Programmation et IA

Cours 4

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-ia-25

Plan

- classes et objets
- exemples de méthodes
- files, piles, files de priorité
- modularité et interfaces

[texte des programmes]

Pour apprendre Python:

w3schools: tutoriel et réference

`https://www.w3schools.com/python/default.asp`

programiz: tutoriel et réference

`https://www.programiz.com/python-programming`

classes

et objets

Classes et objets

- une classe décrit un ensemble d'objets tous de la même forme avec des attributs et des méthodes

```
class Point:
    def __init__ (self, x, y) :
        self.x = x
        self.y = y

    def __str__ (self) :
        return f'({self.x}, {self.y})'

    def __add__ (self, delta) :
        return Point (self.x + delta.x, self.y + delta.y)
```

constructeur d'un nouvel objet

`__str__` est appelé par `print`

`__add__` est appelé par `+`

- objets dans cette classe

```
p1 = Point (100, 200)
print (p1)
(100, 200)

p2 = Point (200, 100)
print (p2)
(200, 100)

p3 = p1 + p2
print (p3)
(300, 300)
```

nouvel objet de la classe `Point`

[texte des programmes]

Classes et objets

- les attributs d'un objet ont des valeurs quelconques

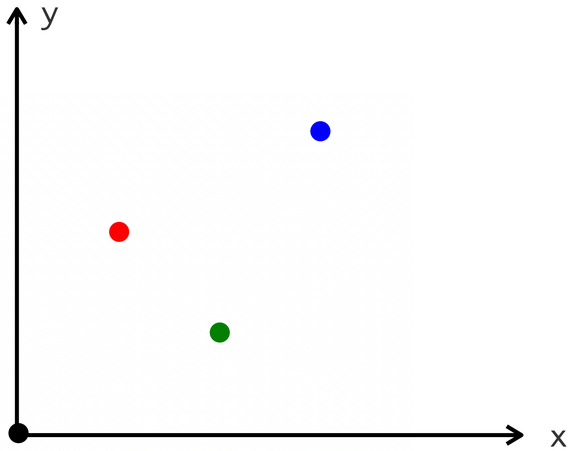
```
class Point:
    # comme avant

    def __le__ (self, p) :
        return self.x <= p.x and self.y <= p.y

    def draw (self) :
        t.penup()
        t.goto (self.x, self.y)
        t.pendown()
        t.dot()
```

`__le__` est appelé par `<=`

pour dessiner l'objet avec la petite tortue

- exemples de dessins

```
t.pensize(10)
p0 = Point (0, 0); p0.draw()
t.pencolor('red'); p1.draw()
t.pencolor('green');p2.draw()
t.pencolor('blue');p3.draw()
t.hideturtle()
t.done()
```



[texte des programmes]

Classes et objets

- dans la classe `Point`, les **attributs** sont `x` et `y` et les **méthodes** `__init__, __str__, __add__, __le__, draw`
- la méthode `__init__` **construit les objets** de cette classe
- en général, les attributs de chaque objet sont spécifiés et initialisés dans la méhode constructeur `__init__`
- les méthodes s'appliquent à l'objet `self` **qui les contient**

- en Python, les **méthodes spéciales** sont encadrés par `__ .. __`

  `print(p)` équivaut à `print (p.__str__())`

  `p+q` équivaut à `p.__add__(q)`

  `p <= q` équivaut à `p.__le__(q)`

- dans la classe `Point` nous avons aussi défini la méthode `draw`

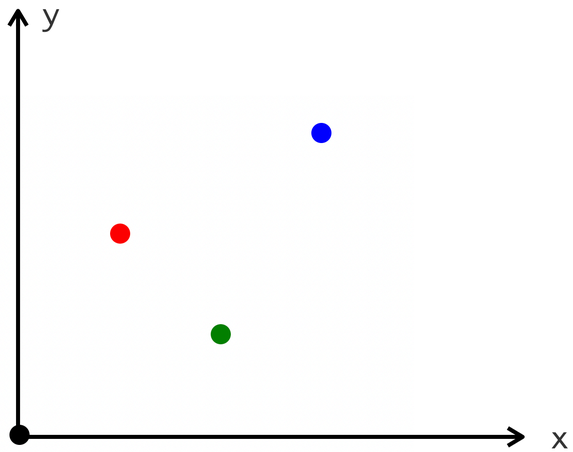
  - `p.draw ()`

[texte des programmes]

Classes et objets

- on aurait aussi pu définir une simple fonction *draw* (en dehors de la classe)

```
import turtle as t

def draw (p) :
    t.penup()
    t.goto (p.x, p.y)
    t.pendown()
    t.dot()


t.pensize(10)
p0 = Point (0, 0); draw(p0)
t.pencolor('red'); draw(p1)
t.pencolor('green'); draw(p2)
t.pencolor('blue'); draw(p3)
t.hideturtle()
t.done()
```



- choisir entre fonctions et méthodes est une affaire de style. (style impératif ou orienté-objet, cf. plus tard)

- dans le graphique, on préfère souvent les méthodes

[texte des programmes]

Classes et objets

- le constructeur de la classe `Rectangle` utilise des objets `Point`

```
class Rectangle:
    def __init__ (self, p, q) :
        self.bas_gauche = p
        self.haut_droite = q
        if not p <= q :
            raise ValueError

    def __str__ (self) :
        return f'Rectangle ({self.bas_gauche}, {self.haut_droite})'

    def draw (self) :
        t.penup()
        t.goto (self.bas_gauche.x, self.bas_gauche.y)
        t.pendown()
        t.goto (self.haut_droite.x, self.bas_gauche.y)
        t.goto (self.haut_droite.x, self.haut_droite.y)
        t.goto (self.bas_gauche.x, self.haut_droite.y)
        t.goto (self.bas_gauche.x, self.bas_gauche.y)
```

on vérifie que q est dans le quadrant supérieur droit

```
r = Rectangle (p1, p3)
print (r)
r.draw()
t.hideturtle()
t.done()
```

[texte des programmes]

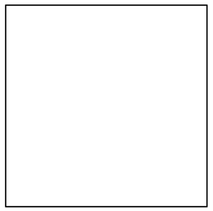
Classes et héritage

- une classe peut être une sous-classe d'une classe plus générale

```
class Carre (Rectangle) :
    def __init__ (self, p, c) :
        super().__init__ (p, p + Point(c, c))
```

on appelle le constructeur de `Rectangle`

```
c0 = Carre (Point(0,0), 150)
print (c0)
r.draw()
c0.draw()
t.hideturtle()
t.done()
```



- un carré est un rectangle particulier
- le constructeur de `Carre` appelle le constructeur de la super classe `Rectangle`

Exercice écrire la classe `Polygone` qui construit des objets à partir d'une liste de `Point`

Exercice écrire la classe `Triangle`

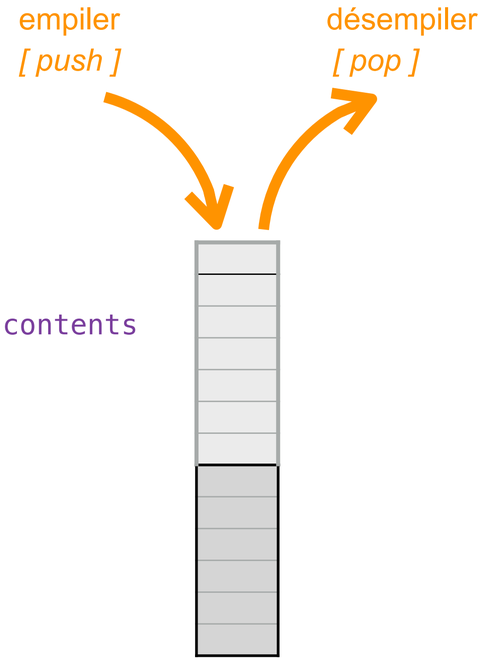
Exercice écrire les méthodes `perimetre` et `surface`

[texte des programmes]

Piles et files d'attente

- classe des piles (LIFO)

```
class Stack :
    def __init__(self) :
        self.contents = []

    def __str__(self) :
        return f'{self.contents}'

    def push (self, x) :
        self.contents = [x] + self.contents

    def pop (self) :
        x = self.contents[0]
        del self.contents[0]
        return x

    def is_empty (self) :
        return self.contents == [ ]
```



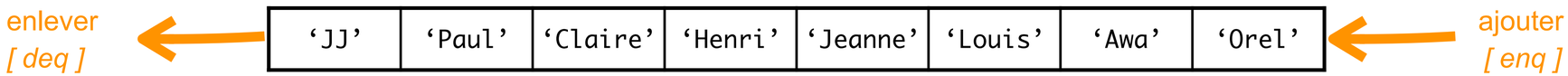
Last In First Out

opérations des piles en O(n)

- classe des files d'attente (FIFO)

```
class Queue :
    def __init__(self) :
        self.contents = []

    def __str__(self) :
        return f'{self.contents}'

    def enq (self, x) :
        self.contents = self.contents + [x]

    def deq (self) :
        x = self.contents[0]
        del self.contents[0]
        return x

    def is_empty (self) :
        return self.contents == [ ]
```

First In First Out



First In First Out

opérations des files en O(n) amorti

avec des doubles-listes

[texte des programmes]

Files de priorité

- classe des files de priorité

```
class PQueue (Queue) :
    def __init__(self) :
        super().__init__()

    def deq (self) :
        i = self.contents.index (max (self.contents))
        x = self.contents[i]
        del self.contents[i]
        return x
```

opérations en O(n) où n est la longeure de la file

mais on peut faire les opérations des files de priorité en O(log n)

avec des tas (*heap*)

[texte des programmes]

Classes et objets

- classe des files de priorité avec des tas (*heap*)

```
class PQueue (Queue) :
    def __init__(self) :
        super().__init__()

    def enq (self, x) :
        a = self.contents; a.append(x)
        n = len (a); i = n - 1
        while i > 0 and a[(i-1) // 2] < x :
            a[i] = a[(i-1) // 2]
            i = (i-1) // 2
        a[i] = x
```

```
def deq (self) :
        a = self.contents; n = len (a)
        res = a[0]
        v = a[0] = a[n-1]
        i = 0
        while 2*i + 1 < n-1 :
            j = 2*i + 1
            if j + 1 < n-1 :
                if a[j+1] > a[j] :
                    j = j + 1
            if v >= a[j] :
                break
            a[i] = a[j]; i = j
        a[i] = v
        del a[n-1]
        return res
```

opérations en O(log n) où n est la longeure de la file

- l'utilisation extérieure est identique

[texte des programmes]

Classes et objets

- l'utilisation extérieure est indépendante de l'implémentation

```
p = Stack()
p.push ('JJ')
p.push ('Paul')
p.push ('Claire')
p.push ('Henri')
print (p)
for i in range (4) :
    x = p.pop()
    print (x, '..', p)
```

```
f = Queue()
f.enq ('JJ')
f.enq ('Paul')
f.enq ('Claire')
f.enq ('Henri')
print (f)
for i in range (4) :
    x = f.deq()
    print (x, '..', f)
```

```
fp = PQueue()
fp.enq ((20, 'JJ'))
fp.enq ((2, 'Paul'))
fp.enq ((5, 'Claire'))
fp.enq ((13, 'Robert'))
print (fp)
for i in range (4) :
    (n, s) = fp.deq()
    print (n, s, '..', fp)
```

encapsulation

abstraction

seul l'interface compte

[texte des programmes]

Classes et objets

- les classes permettent d'**encapsuler** un ensemble de données (attributs) et de fonctions (méthodes)
- les classes représentent donc une forme de **modularité**
- à l'extérieur, le détail de l'implémentation des objets de cette classe est **opaque**
- on peut donc modifier la représentation sans changer leur interface et les fonctions qui utilisent cette classe
- **attention**: classes et modules sont deux notions différentes en Python
  [les modules sont importés et attachés à la notion de fichier]

Prochain cours

- réviser les classes et objets
- arbres
- arbres binaires de recherche
- programmation procédurale ou par objets
- programmation impérative ou fonctionnelle
- arbres binaires équilibrés