
InitProg Documentation

Release 2

Raphael Marvie, Martin Monperrus

September 03, 2013

CONTENTS

1	Introduction à Python par les cartes	3
1.1	Cartes.py	3
1.2	Python en mode interactif	3
1.3	Python en mode programme	3
1.4	Instructions	4
1.5	Modules	4
1.6	Commentaires	5
1.7	Fonctions	5
1.8	Boucles	6
1.9	Instructions conditionnelles	7
1.10	Expressions booléennes	7
2	Hors les cartes	9
2.1	Variables	9
2.2	Arithmétique	9
2.3	Typage	10
2.4	Chaînes de caractères	10
2.5	Boucles (le retour)	12
3	Manuel de Cartes.py	17
3.1	Installation du module Cartes	17
3.2	API	17
4	Exercices Cartes	21
4.1	Descriptions de tas	21
4.2	Séquence	21
4.3	Conditionnelle	22
4.4	Itération	23
4.5	Fonctions	25
5	Environnement de programmation	27

Python est un langage de programmation développé par Guido Von Rossum au CWI, à l'Université d'Amsterdam et nommé par rapport au *Monthy Python's Flying Circus*. Son ancêtre est le langage de script du système d'exploitation *Amoeba* (1990) et *comp.lang.python* a été créé en 1994. Python offre un interpréteur performant et de nombreux modules. Il est disponible sur la grande majorité des plates-formes courantes (BeOS, Mac OS X, Unix, Windows).

Python est un langage *open source* supporté, développé et utilisé par une large communauté: 300 000 utilisateurs et plus de 500 000 téléchargements par an.

Beaucoup d'entreprises et d'universités utilisent Python, comme par exemple Google.

Ces notes de cours sont faites à partir de différentes sources pédagogiques de l'Université de Lille1. Nous remercions chaleureusement leurs auteurs.

INTRODUCTION À PYTHON PAR LES CARTES

1.1 Cartes.py

Le module Cartes permet d'écrire des programmes destinés à un robot manipulateur de cartes. Le domaine de travail du robot est constitué de tas, numérotés 1,2,3,4, sur lesquels sont empilées des cartes à jouer. Ces cartes ont les couleurs habituelles (♣, ♦, ♥ et ♠) et les valeurs habituelles (par ordre croissant : as, deux, trois, . . . , dix, valet, dame, roi). Les cartes proviennent de plusieurs jeux, il est donc possible de trouver plusieurs exemplaires d'une même carte. Chacun des quatre tas peut contenir un nombre quelconque de cartes, y compris aucune. Le module Cartes est un module spécifiquement développé à l'université de Lille 1 pour l'enseignement d'InitProg.

1.2 Python en mode interactif

Python, comme la majorité des langages dits de script, peut être utilisé aussi bien en mode interactif qu'en mode programme.

Dans le premier cas, il y a un dialogue entre l'utilisateur et l'interprète, c'est une boucle REPL (*Read-Eval-Print-Loop*, soit Boucle Lire, Evaluer, Afficher). Les commandes entrées par l'utilisateur sont évaluées au fur et à mesure. Cette première solution est pratique pour réaliser des prototypes, ainsi que pour tester tout ou partie d'un programme ou plus simplement pour interagir aisément et rapidement avec des structures de données complexes. Le listing suivant présente comment lancer l'interprète (simplement en tapant `python` à l'invite du *shell*) et comment en sortir (en tapant `Ctrl-D`).

```
$ python
Python 2.7.2 (v2.7.2:8527427914a2, Jun 11 2011, 15:22:34)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from Cartes import *
>>> init_tas(1, "PCKT")
>>> deplacer_sommet(1,2)
>>> ^D
$
```

1.3 Python en mode programme

Pour une utilisation en mode *programme* les instructions à évaluer par l'interprète sont sauvegardées, comme n'importe quel programme informatique, dans un fichier. Dans ce second cas, l'utilisateur doit saisir l'intégralité des instructions

qu'il souhaite voir évaluer à l'aide de son éditeur de texte favori, puis demander leur exécution à l'interprète. Les fichiers Python sont identifiés par l'extension `.py`. Le listing suivant, l'interprète est appelé avec en paramètre le nom du programme à évaluer. Dans le cas d'un système Unix, la première ligne du fichier précise quel interprète utiliser pour évaluer le fichier si ce dernier est exécutable. Il suffit dans ce cas de taper le nom du fichier à l'invite du *shell*.

```
$ cat tp.py
#! /usr/bin/env python
from Cartes import *
init_tas(1, "PCKT")
deplacer_sommet(1, 2)
$ python tp.py
```

1.4 Instructions

Une instruction est un ordre donné à l'ordinateur. Un programme une séquence d'instructions, comme par exemple:

```
>>> from Cartes import *
>>> init_tas(1, "PCKT")
>>> deplacer_sommet(1, 2)
```

Ce programme met de bas en haut un Pique (P), surmonté d'un Coeur (C), puis d'un Carreau (K) et un Trèfle (T) sur le premier des quatres tas du robot manipulateur de cartes. Il y a plusieurs sortes d'instructions. Dans le programme ci-dessus, le `from Cartes import *` est un `import` (nécessaire pour tous les programmes "Cartes"), `init_tas` et `deplacer_sommet` sont des fonctions, `init_tas(1, "PCKT")` et `deplacer_sommet(1, 2)` sont des appels de fonctions.

1.5 Modules

Un module permet de fournir des des fonctions à intégrer dans les programmes. Par exemple, le module fournit les fonctions suivantes: `init_tas`, `deplacer_sommet`, `couleur_sommet`, `sommet_trefle`, `sommet_carreau`, `sommet_coeur`, `sommet_pique`, `superieur`, `tas_vide`, `tas_non_vide`.

La construction `import` permet d'importer un module et de fournir accès à son contenu. L'importation d'un module peut se faire de deux manière. La première solution est de désigner le module que l'on souhaite utiliser, son contenu est alors utilisable de manière «scopée», c'est-à-dire en préfixant le nom de la fonctionnalité du nom du module.

```
>>> import Cartes
>>> Cartes.init_tas(1, 'C')
```

La seconde solution repose sur la construction `from import` où l'on identifie la ou les fonctionnalités que l'on souhaite importer d'un module. Dans le cas d'un import de plusieurs fonctionnalités, les noms sont séparés par des virgules. L'utilisation se fait alors sans préfixer par le nom de module («non scopé»).

```
>>> from Cartes import init_tas
>>> init_tas(1, 'C')
```

Enfin, il est possible d'importer, avec cette seconde approche, tous les éléments d'un module en utilisant la notation `*`. **Attention**, avec cette dernière forme car il y a pollution du *namespace* (espace de noms) global: deux imports consécutifs peuvent charger deux définitions d'une même fonction. La seconde masquera la première.

```
>>> from Cartes import *
>>> deplacer_sommet(1, 2)
```


1.6 Commentaires

Comme dans la majorité des langages de script, les commentaires Python sont définis à l'aide du caractère #. Qu'il soit utilisé comme premier caractère ou non, le # introduit un commentaire jusqu'à la fin de la ligne. Les commentaires sont à utiliser abondamment avec intelligence. Il ne faut pas hésiter à commenter le code, sans pour autant mettre des commentaires qui n'apportent rien. Le listing suivant présente une ligne de commentaire en Python.

```
>>> # solution du tp 1
>>> init_tas(1, "P+C")
```

1.7 Fonctions

Une fonction est un ensemble d'instructions paramétré. Le programme suivant initialise tous les tas comme vide en définissant la fonction `initialiser_a_vide`.

```
>>> def initialiser_a_vide(numero_tas):
...     init_tas(numero_tas, "")
>>> initialiser_a_vide(1)
>>> initialiser_a_vide(2)
>>> initialiser_a_vide(3)
>>> initialiser_a_vide(4)
```

`numero_tas` est un *paramètre* de la fonction `initialiser_a_vide`.

Spécifier une fonction, c'est :

- choisir un identificateur pour la nommer (en général, commence par un verbe) ;
- préciser ses paramètres en les nommant et en donnant leur rôle ;
- indiquer les conditions d'utilisation (CU) que doivent vérifier les paramètres lors d'un appel à la fonction ;
- et indiquer s'il y a ou non une valeur renvoyée. La cas échéant, indiquer quelle est la relation entre la valeur renvoyée et les paramètres qui lui sont passés.

1.7.1 Note sur l'indentation

La structuration d'un programme Python est définie par son indentation. Le début d'un bloc est défini par un ':', la première ligne pouvant être considérée comme un en-tête (test, boucle, définition, etc.). Le corps du bloc est alors indenté de manière plus importante (mais régulière) que l'en-tête. Enfin, la fin du bloc est délimité par le retour à l'indentation de l'en-tête. La convention en Python est d'utiliser quatre espaces pour chaque niveau d'indentation. Les bloc peuvent être imbriqués.

```
>>> def initialiser_a_vide(numero_tas):
...     init_tas(numero_tas, "")
```

Cette structuration est utilisée aussi bien pour définir des fonctions, des boucles, des tests.

1.7.2 Spécification

La définition des commentaires avec # devrait être réservé aux remarques techniques sur le code et pour les développeur de l'élément en question. La spécification d'une fonction se fait sur la première ligne après la déclaration en utilisant des chaînes de caractères multi-lignes, même si la documentation tient sur une seule, en utilisant des guillemets doubles.

Une specification doit préciser :

- l'action de la fonction
- la fonction des paramètres
- les contraintes d'utilisation (CU) que doivent vérifier les paramètres lors d'un appel à la fonction

```
>>> def passe_en_minuscule(chaine):  
...     """Retourne la chaine de caractere en minuscules.  
...  
...     Contraintes d'utilisation: la chaine ne doit pas etre vide.  
...     """  
  
>>> help(passe_en_minuscule)  
Help on function passe_en_minuscule:  
  
passe_en_minuscule(chaine)  
    Retourne la chaine de caractere en minuscules.  
  
    Contraintes d'utilisation: la chaine ne doit pas etre vide.
```

1.8 Boucles

La boucle permet d'écrire des instructions conditionnellement répétées. Une boucle *tant que* se termine lorsque sa condition n'est plus satisfaite. Par conséquent à la sortie de la boucle, on est assuré que la condition est fausse.

```
>>> init_tas(1, "C[P]")  
>>> while tas_non_vide(1):  
...     deplacer_sommet(1,2)
```

Lorsqu'elle sera exécutée, cette instruction testera si le tas 1 est non vide. Si ce n'est pas le cas (c'est-à-dire si le tas 1 est vide) alors l'instruction est terminée, sinon (si le tas 1 n'est pas vide), une carte sera déplacée du tas numéro 1 vers le tas numéro 2. Ce déplacement effectué, la vacuité du tas numéro 1 sera à nouveau testée, et selon le cas l'instruction est terminée ou un déplacement est effectué. Et on poursuit ainsi *tant que* le tas 1 n'est pas vide.

La conception d'une boucle *tant que* nécessite plusieurs points d'attention :

Premièrement, la situation avant d'entrer dans la boucle (pré-condition) est-elle celle souhaitée? Par exemple, l'instruction qui suit peut être source d'erreur.

```
>>> init_tas(1, "P[T]")  
>>> while sommet_trefle(1):  
...     deplacer_sommet(1,2)
```

En effet, la condition du *tant que* porte sur la couleur de la carte au sommet du tas 1. Pour cela on utilise la fonction `sommet_trefle`. Mais pour être utilisée, celle-ci requiert que le tas numéro 1 ne soit pas vide.

Ensuite, la situation à la sortie de la boucle (post-condition) est-elle bien celle souhaitée ? la boucle ne risque-t-elle pas d'être infinie (problème de l'arrêt) ? Par exemple, l'instruction qui peut conduire à une répétition infinie de déplacement d'une carte du tas 1 vers le tas 2, puis d'un retour de la carte déplacée vers son tas d'origine. En effet si le tas 1 n'est pas vide au moment d'exécuter cette instruction, la condition du *tant que* est satisfaite et l'action est exécutée, action qui aboutit à la même situation que celle de départ : le tas 1 ne sera jamais vide.

```
>>> while tas_non_vide(1):  
...     deplacer_sommet(1,2)  
...     deplacer_sommet(2,1)
```

1.9 Instructions conditionnelles

L'instruction conditionnelle permet de mener une action si une condition est satisfaite, et une autre dans le cas contraire.

```
>>> init_tas(1, "KP+T")

>>> if sommet_trefle(1):
...     deplacer_sommet(1,2)
... else:
...     deplacer_sommet(1,3)
```

Les instructions conditionnelles du module Cartes sont: couleur_sommet, sommet_trefle, sommet_carreau, sommet_coeur, sommet_pique, superieur, tas_vide, tas_non_vide.

L'action exécutée peut comprendre plusieurs instructions (en suivant les règles d'indentation).

```
>>> if sommet_trefle(1):
...     deplacer_sommet(1,2)
...     deplacer_sommet(2,3)
```

Il est possible d'enchaîner plusieurs instructions conditionnelles avec `elif`:

```
>>> init_tas(1, "C")
>>> if sommet_trefle(1):
...     deplacer_sommet(1,2)
... elif sommet_carreau(1):
...     deplacer_sommet(1,3)
... elif sommet_coeur(1):
...     deplacer_sommet(1,4)
```

1.10 Expressions booléennes

Les expressions booléennes permettent d'écrire les conditions des instructions `if` et `while`.

1.10.1 Expressions simples

Les expressions booléennes simples “la carte au sommet du tas 1 est un trèfle”.

```
>>> init_tas(1, "T")
>>> if sommet_trefle(1):
...     deplacer_sommet(1,4)
```

1.10.2 Expressions composées

Les expressions composées utilisent les opérateurs OU, ET et NON suivant les règles classiques de l'algèbre booléennes (cf. [Algèbre de Boole sur Wikipedia](#)).

```
>>> while tas_non_vide(1) and sommet_trefle(1):
...     deplacer_sommet(1,2)

>>> if tas_non_vide(1) and (sommet_coeur(1) or sommet_carreau(1)):
...     deplacer_sommet(1,2)
```

```
>>> init_tas(1, "C")
>>> if not sommet_pique(1):
...     deplacer_sommet(1, 2)
```

HORS LES CARTES

2.1 Variables

Les variables servent à nommer des valeurs. Elles sont caractérisées par leur nom ou identificateur, leur type et leur valeur. Les variables sont nommées par un identificateur. Les identificateurs de variables doivent obligatoirement commencer par une lettre ou un blanc souligné (`_`) et peuvent être complétés par n'importe quelle quantité de lettres minuscules ou majuscules non accentuées, ou chiffres, ou blancs soulignés.

```
>>> x = 1 + 2
>>> y = 'oi'
>>> z = True
```

Le contenu des variables peut servir d'instruction conditionnelle.

```
>>> if x == 'hello':
...     print 'hello too!'

>>> while y == 3:
...     print 'toujours 3'

>>> while z > 4.5:
...     print 'pas fini'
```

2.2 Arithmétique

Python permet d'exprimer très simplement des opérations arithmétiques. Dans le cas où tous les opérandes sont des entiers, alors les résultats seront aussi des entiers. Lorsqu'au moins un des opérandes est de type réel, alors tous les opérandes sont automatiquement convertis en réels.

```
>>> x = 1 + 2
>>> y = 5 * 2

>>> y / x
3

>>> y % x  # modulo : le reste dans la division entière de de y par x
1

>>> y = 5.5 * 2
>>> y
11.0
```

```
>>> x = 12.0 / 3
>>> x
4.0
```

Dans le contexte de l'arithmétique, les affectations peuvent prendre deux formes. Ces deux formes ont un sens différent. Le listing suivant présente deux affectations qui pourraient être comprises de manière identique. Toutefois, la première forme (ligne 2) a pour conséquence de créer une nouvelle instance d'entier pour contenir l'ajout de 2 à la valeur de `x`. La seconde forme (ligne 3) ajoute 2 à la valeur de `x` sans créer de nouvelle instance. La manière d'écrire une opération a donc un impact sur son évaluation.

```
>>> x = 4
>>> x = x + 2
>>> x += 2
```

2.3 Typage

Toutes les variables et les paramètres de fonctions ont un type dynamique, par exemple :

- les entiers (`int`)
- les réels (`float`)
- les booléens (`bool`)
- les chaînes de caractères (`str`)

```
>>> type(0)
<type 'int'>

>>> type(1.2)
<type 'float'>

>>> type(True)
<type 'bool'>

>>> type('ml')
<type 'str'>
```

Dans la spécification, il est d'usage de préciser le type des paramètres et de la valeur retournée :

```
>>> def est_pair(x):
...     """renvoie un booléen True si l'entier x est pair, False sinon"""
...     return (x%2) == 0 # % est le modulo
...
>>> est_pair(56)
True
>>> est_pair(197)
False
```

2.4 Chaînes de caractères

Les chaînes de caractères se définissent de plusieurs manières en Python. Il est possible d'utiliser indifféremment des guillemets simples ou des guillemets doubles. Le choix est souvent imposé par le contenu de la chaîne: une chaîne contenant des guillemets simples sera déclarée avec des guillemets doubles et réciproquement. Pour les autres cas, c'est indifférent. Enfin, comme tout est objet en Python une chaîne est donc un objet. Le listing suivant déclare deux

chaînes référencées par `x` et `y`. Enfin, la chaîne référencée par `z` est une chaîne de caractères multi-lignes (utilisation de trois quotes / guillemets simples ou doubles).

```
>>> x = 'hello '
>>> y = "world!"
>>> z = '''hello
... world'''
```

2.4.1 Concaténation

La concaténation de ces chaînes de caractères peut prendre deux formes. Dans les deux cas, l'opérateur `+` est utilisé pour exprimer la concaténation. La forme de la ligne 4 est un raccourci d'écriture.

```
>>> z = x + y
>>> z
'hello world!'
>>> x += y
>>> x
'hello world!'
```

2.4.2 Affichage

L'affichage de chaînes de caractères à l'aide de la fonction `print` peut se faire en concaténant explicitement des chaînes (que ce soit avec l'opérateur de concaténation ou en utilisant des virgules) ou en utilisant une chaîne de formatage comme la fonction `printf` du langage C. Cette seconde option est un peu plus puissante, mais aussi un peu plus lourde à utiliser. Le listing suivant présente trois manières d'afficher des chaînes de caractères.

```
>>> print 'I say: ' + x
I say: hello world!

>>> print x, 2, 'times'
hello world! 2 times

>>> print "I say: %s %d time(s)" % (x, 2)
I say: hello world! 2 time(s)
```

La longueur d'un chaîne s'obtient avec la fonction `len`:

```
>>> len('Hello')
5
```

2.4.3 Manipulations

Python offre une méthode simple pour accéder aux caractères contenus dans une chaîne: une chaîne est manipulée comme une séquence indexée de caractères. Ainsi, chaque caractère est accessible directement par son index (le premier étant indexé 0) en utilisant des crochets. En plus de cet accès unitaire aux caractères, il est possible d'accéder à des sous-chaînes en précisant la tranche souhaitée l'index de début (qui est inclus) étant séparé de l'index de fin (qui est exclu) par le caractère `:`. Si l'index de début est manquant, python considère que le début est 0, si l'index de fin est manquant la longueur de la chaîne est prise.

```
>>> x = 'hello world!'
>>> x[:5] == x[0:5]
True
>>> x[3:] == x[3:len(x)]
True
```

Dans le cas des sous-chaînes, la valeur fournie est une copie et non un accès à une partie de la chaîne d'origine.

Le listing suivant donne quelques exemples d'accès à un caractère (ligne 2), ou à une sous-chaîne pour le reste. La ligne 6 signifie que l'on souhaite le contenu de `x` du quatrième caractère (les indices commencent à 0) à la fin.

```
>>> x = 'hello world!'

>>> x[4]
'o'

>>> x[2:4]
'll'

>>> x[3:]
'lo world!'

>>> x[:]
'hello world!'

>>> # enlever le deuxième 'l'
>>> x= "hello!"
>>> x[:3]+x[4:]
'helo!'

>>> # insérer un > entre les deux l 'l'
>>> x= "hello!"
>>> x[:3]+"Z"+x[3:]
'helZlo!'

>>> # remplacer le deuxième 'l' par P
>>> x= "hello!"
>>> x[:3]+"P"+x[4:]
'helPo!'
```

Enfin, un index négatif précise que le calcul s'effectue depuis la fin de la chaîne.

```
>>> x[-3:]
'lo!'

>>> x[1:-1]
'ello'
```

2.5 Boucles (le retour)

Deux types de boucles sont disponibles: les boucles énumérées (`for`) et les boucles basées sur un test de fin (`while`) (que nous avons utilisée avec les cartes). Ces deux constructions suivent le même schéma: un en-tête qui décrit l'évolution de la boucle, un ensemble d'instructions qui sont évaluées à chaque tour de boucle.

2.5.1 Boucles énumérées

Une boucle `for` définit une variable qui prend successivement toutes les valeurs de la séquence (liste ou tuple) parcourue.

```
>>> for i in range(3):
...     print i
0
```


1
2

La fonction `range` produit une liste de tous les entiers entre une borne inférieure et une borne supérieure. Cette construction est utile lorsqu'une boucle `for` repose sur l'utilisation d'une séquence d'entiers. La fonction `range` est utilisable de trois manières:

Un seul paramètre spécifiant le nombre d'éléments.

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
```

Deux paramètres spécifiant la borne inférieure (inclue) et supérieure (exclue).

```
>>> range(3, 7)
[3, 4, 5, 6]
```

Trois paramètres spécifiant les bornes et le saut (incrément entre deux éléments de la séquence).

```
>>> range(0, 10, 3)
[0, 3, 6, 9]
```

Boucle `for` parcourant une séquence de chaîne.

```
>>> a = ['hello', 'world']
>>> for elt in a:
...     print elt
...
hello
world
```

Enfin, une chaîne étant une séquence de lettres, elle peut être parcourue comme une séquence.

```
>>> for lettre in 'bar':
...     print lettre
...
b
a
r
```

2.5.2 Boucles conditionnelles

Une boucle `while` définit une condition booléenne avec les mêmes règles que pour les tests (ligne 1). Tant que cette condition est respectée, les instructions du bloc associé au `while` sont évaluées.

```
>>> n = 1
>>> while n < 100:
...     n = n + n
>>> print n
128
```

La boucle suivante représente la boucle minimale définissable avec un `while`. Certes, elle est stupide mais elle permet d'illustrer l'utilisation du `Ctrl-C` pour interrompre un traitement en cours d'exécution.

```
>>> while True:
...     pass
...
KeyboardInterrupt
```

2.5.3 Listes

Le contenu de cette section est hors du programme d'InitProg, mais peut vous servir par la suite.

Les *listes* Python sont des ensembles ordonnés et dynamiques d'éléments. Ces ensembles peuvent contenir des éléments de différents types.

L'exemple suivant crée tout d'abord deux listes vides avec les deux manières possibles.

```
>>> mylist = []
>>> mylist2 = list()
```

Ensuite, après avoir défini `x`, une liste contenant la chaîne `'bar'`, l'entier `12345` et l'objet référencé par la variable `x` est créée.

```
>>> x = True
>>> foo = ['bar', 12345, x]
>>> foo
['bar', 12345, True]
```

Les listes et les chaînes ont pour point commun le fait d'être des ensembles ordonnés. L'accès à un élément d'une liste se fait donc aussi par indexation.

```
>>> foo[2]
True
```

Il est possible de prendre une partie de liste et d'obtenir une copie de liste. Une copie de liste crée une nouvelle liste, mais partage les éléments contenus dans la liste d'origine.

```
>>> foo[1:]
[12345, True]
>>> bar = foo[:]
```

Cette dernière manipulation est très importante lors de parcours de listes pour ne pas modifier la liste qui est parcourue et se retrouver dans une boucle sans fin.

Enfin, le contenu de la liste peut être changé par simple affectation en utilisant l'index cible. Comme la liste référencée par `bar` est une copie de la liste référencée par `foo`, le dernier élément de `bar` n'est pas modifié par l'affectation du dernier élément de `foo`.

```
>>> foo[2] = 1
>>> foo
['bar', 12345, 1]
>>> bar[-1]
True
```

L'ajout d'éléments dans une liste se fait à l'aide des méthodes `append` pour un ajout en fin de liste, et `insert`, pour un ajout à un index donné. Enfin, la méthode `extend` ajoute le contenu d'une liste passé en paramètre à la fin de la liste.

```
>>> foo.append('new')
>>> foo
['bar', 12345, 1, 'new']

>>> foo.insert(2, 'new')
>>> foo
['bar', 12345, 'new', 1, 'new']

>>> foo.extend([67, 89])
>>> foo
['bar', 12345, 'new', 1, 'new', 67, 89]
```

La méthode `index` permet de connaître l'index de la première occurrence d'un élément dans une liste. Dans le cas où l'élément fourni en paramètre n'est pas présent, la méthode lève l'exception `ValueError`. L'utilisation de la construction `in` retourne quant à elle `True` si l'élément est présent dans la liste et `False` sinon.

```
>>> foo.index('new')
2
>>> foo.index(34)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>> 34 in foo
False
```

Les listes peuvent être fusionnées par concaténation. Ces concaténations peuvent aussi bien se faire par copie que par ajout. Une concaténation par copie des deux listes existantes:

```
>>> bar = [0, 1] + [1, 0]
>>> bar
[0, 1, 1, 0]
```

Une concaténation par ajout d'éléments à une liste existante:

```
>>> bar += [2, 3]
>>> bar
[0, 1, 1, 0, 2, 3]
```

Enfin, une manière simple de créer une liste par répétition d'un motif qui doit être lui même une liste:

```
>>> [0, 1] * 3
[0, 1, 0, 1, 0, 1]
```

Attention Dans le cas d'une création de liste par répétition, les éléments ne sont pas dupliqués, mais ce sont leurs références qui sont répétées (et donc présentes plusieurs fois dans la liste finale). Utiliser l'opérateur `*` pour la construction de matrice est donc une mauvaise idée car toutes les lignes seraient alors la même liste référencée plusieurs fois.

```
>>> matrix = [[1, 2, 3]] * 3
>>> matrix
[[1, 2, 3], [1, 2, 3], [1, 2, 3]]

>>> matrix[1][1] = 4
>>> matrix
[[1, 4, 3], [1, 4, 3], [1, 4, 3]]
```

De manière symétrique à l'ajout d'éléments dans une liste, il est possible d'en supprimer. La méthode `remove` permet de supprimer la première occurrence d'un élément d'une liste en le désignant.

```
>>> foo.remove('new')
>>> foo
['bar', 12345, 1, 'new', 67, 89]
```

Si l'élément fourni en paramètre n'existe pas dans la liste, l'exception `ValueError` est levée.

```
>>> foo.remove(34)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

L'opérateur `del` (*delete*) permet de détruire une référence à un objet Python, ou à une partie d'une liste.

```
>>> del foo[1:3]
>>> foo
['bar', 'new', 67, 89]
```

Listes et chaînes de caractères

Les listes et les chaînes de caractères sont similaires dans leur structure et dans leur manipulation. Certaines méthodes disponibles sur les chaînes manipulent les deux structures de données.

La méthode `join` disponible sur les chaînes permet de construire une chaîne de caractère depuis une liste de chaînes. La chaîne sur laquelle est invoquée la méthode `join` est alors utilisée comme séparateur des différents éléments de la liste.

```
>>> ' ; '.join(['a', 'b', 'c'])
'a ; b ; c'
```

De manière symétrique, la méthode `split`, disponible sur les chaînes, permet de décomposer une chaîne de caractères en une liste de sous chaînes. Ce découpage se fait par rapport à un ou plusieurs caractères. Dans le cas où un entier est passé comme second argument, il précise le nombre maximum de découpage.

```
>>> 'hello crazy world!'.split(" ")
['hello', 'crazy', 'world!']

>>> 'hello crazy world!'.split(" ", 1)
['hello', 'crazy world!']
```

MANUEL DE CARTES.PY

3.1 Installation du module Cartes

3.1.1 En salle de TP dans un terminal

```
$ export HTTP_PROXY=http://cacheserv.univ-lille1.fr:3128
$ mkdir -p .local/lib/python2.6/site-packages
$ easy_install --install-dir ~/.local/lib/python2.6/site-packages CartesInitProg
$ wget http://www.monperrus.net/martin/exemple1.py
$ python exemple1.py
```

3.1.2 Sous Linux dans un terminal

```
$ sudo apt-get install python-setuptools
$ # ou sudo apt-get install easy_install
$ easy_install CartesInitProg
$ wget http://www.monperrus.net/martin/exemple1.py
$ python exemple1.py
```

3.1.3 Sous Windows

Voir Moodle.

3.1.4 Sous Mac

Voir Moodle.

3.2 API

3.2.1 `init_tas(n, s)`

La fonction `init_tas(n, s)` initialise le tas `n` avec la chaîne de description `s` (en Python, les chaînes sont délimitées par des guillemets ”)

Tout programme de résolution d’un problème sur les cartes doit débiter par une initialisation des quatre tas à l’aide de l’instruction `init_tas`. Une fois cette initialisation effectuée, cette instruction n’est plus utilisée.

La chaîne de description, lue de gauche à droite, indique les cartes d'un tas du bas vers le haut avec les conventions suivantes : Les lettres T, K, C, P représentent respectivement : ♣, ♦, ♥, ♠ ; Si A et B sont des chaînes de description, la chaîne "AB" est aussi une chaîne de description qui représente les cartes de A surmontées de celles de B. Si A et B sont des chaînes de description, la chaîne A+B est aussi une chaîne de description qui représente un choix entre les cartes de A ou celles de B ; Si A est une chaîne de description, [A] représente la répétition des cartes de A un nombre quelconque (défini de manière aléatoire) de fois (y compris zéro). Les parenthèses sont autorisées pour éviter les ambiguïtés.

```
>>> init_tas(1, "PCKT")
>>> init_tas(2, "")
>>> init_tas(3, "[P]")
>>> init_tas(4, "T+C")
```

Exemples:

- `init_tas(1, "")` déclare le tas numéro 1 vide
- l'instruction `init_tas(2, "TCK")` décrit le tas numéro 2 contenant de bas en haut un ♣, un ♥ et un ♦
- l'instruction `init_tas(3, "T+P")` décrit le tas numéro 3 contenant une carte qui est soit un ♣ soit un ♠
- `init_tas(4, "[T+P]")` décrit le tas numéro 4 contenant un nombre quelconque, non déterminé, de cartes de couleur ♣ ou ♠
- l'instruction `init_tas(1, "T+[P]CC")` décrit le tas numéro 1 contenant soit une seule carte de couleur ♣, soit un nombre quelconque de ♠ surmonté de deux ♥
- l'instruction `init_tas(1, "(T+[P])CC")` décrit le tas numéro 1 contenant soit trois cartes un ♣ surmonté de deux ♥, soit un nombre quelconque de ♠ surmonté de deux ♥. À noter qu'une autre façon de décrire la même initialisation est d'utiliser la chaîne "TCC+[P]CC".

3.2.2 `deplacer_sommet(n1, n2)`

La fonction `deplacer_sommet(n1, n2)` modifie l'état des tas par le déplacement de la carte située au sommet du tas `n1` vers le sommet du tas `n2`. CU : Il n'est pas permis de déplacer une carte située sur un tas vide. Toute tentative d'action `deplacer_sommet(n, p)` déclenche une exception `Tas_Vide` si le tas `n` est vide.

```
>>> init_tas(1, "PCKT")
>>> deplacer_sommet(1, 2)
```

3.2.3 `tas_vide(n)`

La fonction `tas_vide(n)` teste si le tas `n` est vide, et renvoie une valeur booléenne (`True` si le tas est vide, `False` sinon).

```
>>> init_tas(1, "")
>>> if tas_vide(1):
...     print "le tas 1 est vide"
le tas 1 est vide
```

3.2.4 `tas_non_vide(n)`

La fonction `tas_non_vide(n)` teste si le tas `n` n'est pas vide, et renvoie une valeur booléenne (`True` si le tas contient au moins une carte, `False` sinon). Noter que `tas__non_vide(n)` est équivalent à `not tas_vide(n)`

```
>>> init_tas(1,"T")
>>> if tas_non_vide(1):
...     print "le tas 1 n'est pas vide"
le tas 1 n'est pas vide
```

3.2.5 sommet_trefle(n)

La fonction `sommet_trefle(n)` teste si la carte au sommet du tas `n` est un trèfle, et renvoie une valeur booléenne (True si c'est un trèfle, False sinon). Contrainte d'utilisation, le tas `n` doit contenir au moins une carte, sinon une exception est lancée.

```
>>> init_tas(1,"T")
>>> if sommet_trefle(1):
...     print "le sommet du tas 1 est un trefle"
le sommet du tas 1 est un trefle
```

3.2.6 sommet_pique(n)

La fonction `sommet_pique(n)` teste si la carte au sommet du tas `n` est un pique, et renvoie une valeur booléenne (True si c'est un pique, False sinon). Contrainte d'utilisation, le tas `n` doit contenir au moins une carte, sinon une exception est lancée.

```
>>> init_tas(1,"T")
>>> if sommet_pique(1):
...     print "le sommet du tas 1 est un pique"
```

3.2.7 sommet_coeur(n)

La fonction `sommet_coeur(n)` teste si la carte au sommet du tas `n` est un coeur, et renvoie une valeur booléenne (True si c'est un coeur, False sinon). Contrainte d'utilisation, le tas `n` doit contenir au moins une carte, sinon une exception est lancée.

```
>>> init_tas(1,"T")
>>> if sommet_coeur(1):
...     print "le sommet du tas 1 est un coeur"
```

3.2.8 sommet_carreau(n)

La fonction `sommet_carreau(n)` teste si la carte au sommet du tas `n` est un carreau, et renvoie une valeur booléenne (True si c'est un carreau, False sinon). Contrainte d'utilisation, le tas `n` doit contenir au moins une carte, sinon une exception est lancée.

```
>>> init_tas(1,"")
>>> if tas_non_vide(1) and sommet_carreau(1):
...     print "le sommet du tas 1 est un carreau"
```

3.2.9 superieur(n, p)

La fonction `superieur(n, p)` retourne la valeur True si la carte au sommet du tas numéro `n` a une valeur supérieure ou égale à celle du tas numéro `p`, et la valeur False dans le cas contraire. Contraire d'utilisation: On ne

peut comparer les cartes situées au sommet de deux tas que s'ils ne sont pas vides. Tout appel à `superieur(n, p)` déclenche une exception si l'un des deux tas est vide.

```
>>> init_tas(1, "P[P+C]")
>>> init_tas(2, "P")
>>> while superieur(1, 2):
...     deplacer_sommet(1, 2)
```

3.2.10 couleur_sommet(n)

La fonction `couleur_sommet(n)` renvoie la couleur de la carte située au sommet du tas numéro `n`. Les quatre couleurs sont TREFLE, COEUR, PIQUE, CARREAU. La couleur renvoyée peut être stockée dans une variable ou comparée à une autre couleur.

```
>>> init_tas(1, "P")
>>> init_tas(2, "P")
>>> init_tas(3, "T")

>>> if couleur_sommet(1) == couleur_sommet(2):
...     print "les tas 1 et 2 ont la meme couleur"
les tas 1 et 2 ont la meme couleur

>>> if couleur_sommet(1) != couleur_sommet(3):
...     print "les tas 1 et 3 n'ont pas la meme couleur"
les tas 1 et 3 n'ont pas la meme couleur

>>> if couleur_sommet(3) == TREFLE:
...     print "la carte du sommet du tas 3 est un TREFLE"
la carte du sommet du tas 3 est un TREFLE
```

3.2.11 fixer_delai(t)

La fonction `fixer_delai(t)` fixe le délai d'attente entre deux mouvements de carte. L'unité est la seconde: `fixer_delai(0)` résulte en un affichage quasi instantané.

```
>>> fixer_delai(1.5) # 1 seconde et demi

>>> fixer_delai(0) # instantané

>>> init_tas(1, "[P]")
>>> while tas_non_vide(1):
...     deplacer_sommet(1, 2)
```


EXERCICES CARTES

4.1 Descriptions de tas

4.1.1 Exercice A-1

Pour chacune des descriptions qui suivent, donnez l’instruction d’initialisation du tas qui convient :

- le tas 1 contient une carte de couleur ♣ ;
- le tas 1 contient une carte de couleur ♣ ou ♠ ;
- le tas 1 contient une carte de couleur quelconque ;
- le tas 1 contient deux cartes de couleur ♥ ;
- le tas 1 contient une carte de couleur ♥ surmontée d’un ♦ ;
- le tas 1 contient un nombre quelconque de ♠ ;
- le tas 1 contient un nombre quelconque de ♠ ou bien un nombre quelconque de ♥ ;
- le tas 1 contient un nombre quelconque de cartes de couleur ♠ ou ♥ ;
- le tas 1 contient un nombre quelconque de cartes de couleur quelconque ;
- le tas 1 contient au moins un carreau ;
- le tas 1 contient un ♣ surmonté soit d’un nombre quelconque de ♥, soit d’un nombre quelconque non nul de ♠ ;
- le tas 1 contient un nombre pair de ♥ ;
- le tas 1 contient un nombre impair de ♥ ;
- le tas 1 contient un nombre pair de ♣ ou un nombre multiple de 3 de ♠ ;
- les deux cartes extrêmes du tas 1 (la plus basse et la plus haute) sont des ♣, entre les deux il y a un nombre quelconque de successions de deux cartes de couleur ♦♥.

4.2 Séquence

Dans tous les exercices qui suivent, l’énoncé décrit une situation initiale des quatre tas de cartes (dans la syntaxe du module Cartes), et la situation finale à atteindre.

4.2.1 Exercice A-2

Situation initiale : Tas 1 : “TT” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “TT” Tas 3 : “” Tas 4 : “”

4.2.2 Exercice A-3

Situation initiale : Tas 1 : “TK” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “KT” Tas 2 : “” Tas 3 : “” Tas 4 : “”

4.2.3 Exercice A-4

Situation initiale : Tas 1 : “TKTK” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “KKTT” Tas 2 : “” Tas 3 : “” Tas 4 : “”

4.2.4 Exercice A-5

Situation initiale : Tas 1 : “TKCP” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “PCKT” Tas 2 : “” Tas 3 : “” Tas 4 : “”

4.3 Conditionnelle

4.3.1 Exercice A-6

Situation initiale : Tas 1 : “T+P” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “[T]” Tas 3 : “[P]” Tas 4 : “”

4.3.2 Exercice A-7

Situation initiale : Tas 1 : “(T+K+C+P)(T+K+C+P)” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “” Tas 3 : “(T+K+C+P)(T+K+C+P)”[↑] Tas 4 : “”

Le symbole [↑] signifie que les cartes sont dans l’ordre croissant (i.e. la carte du dessous a une valeur inférieure ou égale à celle du dessus).

4.3.3 Exercice A-8

Situation initiale : Tas 1 : “T+K+C+P” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[T]” Tas 2 : “[K]” Tas 3 : “[C]” Tas 4 : “[P]”

4.3.4 Exercice A-9

Situation initiale : Tas 1 : “(T+K+C+P)(T+K+C+P)” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[K+C]” Tas 2 : “[T+P]” Tas 3 : “” Tas 4 : “”

4.4 Itération

4.4.1 Exercice A-10

Situation initiale : Tas 1 : “[T]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “[T]” Tas 3 : “” Tas 4 : “”

4.4.2 Exercice A-11

Situation initiale : Tas 1 : “[K+C][T+P]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[T+P][K+C]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

4.4.3 Exercice A-12

Situation initiale : Tas 1 : “[K]” Tas 2 : “[T]” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[K]” Tas 2 : “” Tas 3 : “[KT]” Tas 4 : “”

OU

Situation finale : Tas 1 : “” Tas 2 : “[T]” Tas 3 : “[KT]” Tas 4 : “”

4.4.4 Exercice A-13

Situation initiale : Tas 1 : “[T]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “[T]” Tas 3 : “[T]” Tas 4 : “”

Le nombre de cartes des deux tas 2 et 3 différant d’au plus 1 dans la situation finale.

4.4.5 Exercice A-14

Situation initiale : Tas 1 : “[T]” Tas 2 : “[K]” Tas 3 : “[P]” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “[T]” Tas 3 : “[K]” Tas 4 : “[P]”

(En faire deux versions, la seconde utilisant une procédure `vider_tas(depart, arrivee)` qui vide le tas depart sur le tas arrivee.)

4.4.6 Exercice A-15

Situation initiale : Tas 1 : “[T]” Tas 2 : “[K]” Tas 3 : “[C]” Tas 4 : “[P]”

Situation finale : Tas 1 : “[P]” Tas 2 : “[T]” Tas 3 : “[K]” Tas 4 : “[C]”

4.4.7 Exercice A-16

Situation initiale : Tas 1 : “[T][K][C][P]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[P][C][K][T]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

4.4.8 Exercice A-17

Situation initiale : Tas 1 : “[T]” Tas 2 : “[K]” Tas 3 : “[P]” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “” Tas 3 : “” Tas 4 : “[TKP][XY][Z]”

où X et Y désignent les deux couleurs restantes lorsque l’une des couleurs manque, et Z désigne la couleur restante lorsque X ou Y manque.

4.4.9 Exercice A-18

Situation initiale : Tas 1 : “[T+K+C+P]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[T]” Tas 2 : “[K]” Tas 3 : “[C]” Tas 4 : “[P]”

4.4.10 Exercice A-19

Situation initiale : Tas 1 : “[T[T]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “[T]”- Tas 3 : “[T]” Tas 4 : “”

le symbole – indique que la carte est de valeur minimale.

4.4.11 Exercice A-20

Situation initiale : Tas 1 : “[T]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “” Tas 2 : “[T]”↑ Tas 3 : “” Tas 4 : “”

Le symbole ↑ signifiant que les cartes sont rangées par ordre croissant de valeurs de bas en haut.

4.4.12 Exercice A-21

Situation initiale : Tas 1 : “[T+K]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[X][Y]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Le symbole X désigne la couleur (♣ ou ◇) la plus nombreuse, l’autre couleur étant désignée par Y.

4.4.13 Exercice A-22

Situation initiale : Tas 1 : “[K[T]” Tas 2 : “” Tas 3 : “” Tas 4 : “”

Situation finale : Tas 1 : “[T+K]” Tas 2 : “[T+K]” Tas 3 : “[T+K]” Tas 4 : “[T+K]”

Les trèfles étant équitablement répartis sur les quatre tas, l’unique carreau se trouvant n’importe où (Remarque : ce problème est infaisable sans le carreau).

4.4.14 Exercice A-23

Situation initiale : Tas 1 : “[T]” Tas 2 : “[K]” Tas 3 : “[C]” Tas 4 : “[P]”

Situation finale : Tas 1 : “[T]”↑ Tas 2 : “[K]”↑ Tas 3 : “[C]”↑ Tas 4 : “[P]”↑

Le symbole ↑ signifiant que les cartes sont rangées par ordre croissant de valeurs de bas en haut.

4.5 Fonctions

4.5.1 Exercice A-24

Soit la fonction:

```
>>> def meme_couleur (couleur, tas):  
...     return couleur == couleur_sommet(tas)
```

1. Quel problème pose cette fonction si lors d'un appel le tas `tas` passé en paramètre est vide ?
2. Comment modifier la fonction `meme_couleur` pour qu'elle renvoie la valeur `faux` si le tas `tas` est vide.

4.5.2 Exercice A-25

1. Écrivez une fonction qui teste l'égalité de la valeur de deux cartes situées au sommet de deux tas que l'on supposera non vides.
2. Puis écrivez une fonction qui teste l'égalité de deux cartes (valeur et couleur) situées au sommet de deux tas supposés non vides.

4.5.3 Exercice A-26

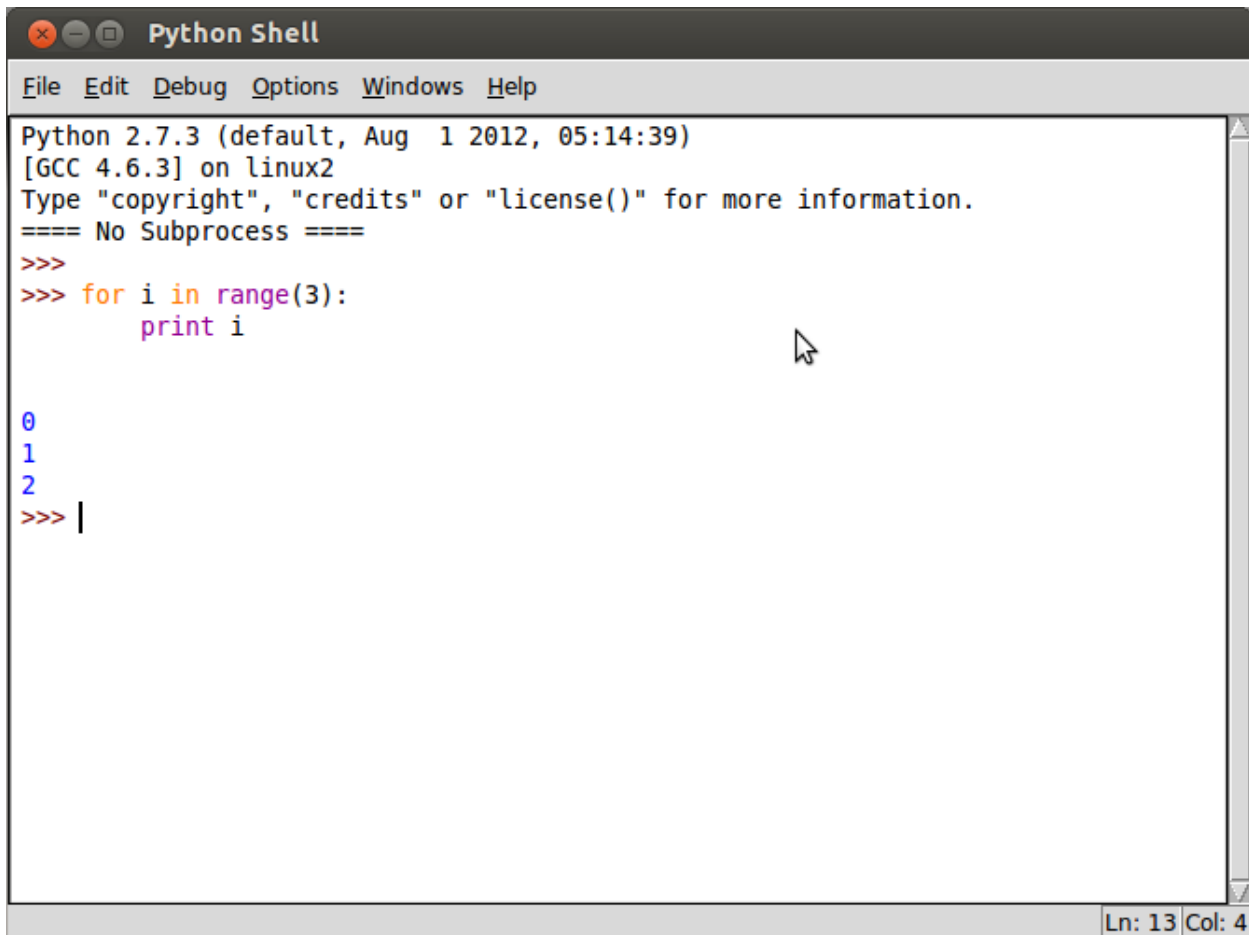
Il s'agit d'écrire une procédure `inverser_tas1` pour inverser l'ordre des tas du tas 1. Par exemple, si on a Tas 1: "TCCPK" alors après exécution de l'instruction `inverser_tas1()`, on doit avoir Tas 1: "KPCCT".

1. Faites-le avec l'hypothèse que les autres tas sont vides.
2. Réaliser la procédure sans supposer que les autres tas sont vides.

ENVIRONNEMENT DE PROGRAMMATION

Pour l'ensemble des travaux pratiques, nous allons utiliser `Idle`, un outil pour écrire des programmes Python. Il propose une réponse aux deux facettes de l'écriture de programmes : éditer du code source et exécuter ce code source.

La première fenêtre qu'`Idle` vous propose est le `Shell`. Un prompt (`>>>`) vous propose d'entrer vos commandes afin de les évaluer.

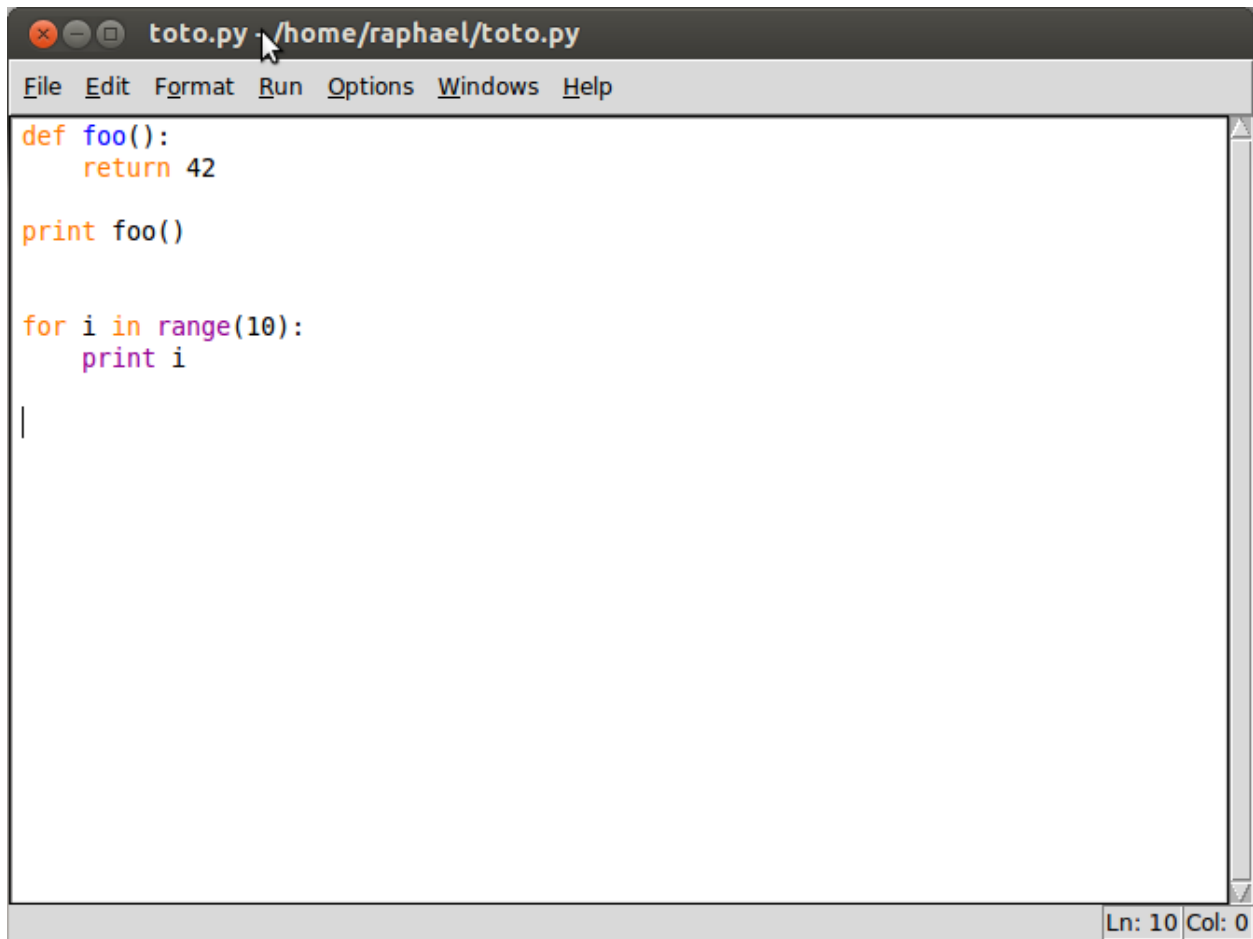


```
Python 2.7.3 (default, Aug 1 2012, 05:14:39)
[GCC 4.6.3] on linux2
Type "copyright", "credits" or "license()" for more information.
==== No Subprocess ====
>>>
>>> for i in range(3):
    print i

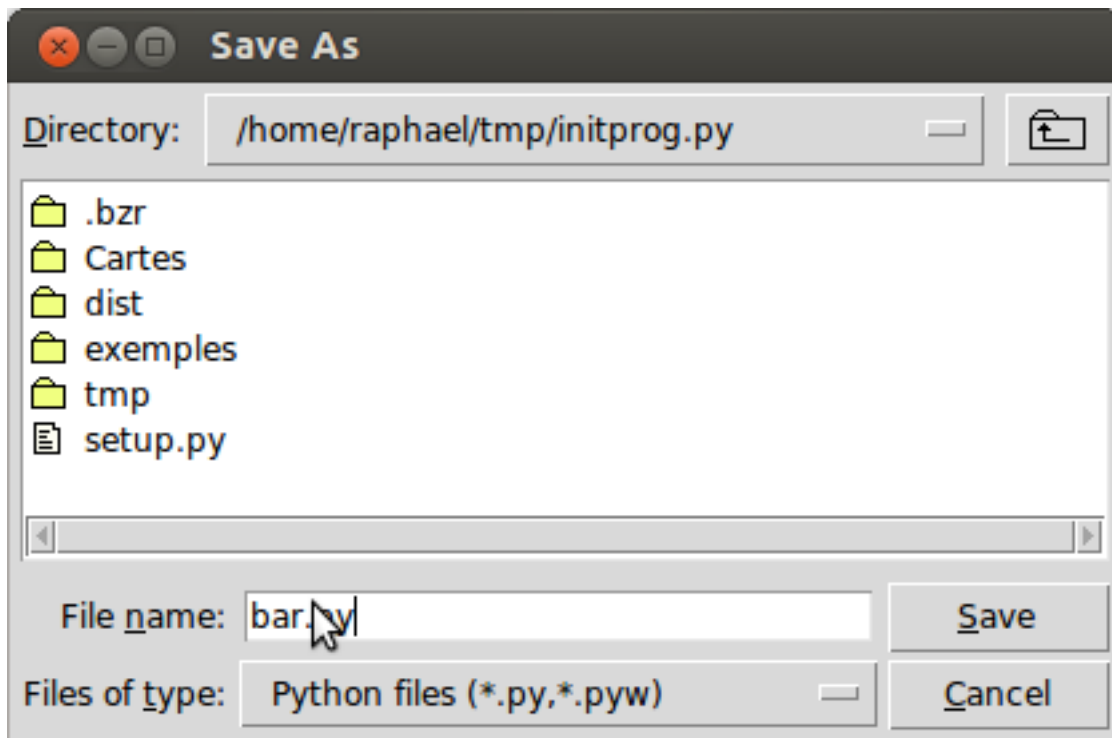
0
1
2
>>> |
```

Ln: 13 Col: 4

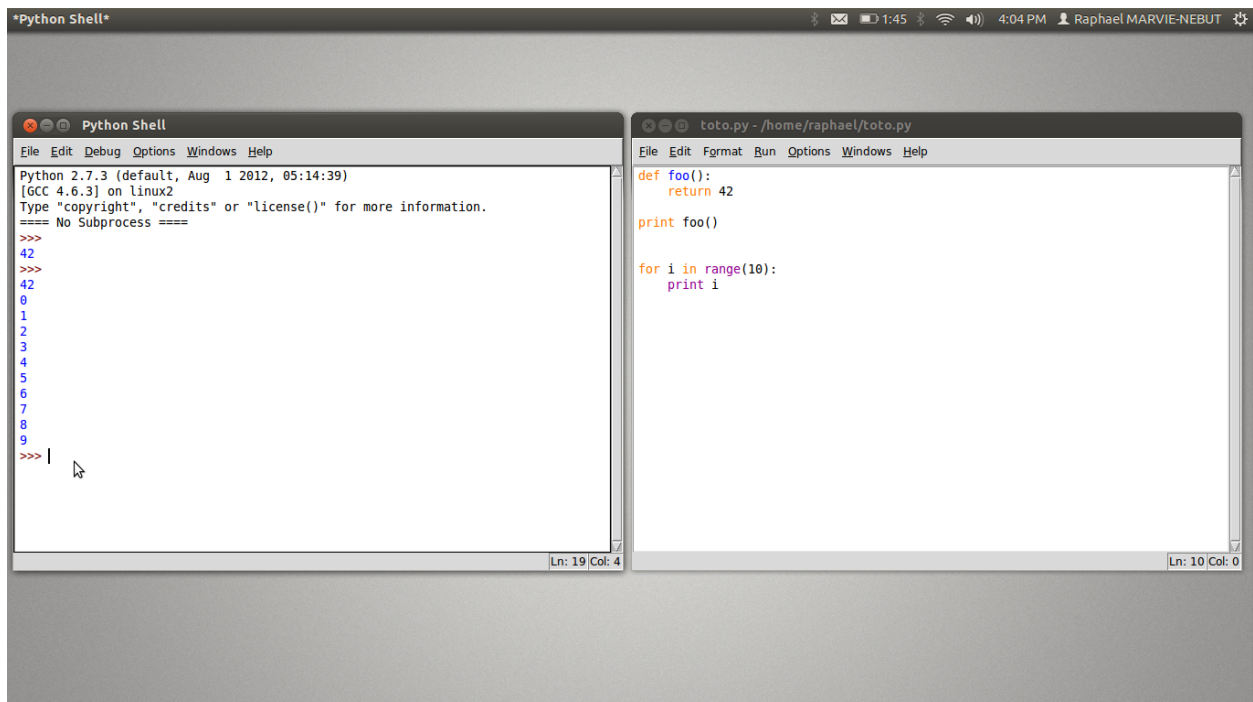
Pour sauvegarder les commandes que vous souhaitez exécuter, `Idle` vous permet de créer un nouveau fichier (`Ctrl+N`). Il ouvre alors une seconde fenêtre qui permet d'éditer du texte.



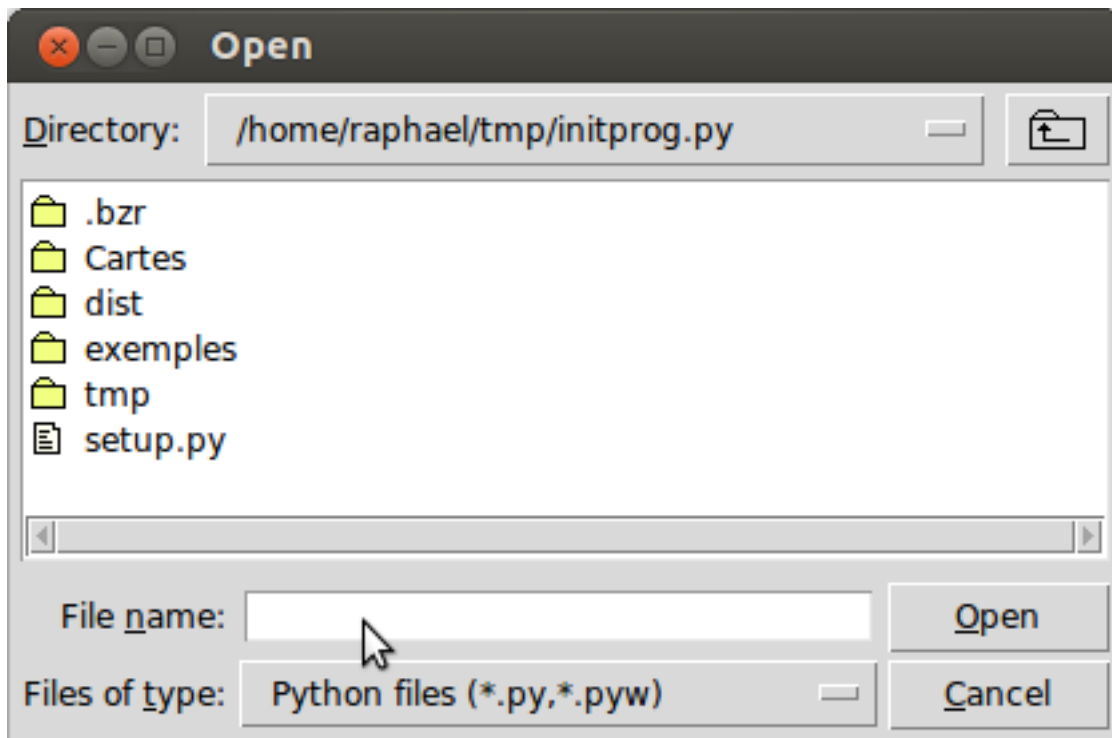
Il est important de sauvegarder régulièrement votre travail (`Ctrl+S`) afin de ne pas le perdre. Idle (comme tout éditeur de texte) vous demandera un nom de fichier et la localisation de ce fichier sur votre espace disque personnel.



Une fois votre travail sauvegardé dans un fichier, Idle permet son exécution en utilisant la touche F5. Le résultat de l'exécution de votre programme contenu dans la fenêtre d'édition s'affichera dans la fenêtre Shell.



Lors d'une prochaine session vous pourrez utiliser de nouveau un de vos programmes en ouvrant le fichier le contenant (Ctrl+O).



Voici la base, pour le reste n'hésitez pas à poser des questions à vos enseignants ou à chercher des informations sur votre moteur de recherche préféré.