INF1 : Algorithmique et Programmation

Cours 13 : Compléments

Domitile Lourdeaux

Université de technologie de Compiègne

Printemps 2024

Différences : fonctions, procédures et méthodes

Une fonction retourne une valeur, une procédure n'en retourne pas

- Une fonction s'utilise au niveau d'une expression, une procédure au niveau d'une instruction
  - c = carre(a)
  - affichage(t)
- En Python, une procédure est en fait une fonction qui retourne None

Une méthode s'applique à un objet. Elle peut le modifier.

- objet.méthode(paramètres)

En python, une méthode retourne une valeur ou la valeur None

- Si elle retourne une valeur elle s'utilise au niveau d'une expression, sinon au niveau d'une instruction
- Par exemple, pour un cercle c (voir cours précédent) :
  - p = c.perimetre() # la méthode perimetre retourne un réel
  - c.translate(3, 5) # la méthode translate s'utilise au niveau d'une instruction. Elle modifie l'objet c

Fonctions retournant un tuple (1)

Il est parfois nécessaire qu'une fonction retourne plus d'une valeur

Une solution est alors de retourner un tuple

Exemple :

- Ecrire une fonction permettant de convertir en heure, minutes, secondes une durée exprimée en secondes

```
h, m, s = conversion(duree)
print(f'{h} heures, {m} minutes, {s} secondes')
```

Fonctions retournant un tuple (2)

```
def conversion(duree):
    "convertit la duree exprimee en secondes en h, m, s"
    heures = duree // 3600
    minutes = (duree % 3600) // 60
    secondes = duree - heures * 3600 - minutes * 60
    return heures, minutes, secondes

temps = int(input("duree en secondes ? "))
h, m, s = conversion(temps)
print(f'{h} heures, {m} minutes, {s} secondes')
```

Lisibilité et accessibilité des fonctions (1)

```
def moyenne(tab):
    def somme(tab):
        s = 0
        for e in tab:
            s += e
        return s
    def afficheTab(tab):
        s = str(tab[0])
        for i in range(1,len(tab)):
            s += "+" + str(tab[i])
        return s
    s = somme(tab)
    moy = s / len(tab)
    print(f'La moyenne de {afficheTab(tab)} = {moy}')
moyenne([1, 2, 3, 4, 5])
```

Lisibilité et accessibilité des fonctions (2)

```
def somme(tab):
    s = 0
    for e in tab:
        s += e
    return s
def afficheTab(tab):
    s = str(tab[0])
    for i in range(1, len(tab)):
        s += "+" + str(tab[i])
    return s
def moyenne(tab):
    s = somme(tab)
    moy = s / len(tab)
    print(f'La moyenne de {afficheTab(tab)} = {moy}')
moyenne([1, 2, 3, 4, 5])
```

Accès aux attributs depuis l'extérieur d'une classe (1)

Accès direct possible mais déconseillé

```
print(p.x)
p.y = 10
```

Il est fortement conseillé de définir si nécessaire des méthodes :

- d'accès (accesseurs) : **getter**
- de modification (mutateurs) : **setter**

```
    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_y(self, value):
        self.y = value

print(p.get_x())
p.set_y(10)
print(p.get_y())
```

Accès aux attributs depuis l'extérieur d'une classe (2)

Utilité

Les accesseurs et les mutateurs permettent de **sécuriser** (e.g. vérifications)

```
class Personne:
    def __init__(self, nom):
        "Constructeur"
        self.nom = nom

    def get_age(self):
        return self.age

    def set_age(self, value):
        if age <= 0 or age >= 150
            print('Age out of range')
        else:
            self.age = value
```

Dictionnaire ou objet ? (1)

On peut représenter une structure composite par un dictionnaire ou par un objet

Exemple : notes d'un étudiant (nom, prénom, note_median, note_final)

- Dictionnaire
  - etu = {'nom' : 'xxx', 'prenom' : 'yyy', 'median' : 9, 'final' : 15}
- Objet

```
class Etudiant:
    def __init__(self, n, p, m, f):
        self.nom = n
        self.prenom = p
        self.median = m
        self.final = f
    def get_final(self):
        return self.final
etu = Etudiant('xxx', 'yyy', 9, 15)
```

Dictionnaire ou objet ? (2)

Pour représenter les étudiants d'une UV

- Tableau de dictionnaires
- UV = [{'*nom*' : '*xxx*','*prenom*' : '*yyy*','*median*' : 9,'*final*' : 15},{....},{....}]
- Ou objet contenant un tableau d'étudiants

```
class UV:
    def __init__ (self):
        self.tab_etu = []
    def ajoute(self, etudiant):
        self.tab_etu.append(etudiant)
uv = UV()
etu = Etudiant('xxx', 'yyy', 9, 15)
uv.ajoute(etu)
```

Dictionnaire ou objet ? (3)

Calcul de la moyenne du final

Fonction avec un tableau de dictionnaires

```
def moyenne_final(uv):
    somme = 0
    n = len(uv)
    for i in range(n):
        somme += uv[i]['final']
    return somme / n
```

Méthode avec l'objet UV

```
class UV:
    ....
    def moyenne_final(self):
        somme = 0
        n = len(self.tab_etu)
        for i in range(n):
            somme += self.tab_etu[i].get_final()
        return somme / n
uv = UV()
etu = Etudiant('xxx', 'yyy', 9, 15)
uv.ajoute(etu)
etu = Etudiant('zzz', 'ttt', 9, 10)
uv.ajoute(etu)
print(uv.moyenne_final())
```

Rappel sur les choix multiples

```
# Calcul de l'etat de l'eau a partir de sa temperature
temperature = int(input("Quelle est la temperature de l'eau
if temperature < 0 :
    # Etat de glace (t < 0)
    etat = 'glace'
elif temperature < 100 :
    # Etat liquide (0 < t < 100)
    etat = 'liquide'
else :
    # Etat de vapeur (t >= 0)
    etat = 'vapeur'
print("Etat = ", etat)
```

Problème : rendu de monnaie (1)

Un commerçant souhaite rendre la monnaie à l'un de ses clients en utilisant le moins de pièces et de billets possibles

Ecrire un programme qui détermine la **combinaison optimale de pièces et de billets** en fonction de la somme à rendre

- On ne considère pas les centimes
- On considère seulement les coupures et pièces en euros : $1, 2, 5, 10, 20, 50, 100, 200$
- On suppose que le commerçant dispose d'une réserve suffisante pour chaque espèce

Problème : rendu de monnaie (2)

Exemple : Somme à rendre = 9 €

Combinaison	Pièces
9 x 1€	9
7 x 1€ + 1 x 2€	8
5 x 1€ + 2 x 2€	7
3 x 1€ + 3 x 2€	6
1 x 1€ + 4 x 2€	5
4 x 1€ + 1 x 5€	5
2 x 1€ + 1 x 2€ + 1 x 5€	4
2 x 2€ + 1 x 5€	3

Problème : rendu de monnaie (3)

Algorithme « glouton »

- On sélectionne les billets ou pièces à rendre un à un
- On choisit à chaque fois la meilleure solution vis-à-vis de l'objectif, c'est-à-dire la plus grande valeur possible

---

Algorithm 1 Algorithme glouton : rendu de monnaie

$euros \leftarrow [200, 100, 50, 20, 10, 5, 2, 1]$
Lire (s) `// s la somme à rendre`
$i \leftarrow 0, nb \leftarrow 0$
`tant que` $s > 0$ `faire`
  `si` $euros[i] <= s$ `alors`
    $s \leftarrow s - euros[i]$
    $nb = nb + 1$
  `sinon`
    $i \leftarrow i + 1$
  `fin`

Problème : rendu de monnaie (4)

Code Python « glouton »

```
def monnaie(s):
    euros = [200, 100, 20, 10, 5, 2, 1]
    nb = 0
    i = 0
    while s > 0:
        if euros[i] <= s:
            n = s // euros[i]
            print(n, ' fois ', euros[i], ' euros')
            s = s - n * euros[i]
            nb = nb + n
        else:
            i += 1
    return nb
```

On peut montrer que cet algorithme est optimal

Parcours de villes (1)

- On considère un ensemble de villes
- Comment **visiter toutes les villes en minimisant la distance totale parcourue**, en partant d'une ville et en revenant à cette même ville ?
- Exemple : Nancy, Metz, Paris, Reims, Troyes
- S'il faut partir et revenir de Nancy, il y a en tout $4! = 24$ circuits différents

Remarques

- Si le nombre de villes devient grand, le problème peut nécessiter un très grand nombre d'opérations
- On peut là-aussi appliquer un algorithme glouton, en choisissant à chaque étape la ville la plus proche

Source : "NumÃľriqueetSciencesInformatiques", ÃľditionsEllipses2019

Parcours de villes (2)

Parcours possibles :

Circuit	Détail	Total
Metz – Paris – Reims- Troyes	55 + 306 + 142 + 123 + 183	809
Metz – Paris – Troyes - Reims	55 + 306 + 153 + 123 + 188	825
Metz – Troyes – Paris - Reims	55 + 203 + 153 + 142 + 188	741
Troyes – Metz - Paris - Reims	183 + 203 + 306 + 142 + 188	1022
Troyes – Metz – Reims - Paris	183 + 203 + 176 + 142 + 303	1007
Metz –Troyes - Reims - Paris	55 + 203 + 123 + 142 + 303	826
Metz - Reims – Troyes - Paris	…	810
Metz - Reims – Paris - Troyes	…	709
Reims – Metz– Paris - Troyes	…	1006
Reims – Metz – Troyes - Paris	…	1023
Reims – Troyes – Metz - Paris	…	1123
Troyes – Reims – Metz - Paris	…	1091

+ les douze circuits en sens inverse

Parcours de villes (3)

Algorithme glouton

- on choisit à chaque étape la ville la plus proche
- cela permet de déterminer un circuit parmi les meilleurs, mais pas forcément le meilleur

Données

- Tableau des noms de villes
- Tableau booléen des villes visitées
- Matrice des distances entre villes

```
villes = ['Nancy','Metz', 'Paris', 'Reims', 'Troyes']
dist = [[0, 55, 303, 188, 183],
        [55, 0, 306, 176, 203],
        [303, 306, 0, 142, 153],
        [188, 176, 142, 0, 123],
        [183, 203, 153, 123, 0]]
```

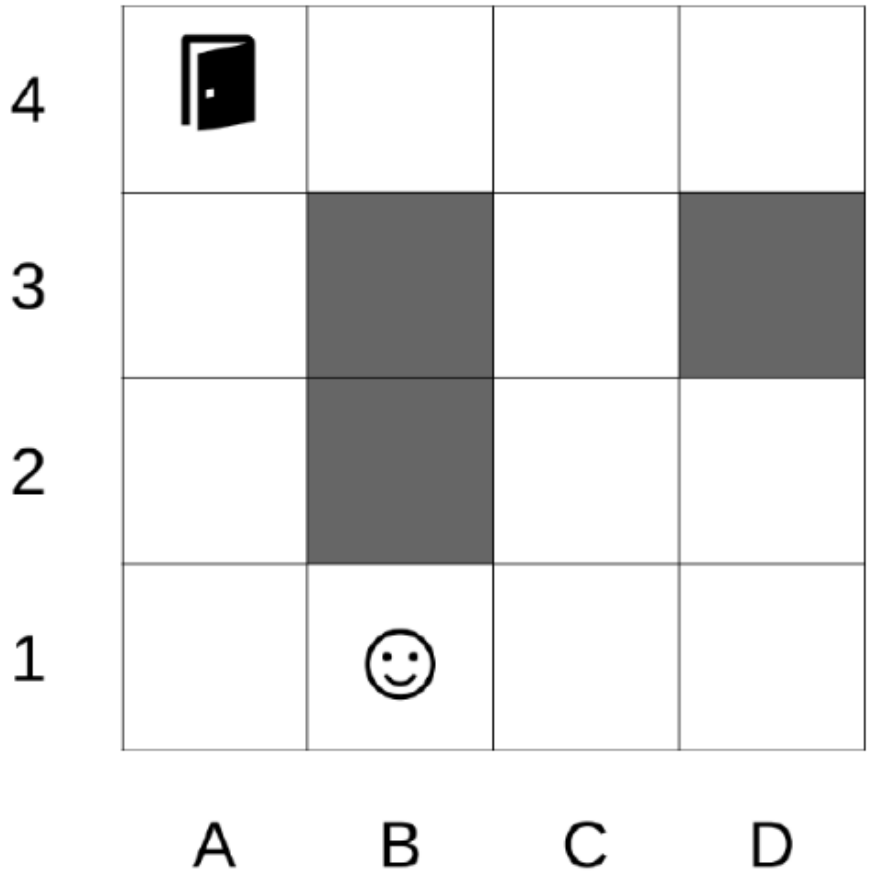
Parcours de villes (4)

Détermination d'un circuit acceptable

```
def circuit_glouton(villes, dist, depart):
    n = len(villes)
    visitees = [False] * n
    distance_totale = 0
    courante = depart
    for i in range(n - 1):
        visitees[courante] = True
        suivante = plus_proche(courante, dist, visitees)
        distance_totale += cumul_etape(courante, suivante,
        courante = suivante
    distance_totale += cumul_etape(courante, depart, dist)
    print('distance totale :', distance_totale)
```

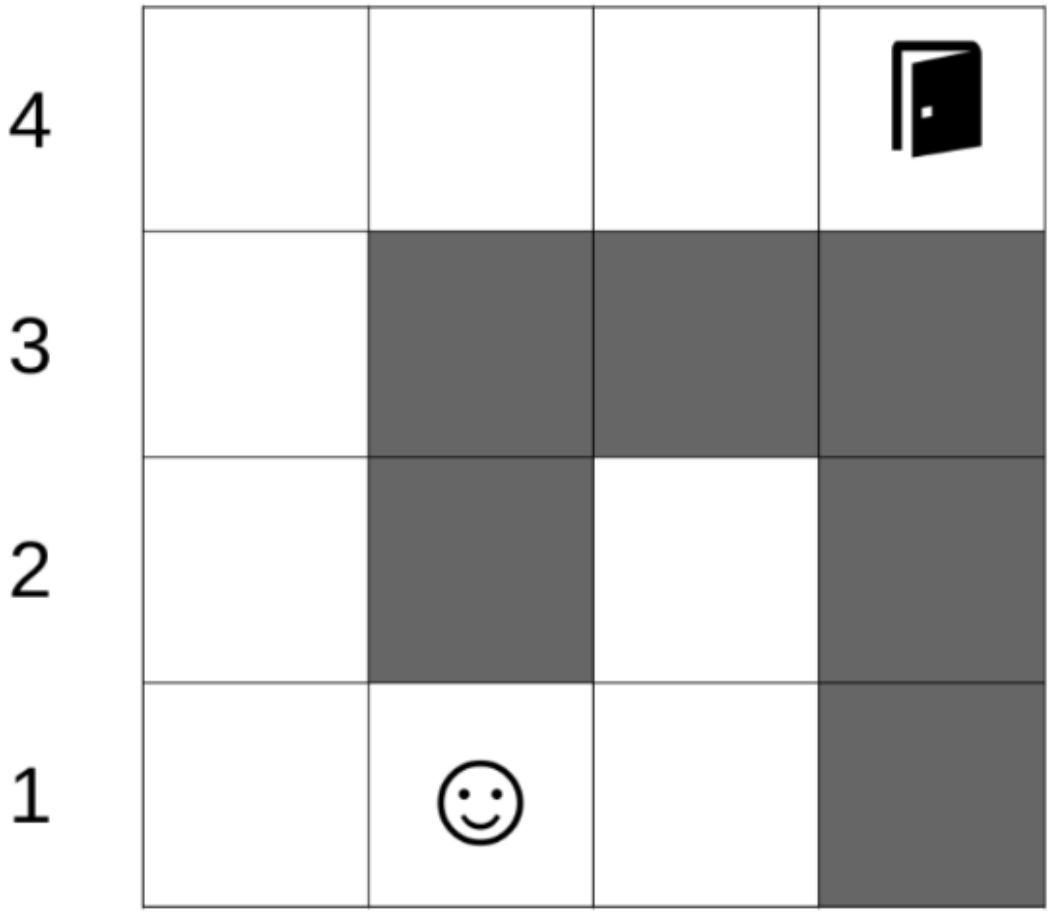
Parcours de labyrinthe (1)

Heuristique : $h(s)$ estimation du coût minimal entre l'état courant et l'un des buts (ex : Distance de Manathan)



Parcours de labyrinthe (2)

Heuristique : $h(s)$ estimation du coût minimal entre l'état courant et l'un des buts (ex : Distance de Manathan)



Questions...