UE121 - Programmation : Langage

HAUTE ÉCOLE DE NAMUR-LIÈGE-LUXEMBOURG

Technologie de l'informatique - bloc 1

Sécurité des systèmes - bloc 1

Exercices 5 : Dictionnaires

Objectifs

- Continuer à manipuler les notions de base du langage : variables, littéraux, affectation, opérations arithmétiques, entrées/sorties...
- Manipuler les structures de contrôles et les fonctions.
- Manipuler les séquences : listes, chaînes de caractères et n-uplets.
- Apprendre à utiliser les dictionnaires et le déballage de séqeunces.
- Veiller au Clean Code.
- Veiller à la portabilité du programme.

A. Introduction

Les séries d'exercices visent à mettre en pratique les notions vues lors des ateliers et des séances de mise en commun. Il est important de faire un maximum d'exercices pour vous familiariser avec l'IDLE et avec le langage Python.

Au cas où vous ne termineriez pas les exercices durant la séance, nous vous conseillons de les achever chez vous au plus tôt et surtout avant l'atelier suivant.

Cette quatrième série concerne la manipulation des séquences; listes, strings et tuples.

Note.

Dans les exemples qui suivent, les passages en texte normal sont à sortir tels quels.

Les parties en *italique souligné* correspondent aux entrées de l'utilisateur.

Les portions en **gras** varient en fonction des entrées.

Sur votre partition de travail (`U:`), créez un répertoire appelé Python. Ensuite, pour chaque exercice ci-après, ajoutez-y un fichier intitulé `ex5-XX.py` avec `XX` à remplacer par le numéro de l'exercice en question.

B. Exercices à réaliser dans l'IDLE

Écrivez les fonctions/procédures permettant d'effectuer les actions suivantes.

Un en-tête avec *type hinting* vous est proposé, tentez d'écrire l'algorithme correspondant.

Exercice 1

Ajouter une paire (nom, email) à un dictionnaire qui sert de carnet d'adresse. Si le nom existe déjà, remplacer l'email.

def update_contact(contacts: dict, name: str, mail: str) -> None:

Exercice 2

Afficher l'email correspondant à un nom dans un dictionnaire servant de carnet d'adresse. Afficher une erreur si le nom est introuvable.

def print_mail(contacts: dict, name: str) -> None

Exercice 3

Afficher toutes les paires (nom, email) présent dans le carnet d'adresse sous la forme `"Nom: <nom> - Email: <email>"` avec une entrée par ligne. Utiliser le déballage.

def print_all(contacts: dict) -> None:

Exercice 4

Renvoyer un dictionnaire contenant un compte des lettres d'un mot.

Par exemple, `{'h':1, 'e':1, 'l':2, 'o':1}` pour "hello".

def letters_count(word: str) -> dict:

Exercice 5

Séparer les paires d'un dictionnaire hétérogène en deux dictionnaires distincts, l'un contenant les paires aux clés de type `int`, l'autre de type `str`.

Renvoyer ces deux dictionnaires dans un tuple.

def sort_keys(d: dict) -> tuple:

Exercice 6

Renvoyer une liste de tuples contenant les paires clé-valeur communes à deux dictionnaires.

def common_pairs(d1: dict, d2: dict) -> list:

Exercice 7

Renvoyer un dictionnaire contenant les paramètres d'une Query String d'URL.

Par exemple : {'item'='car', 'color'='red} pour l'URL http://example.com/show_item.php?item=car&color=red.

Servez-vous des méthodes de la classe `str` et du déballage.

def parse_query_string(url: str) -> dict:

C. Exercice FINAL

Dans le cadre de l'élaboration d'un jeu du type Civilisation, la modélisation d'un plateau de jeu est effectuée de la manière suivante :

Chaque case (ou *tile*) est représentée par un dictionnaire qui contient les paires suivantes :

	key	**value**	
		type	**data**
Obligatoires	"terrain"	`str`	"water" | "plain" | "forest" | "mountain"
	"production"	`dict`	un dictionnaire représentant la production
Facultatives	"city"	`str`	le nom de la ville présente sur la case
	"tradepost"	`bool`	`True | False`
	"owner"	`str`	le nom du joueur possédant la case

La production d'une case est représentée par un dictionnaire qui contient les paires suivantes :

	key	**value**	
		type	**data**
Obligatoires	"food"	`int`	Quantité de nourriture produite
	"materials"	`int`	Quantité de matériaux produite
	"gold"	`int`	Quantité d'or produite

Les quantités produites de base dépendent du terrain de la case :

terrain	**food**	**materials**	**gold**
"water"	1	0	0
"plain"	2	0	0
"forest"	1	1	0
"mountain"	0	2	0

Certaines contraintes supplémentaires sont à prendre en compte :

- Il ne peut y avoir qu'une construction par case, ville ('city') ou comptoir ('tradepost')
- On ne peut rien construire sur une case n'appartenant pas à un joueur
- Une case d'eau ('water') ne peut pas posséder de comptoir
- Seules les cases de plaine ('plain') peuvent posséder une ville
- La quantité produite de chaque ressource ne peut jamais être négative

Selon ce qui est construit sur une case, la production est augmentée… Les modificateurs sont les suivants :

- Une ville ajoute 3 'gold' à la production de base la case
- Un comptoir ajoute 1 'gold' à la production de base de la case

Le joueur peut faire certaines opérations sur une case :

- Construire un comptoir ou une ville
- Transformer un comptoir en ville
- Couper la forêt ('forest'), ce qui transforme le terrain en plaine
- Changer/supprimer le propriétaire
- Détruire une construction

Dans un même fichier, écrivez les fonctions suivantes permettant la gestion des cases. Utilisez vos différentes fonctions, de manière à ne pas répéter votre code. Si des contraintes empêchent l'application correcte d'une fonction, afficher un message d'erreur.

Modifier la production

Modifier - et non remplacer - la valeur de la production d'une certaine ressource d'un certain montant, négatif ou positif. Attention à ne pas descendre en-dessous de 0.

def modify_production(tile: dict, resource: str, amount: int) -> None:

Créer une nouvelle case

Créer une nouvelle case vierge d'un certain type de terrain.

def new_tile(terrain: str) -> dict:

Y a-t-il un propriétaire ?

Vérifier si la case a un propriétaire d'un nom donné. Si aucun nom n'est donné, si elle a un propriétaire tout court.

def has_owner(tile: dict, name: str = None) -> bool:

Constructions possibles…

Vérifier si la construction est possible.

def has_city(tile: dict) -> bool:

def has_tradepost(tile: dict) -> bool:

def can_build_city(tile: dict) -> bool:

def can_build_tradepost(tile: dict) -> bool:

Construire sur une case

Construire un comptoir/une ville sur la case.

def build_tradepost(tile: dict) -> None:

def build_city(tile: dict, name: str) -> None:

Attention au cas où on transforme un comptoir en ville...

Couper du bois…

Couper la forêt.

def cut_forest(tile: dict) -> None:

Changer de propriétaire

Changer le propriétaire et renvoyer le nom de l'ancien, ou `None` si aucun. Si aucun nom n'est fourni, supprimer le propriétaire.

def change_owner(tile: dict, name: str = None) -> str:

Détruire une construction

Détruire toute construction. Renvoyer si oui ou non quelque chose a bien été détruit.

def raze_building(tile: dict) -> bool:

Combien de ville ?

Compter le nombre de villes possédées par un joueur. Une liste contenant toutes les cases du plateau est reçue en paramètre.

def count_cities(tiles: list, name: str) -> int:

Combien ça rapporte ?

Compter la production combinée de toutes les cases d'un joueur. Une liste contenant toutes les cases du plateau est reçue en paramètre. Un dictionnaire représentant la production totale est renvoyé. Servez-vous du déballage.

def count_total_production(tiles: list, name: str) -> dict: