouter(2, 3)) # Affiche 5
```

```
# Méthode statique
class Calculatrice:
    @staticmethod
    def ajouter(x, y):
        return x + y
# Appel direct via la classe
print(Calculatrice.ajouter(2, 3))
# Affiche 5
```

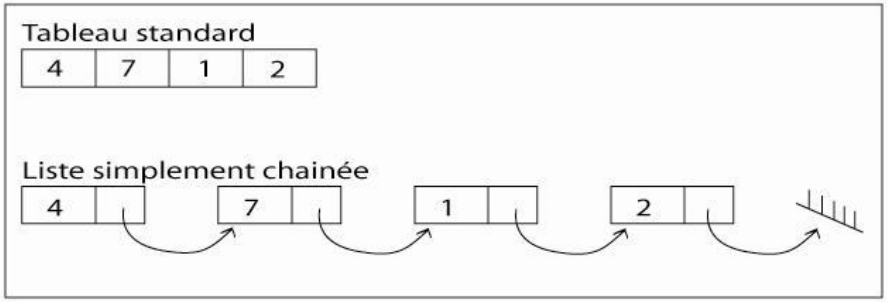
```
# Méthode de classe

class Calculatrice:

    @classmethod
    def ajouter(cls, x, y):
        return x + y

# Appel direct sur la classe
print(Calculatrice.ajouter(2, 3)) #
Affiche 5
```

Listes ou listes chaînées?



Quand utilise-t-on réellement les listes chaînées ?

- listes standards (list) en Python = tableaux dynamiques
  - accès direct à un élément à un index donné
  - optimisées pour les ajouts et suppressions en fin de liste
  - redimensionné dynamiquement en recopiant les données dans un nouveau tableau plus grande
- Listes chaînées
  - **Plus efficace si insertions et suppressions fréquentes**
    - n'importe où dans la liste en temps constant, à condition d'avoir une référence au nœud précedent
    - listes standards nécessitent de décaler les éléments lors de l'insertion ou la suppression d'éléments au milieu de la liste
  - **Implémentation de structures de données spécifiques**
    - Certaines structures de données spécifiques comme les files et les piles (stacks) peuvent être implémentées plus naturellement et plus efficacement avec des listes chaînées
  - Inconvénients:
    - utilisation plus importante de la mémoire due à l'espace supplémentaire nécessaire pour stocker les références (ou pointeurs)
    - plus grande complexité d'implementation
    - accès moins rapide aux éléments (besoin de parcourir la liste séquentiellement).

Dans la pratique, les listes standards de Python sont suffisantes pour la majorité des besoins, et les listes chaînées sont utilisées pour des besoins plus spécifiques.

Listes chaînées (Node)

```
class Node:
  def __init__(self,d,n):
    self.data = d
    self.next = n
```

```
l = None
print(l)
# None
l = Node(9,l)
print(l)
# <__main__.Node object at
0x1052ba8c0>
```

Listes chaînées (Node)

```
class Node:
  def __init__(self,d,n):
    self.data = d
    self.next = n
  def __str__(self):
    return str(self.data)
```

```
>>> l = None
>>> print(l)
None
>>> l = Node(9,l)
>>> print(l)
9
```

Listes chaînées (Node)

```
class Node:
  def __init__(self,d,n):
    self.data = d
    self.next = n
  def __str__(self):
    return str(self.data)
```

```
l = None
for i in range(9,0,-1) :
    l = Node(i,l)
node = l
while node is not None :
    print(node,end="")
    node = node.next
# 123456789
```

Listes chaînées (Node)

```
class Node:
  def __init__(self,d,n):
    self.data = d
    self.next = n
  def __str__(self):
    return str(self.data)
```

```
class LinkedList :
  def __init__(self):
    self.head = None
  def add(self, cargo):
    node = Node(cargo,self.head)
    self.head = node
```

```
l = LinkedList()
for i in range(9,0,-1):
    l.add(i)
node = l.head
while node is not None:
    print(node,end="")
    node = node.next
# 123456789
```

Listes chaînées (Node)

```
class Node:
  def __init__(self,d,n):
    self.data = d
    self.next = n
  def __str__(self):
    return str(self.data)
```

```
l = LinkedList()
for i in range(9,0,-1):
    l.add(i)
print(l)
# 123456789
```

```
class LinkedList :
  def __init__(self):
    self. head = None
  def add(self, cargo):
    node = Node(cargo,self.head)
    self.head = node
  def __str__(self):
    s = ""
    node = self.head
    while node is not None :
      s += str(node)
      node = node.next
    return s
```

Listes chaînées (Node)

```
class Node:
  def __init__(self,d,n):
    self.data = d
    self.next = n
  def __str__(self):
    return str(self.data)
```

```
l = LinkedList()
for i in range(9,0,-1):
    l.add(i)
print(l)
# 123456789
l.remove(3)
print(l)
# 12456789
l.remove(1)
print(l)
# 2456789
```

```
class LinkedList :
  def __init__(self):
    self. head = None
  ...
 def remove(self,cargo):
    node = self.head
    if node is None :
        return False
    if node.data == cargo :
        self.head = node.next
        return True
    while node.next is not None :
      if node.next.data == cargo:
        node.next = node.next.next
        return True
      node = node.next
    return False
```


FIN
RESTEZ
CALME
ET
BONNE
CHANCE
KeepCalmAndPosters.com