Laboratori de LP: Python 2

1 Classes a Python

Python ha estat sempre un llenguatge orientat a objectes, per això crear i usar classes i objectes és molt fàcil. Veiem un primer exemple.

```
class Treballador:
    'classe comuna per a tots el traballadors'
    treCompt = 0

    def __init__(self, nom, salari):
        self.nom = nom
        self.salari = salari
        Treballador.treCompt += 1

    def mostrarCompt(self):
        print("Total Treballadors %d" % Treballador.treCompt)

    def mostrarTreballador(self):
        print("Nom : ", self.nom, ", Salari: ", self.salari)
```

En aquesta definició de la classe `Treballador`, tenim que `treCompt` és una *variable de classe*, ja que és compartida per totes les instàncies de la classe.

Noteu novament que els atributs es declaren amb assignacions dins dels mètodes. Així, els objectes (instància) de la classe `Treballador`, tindran `nom` i `salari`, com a *variables d'instància*.

El mètode `__init__`, defineix l'operació creadora, que es crida amb el nom de la classe.

El paràmetre `self` representa l'objecte que s'està creant en el `__init__` o l'objecte al que s'aplica l'operació en els altres casos. És com el `this` de `Java`, però a Python cal posar-lo explícitament com a paràmetre i quan s'usa.

```
tre1 = Treballador("Albert", 2000)
tre2 = Treballador("Jordi", 2500)
```

i podem accedir a les operacions i als atributs.

```
tre1.mostrarTreballador()
tre2.mostrarTreballador()
```

```
print("Total Treballadors %d" % Treballador.treCompt)
tre1.mostrarCompt()
```

Podem afegir, eliminar o modificar atributs de classes i objectes en qualsevol moment.

```
tre1.edat = 7
tre1.edat = 8
del tre1.edat
Treballador.maxSalari = 0
Treballador.maxSalari += 1
del Treballador.maxSalari
```

També podem usar les següents funcions. Noteu que per accedir a un atribut ha d'existir, sino es produeix un error.

```
if hasattr(tre1, 'edat'):
    print(getattr(tre1, 'edat'))
setattr(tre1, 'edat', 8)
delattr(tre1, 'edat')
```

Existeixen alguns atributs predefinits, que té tota classe: `__dict__`, `__doc__`, `__name__`, `__module__`, `__bases__`.

Igual que a Java, els objectes es poden compartir (aliasing) i són esborrats pel *garbage collector* quan el seu comptador de referències està a zero (és a dir, que ningú el referencia).

Encara que existeix una operació d'esborrat per defecte, la podem redefinir.

```
class Punt:
      def __init__( self, x=0, y=0):
            self.x = x
            self.y = y
      def __del__(self):
            nom_classe = self.__class__.__name__
            print(nom_classe, "destroyed")
      def mostraPunt(self):
            print("({},{})".format(self.x,self.y))

pt1 = Punt()
pt2 = pt1
pt3 = Punt(3,2)
print(id(pt1), id(pt2), id(pt3))
del pt1
del pt2
del pt3
```

En Python per ocultar atributs, només cal posar un nom a l'atribut que comenci per dues barres baixes, com ara

```
class Comptador:
   __ocultCompt = 0

   def compta(self):
      self.__ocultCompt += 1
      print(self.__ocultCompt)
```

Si ara fem

```
compt = Comptador()
compt.compta()
print(compt.__ocultCompt)
```

dona error, però sempre podem fer

```
print(compt._Comptador__ocultCompt)
```

1.1 Herència

En Python podem definir fàcilment subclasses que hereten els atributs no redefinits del la classe pare.

```
class Fill(Punt):
      def fillMetode(self):
            print('Cridem al metode del fill')

c = Fill()
c1 = Fill(3,2)
c.fillMetode()
c1.mostraPunt()
print(c1.x)
```

En Python podem tenir herència mùltiple.

```
class A: # definim la classe A
.....
class B: # definim la calsse B
.....
class C(A, B): # subclasse de A i B
.....
```

Per fer comprovacions, podem usar les funcions `issubclass(sub, sup)` i, com ja s'ha vist, `isinstance(obj, Class)`.

2 Memorització

En aquesta secció veurem com algunes de les característiques de Python ens ajuden a fer programació dinàmica fàcilment.

Les dues primeres versions són específiques per una funció donada. En canvi les dues darreres ens proporcionen una funció general que aplica memorització a una funció donada.

2.1 Memorització usant paràmetres amb valor per defecte

La solució passa per utilitzar una funció amb un parametre amb valor per defecte que es un map. L'efecte de memorització s'obté perquè Python usa com a valor per defecte un objecte "mutable", és a dir que sempre és el mateix objecte en totes les crides i per tant te efectes laterals en totes elles.

```
def efib (n,mem={0:0,1:1}):
    if n not in mem:
        mem[n] = efib (n-1) + efib (n-2)   #es passa mem per defecte amb l'acumulat
    return mem[n]
```

Fixeu-vos que aquest tractament dels valors per defecte pot donar resultats inesperats. Per exemple, amb la funció:

```
def funcio(llista=[]):
    llista.append(1)
    return llista
```

Cada cop que crideu a `funcio()` en un programa el resultat és diferent.

```
print(funcio())
print(funcio())
print(funcio())
```

retorna

```
[1]
[1, 1]
[1, 1, 1]
```

2.2 Memorització usant funcions imbricades

La solució passa per definir un map i una funció imbricada que usa aquest map com a global.

```
def efib (x):
    mem = {0:0,1:1}    # la memòria
    def mfib (n):
        if n not in mem: mem[n] = mfib(n-1) + mfib(n-2)
        return mem[n]
    return mfib (x)
```

2.3 Memorització genèrica usant ordre-superior i funció imbricada

La solució és com l'anterior però passant la funció per paràmetre. En aquest cas donada una funció ens retorna una funció que memoritza.

```
def memoritza (f):
    mem = {}    # la memòria
    def f2 (x):
        if x not in mem:
            mem[x] = f(x)
        return mem[x]
    return f2
```

Fixeu-vos que això retorna una funció que té una variable global associada. Aquesta variable global té un comportament similar al del paràmetre amb valor per defecte, ja que serà el mateix objecte mutable per a qualsevol crida que fem. Noteu que això només es pot fer amb objectes mutables (que recordeu que no inclou, els nombres, els strings o les tuples).

Si la funció a la que s'aplica memorització és recursiva, llavors per tal que aquesta solució apliqui memorització recursivament cal fer una redefinició de la funció.

```
def mfib (x):
    if x==0: return 0
    if x==1: return 1
    return mfib(x-1) + mfib(x-2)
mfib = memoritza (mfib)
```

Això funciona perquè Python és interpretat i per tant la redefinició que hem fet té efecte en totes les crides posteriors de la funció.
Si, en canvi, fem

```
mfib1 = memoritza (mfib)
```

la primera crida a `mfib1(32)` serà lenta, però si tornem a cridar, ja serà ràpida, ja que la funció retornada té la mateixa "variable global" `mem`.

2.4 Memorització genèrica usant classes

```
class Memoritza:
    def __init__(self, f):
        self.f = f
        self.mem = {}
    def __call__(self, arg):
        if arg not in self.mem:
            self.mem[arg] = self.f(arg)
        return self.mem[arg]
```

Noteu que, igual que a l'anterior secció aquesta generalització és només per funcions d'un paràmetre. Per canviar això s'han d'usar transformacions a tupla com ara usar `*args` com a argument.
La crida, ha de ser com abans

```
mfib = Memoritza (mfib)
```

3 Python Standard Library

Python ens proporciona un gran nombre de classes i mòduls predefinits. Aquí en veurem uns pocs representatius per tractar expressions regulars, tractar arxius en format CSV, processar HTML, processar XML o accedir a dades mitjançant una url.

En cada cas us indicarem quin mòdul us recomanem i podeu trobar exemples d'ús als programes Python adjunts.

3.1 Expressions regulars

El modul predefinit `re` ens dona operacions similars a les que hi ha a Perl. Podeu trobar els detalls, per exemple, a `https://docs.python.org/3/library/re.html`

3.2 Lectura/escriptura d'arxius CSV

El modul predefinit `csv` permet tractar llegir i escriure arxius en format CSV (Comma Separated Values). Podeu trobar els detalls, per exemple, a `https://docs.python.org/3/library/csv.html`

3.3 Processar HTML

Hi ha diverses opcions, però us proposem utilitzar la calsse `HTMLParser`. Podeu trobar els detalls, per exemple, a `https://docs.python.org/3/library/html.parser.html`

3.4 Processar XML

Els mòduls més usuals (i que és mantenen a Python3) per manegar XML són

- `xml.etree.ElementTree`
- `xml.dom` i `xml.dom.minidom`
- `xml.sax`
- `xml.parsers.expat`

Per simplicitat i eficiència (relativa) us recomanem que useu el primer. Podeu trobar els detalls, per exemple, a `https://docs.python.org/3/library/xml.etree.elementtree.html`

3.5 Accedir a recursos mitjançant una url

El mòdul que ens proporciona aquest tipus d'acces és el `urllib`. Podeu trobar els detalls, per exemple, a `https://docs.python.org/3/library/urllib.request.html`

4 Exercici

Feu un script que accedeixi a
`http://wservice.viabicing.cat/getstations.php?v=1`
que us dona en format XML l'estat actual (en temps real) de les estacions de bicing de Barcelona, amb informació sobre la ubicació, bicicletes i places d'aparcament disponibles, etc. Feu cerques sobre aquest arxiu, com ara que ens mostri l'adreça de les estacions que tenen més d'un cert nombre de bicicletes disponibles. Mireu primer l'arxiu per veure el format.