La majoria de programes que hem fet fins ara són una seqüència d'instruccions, més o menys una darrera de l'altra. Les excepcions són les condicions, que fan que s'executi o no un fragment del codi; i els bucles, que fan que una sèrie d'instruccions s'executin més d'una vegada.

A mesura que els programes es fan més complexos, necessitem mecanismes per fer el codi més compacte i ben organitzat. Sovint ens trobem que necessitem una sèrie d'instruccions que fan una funció determinada i que la necessitem en diverses part del programa. No és pràctic copiar i enganxar aquest seguit d'instruccions en diverses part del codi, ja que complicaria molt el manteniment de l'aplicació. També passa sovint que en un programa necessitem una sèrie d'instruccions que fan una determinada funció, i que això ja ho tenim implementat en un altre programa. En comptes d'anar a buscar aquell programa i copiar aquestes instruccions, pot resultar més pràctic tenir-les en un arxiu a part i cridar-les quan les necessitem.

En aquesta unitat estudiarem tres mecanismes que ens permeten organitzar millor el nostre codi:

- les funcions
- la orientació a objectes
- els mòduls i paquets

D'aquesta unitat disposes dels següents arxius:

- La unitat en pdf: 07-Programes_ben_estructurats.pdf
- Els arxius i programes: programes7-cat.zip

7.1. Funcions

Una funció és un fragment de codi amb un nom associat i que realitza una sèrie de tasques i molt sovint retorna un valor. En Python les funcions es declaren de la forma següent (programa-7-1.py):

```
def universal(etiqueta):
 if etiqueta.startswith("N"): etiqueta_universal="NOUN"
 elif etiqueta.startswith("V"): etiqueta_universal="VERB"
 elif etiqueta.startswith("A"): etiqueta_universal="ADJ"
 elif etiqueta.startswith("R"): etiqueta_universal="ADV"
 else: etiqueta_universal="X"
 return(etiqueta_universal)


 tag="NCMS"
tag_u=universal(tag)
print(tag,tag_u)
```

Aquest programa retorna l'etiqueta universal a partir d'un altre etiquetari. Quan l'executes et mostra l'etiqueta completa i la universal. La funció no està completa i és simplement un exemple.

T'atreviries a modificar el programa-5-15.py (que ja deus tenir però que també adjunto en aquesta unitat), afegint aquesta funció adaptada a les etiquetes que fa servir i completada amb totes les opcions necessàries? L'etiquetador entrenat l'adjunto (etiquetador-cat.pkl) però pots fer servir el que vas entrenar tu en aquella unitat. Les etiquetes que fa servir aquest etiquetador les podeu trobar descrites a: https://freeling-user-manual.readthedocs.io/en/latest/tagsets/tagset-ca/

Teniu una descripció detallada sobre les funcions en Python en el següent enllaç: https://docs.python.org/3/tutorial/controlflow.html#defining-functions i en https://docs.python.org/3/tutorial/controlflow.html#more-on-defining-functions

7.2. Orientació a objectes

En l'apartat d'introducció ja comentàvem que Python és un llenguatge multiparadigma amb el qual es pot treballar amb programació estructurada, com veníem fent fins ara, o amb programació orientada a objectes o programació funcional. En aquest apartat anem a explicar els fonaments de la programació orientada a objectes.

Per explicar els principals conceptes, observeu el programa-7-2.py:

```
class Paraula:
 '''classe per representar paraules. Les paraules estan composades per una forma, un lema i una etiqueta'''
 def __init__(self,forma,lema,etiqueta):
     self.forma=forma
     self.lema=lema
```

```
        self.etiqueta=etiqueta
    def torna_forma(self):
        '''Retorna la forma de la paraula'''
        return(self.forma)
    def torna_lema(self):
        '''Retorna el lem de la paraula'''
        return(self.lema)
    def torna_etiqueta(self):
        '''Retorna l'etiqueta de la paraula'''
        return(self.etiqueta)
    def es_lema(self):
        '''Diu si una determinada forma és un lema'''
        if self.forma==self.lema:
            return(True)
        else:
            return(False)
    def categoria(self):
        '''Retorna el nom de la categoria de la paraula'''
        if self.etiqueta.startswith("N"):return("NOM")
        elif self.etiqueta.startswith("V"):return("VERB")
        elif self.etiqueta.startswith("A"):return("VERB")
        elif self.etiqueta.startswith("R"):return("VERB")
        else: return("CATEGORIA TANCADA")


    p1=Paraula("casa","casa","NCFS")
p2=Paraula("cases","casa","NCFP")
p3=Paraula("noi","npi","NCMS")

    print("LEMA DE CASES:",p1.torna_lema())
print("CASA ÉS LEMA?:",p1.es_lema())
print("CASES ÉS LEMA?:",p2.es_lema())
print("CATEGORIA DE NOI:",p3.categoria())
```

Si l'executem, proporciona la següent sortida:

```
    LEMA DE CASES: casa
CASA ÉS LEMA?: True
CASES ÉS LEMA?: False
CATEGORIA DE NOI: NOM
```

En l'exemple anterior definim una classe que s'anomena Paraula. Aquesta classe ens permet definir una sèrie d'objectes que pertanyen a aquesta classe: p1, p2 i p3. Quan creem un d'aquests objectes s'invoca el mètode __init__ quem indica que necessitem tres valors per crear la paraula: forma, lema i etiqueta, La classe té 5 mètodes més: torna_forma, torna_lema, torna_etiqueta, es_lema i categoria. Tots els objectes que hem creat d'aquesta classe dispondran d'aquests mètodes.

Fixeu-vos en les cadenes de text entre tres cometes simples (''). Són la documentació de la classe. En el programa-7-3.py hem mantingut la definició de la classe i al final hem posat

```
    print(help(Paraula))
```

i ens mostra per pantalla la documentació de la classe, que no és més que el nom de la classe i els seus mètodes i l'explicació que hem posat entre les triples cometes.

```
    class Paraula(builtins.object)
 |  classe per representar paraules. Les paraules estan composades per una forma, un lema i una etiqueta
 |
 |  Methods defined here:
 |
 |  __init__(self, forma, lema, etiqueta)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  categoria(self)
 |      Retorna el nom de la categoria de la paraula
 |
 |  es_lema(self)
```

```
|      Diu si una determinada forma és un lema
|
|  torna_etiqueta(self)
|      Retorna l'etiqueta de la paraula
|
|  torna_forma(self)
|      Retorna la forma de la paraula
|
|  torna_lema(self)
|      Retorna el lem de la paraula
|
|  ----------------------------------------------------------------------
|  Data descriptors defined here:
|
|  __dict__
|      dictionary for instance variables (if defined)
|
|  __weakref__
|      list of weak references to the object (if defined)
```

Una altra cosa que us pot cridar l'ateció, és l'ús de self tant en la definició dels mètodes com en el nom de les variables que fem servir. Aquest self es refereix a cada propi objecte de la classe. Així, en els mètodes passem self perquè ens referim a un objecte concret de la cllase. Si no fem servir self en les variables, es tractaria d'una variable de classe que estaria compartida per tots els objectes de la classe. Fent servir self, les variables són d'instància, és a dir, cada objecte té la seva pròpia variable i modificant el valor d'una deterninada variable d'un dels objectes no modificaria el valor d'aquesta variable en la resta de classe. Per regla general, voldrem que les variables siguin d'instància, és a dir, que cada objecte tingui els seu propi valor per aquesta classe,

En Python 3 tot són classes. Si iniciem un intèrpret interactiu i fem:

```
>>> a=2
>>> type(a)
<class 'int'>
```

ens està indicant que a és un objecte de tipus 'int'. Podrem accedir a la documentació sobre la classe int fent:

```
>>> help(int)
```

o bé, com que a és un objecte d'aquesta classe, també podrem fer:

```
>>> help(a)
```

El tema de les classes en Python és molt ampli i complex. Amb el que hem explicat fins aquí podrem començar a desenvolupar les nostres classes. Qui vulgui aprofundir més en el tema pot consultar el tutoria oficial de Python en aquest enllaç: https://docs.python.org/3/tutorial/classes.html

7.3. Mòduls i paquets

És molt possible que una determinada funció o bé una determinada classe la haguem de necessitar en diversos programes. La solució d'anar copiant i enganxant aquesta funció o classe no és adequada. Imaginem-nos que fem una millora a la funció. Si volem actualitzar tots els programes que la utilitzen, hauríem d'anar buscant-los tots, editar-los per canviar la funció i tornar-los a guardar.

Podem fer un arxiu de python que contingui la funció o classe i fer-la servir sempre que vulguem en qualsevol programa. Anem a crear un mòdul que contingui la funció que transforma una determinada etiqueta a l'etiquetari universal (aquesta funció la vam crear al programa-7-1.py). A aquest mòdul li direme elmeumodul.py (en casos reals caldria buscar un nom més adequat.

```
def universal(etiqueta):
    if etiqueta.startswith("N"): etiqueta_universal="NOUN"
    elif etiqueta.startswith("V"): etiqueta_universal="VERB"
    elif etiqueta.startswith("A"): etiqueta_universal="ADJ"
    elif etiqueta.startswith("R"): etiqueta_universal="ADV"
    else: etiqueta_universal="X"
```

```
    return(etiqueta_universal)
```

Ara podem cridar aquest mòdul de diverses maneres. En el programa-7-5.py importem concretament la funció universal del mòdul (el mòdul podria contenir més funcions i si les volguéssim importar totes faríem from elmeumodul import *. Com que hem importat concretament la funció universal, la podrem cridar directament amb aquest nom.

```
from elmeumodul import universal

tag="NCMS"
tag_u=universal(tag)
print(tag,tag_u)
```

En el programa-7-5b.py cridem el mòdul sense especificar la funció. Llavor, per cridar a la funció haurem d'especificar el nom del módul i farem elmeumodul.universal.

```
import elmeumodul

tag="NCMS"
tag_u=elmeumodul.universal(tag)
print(tag,tag_u)
```

El que hem explicat per funcions també és vàlid per a classes. Anem a implementar la classe Paraula (creada al program-7-2.py) com a un mòdu, que anomenarem elmeumodul2 (recordeu que és millor posar uns noms més significatius):

```
class Paraula:
    '''classe per representar paraules. Les paraules estan composades per una forma, un lema i una etiqueta'''
    def __init__(self,forma,lema,etiqueta):
        self.forma=forma
        self.lema=lema
        self.etiqueta=etiqueta
    def torna_forma(self):
        return(self.forma)
    def torna_lema(self):
        return(self.lema)
    def torna_etiqueta(self):
        return(self.etiqueta)
    def es_lema(self):
        if self.forma==self.lema:
            return(True)
        else:
            return(False)
    def categoria(self):
        if self.etiqueta.startswith("N"):return("NOM")
        elif self.etiqueta.startswith("V"):return("VERB")
        elif self.etiqueta.startswith("A"):return("VERB")
        elif self.etiqueta.startswith("R"):return("VERB")
        else: return("CATEGORIA TANCADA")
```

Ara el podem cridar també de dues maneres. La primera la veiem al programa-7-6.py:

```
from elmeumodul2 import Paraula

p1=Paraula("casa","casa","NCFS")
```

```
p2=Paraula("cases","casa","NCFP")
p3=Paraula("noi","npi","NCMS")

print("LEMA DE CASES:",p1.torna_lema())
print("CASA ÉS LEMA?:",p1.es_lema())
print("CASES ÉS LEMA?:",p2.es_lema())
print("CATEGORIA DE NOI:",p3.categoria())
```

I la segona al programa-7-6b.py:

```
import elmeumodul2

p1=elmeumodul2.Paraula("casa","casa","NCFS")
p2=elmeumodul2.Paraula("cases","casa","NCFP")
p3=elmeumodul2.Paraula("noi","npi","NCMS")

print("LEMA DE CASES:",p1.torna_lema())
print("CASA ÉS LEMA?:",p1.es_lema())
print("CASES ÉS LEMA?:",p2.es_lema())
print("CATEGORIA DE NOI:",p3.categoria())
```

Ara per ara hem situat els nostres mòduls al mateix directori que els programes que els criden. Quan cridem a un mòdul, Python busca primer al mateix directori on es troba el programa. Si no està allà, busca a un director de sistema que pot variar una mica segons la instal·lació, però que probablement sigui:

- En Windows: C:\PythonXY\Lib\site-packages
- En Linux i Mac: /usr/local/lib/pythonX.Y/site-packages

on X i Y són els números de versió. Si desenvolupem un mòdul que pensem fer servir molt el podem situar en aquest directori i estarà disponible per a tots els programes.

Amb el que hem après aquí sobre mòduls serà en general suficient per desenvolupar els nostres programes. Si volem ampliar els coneixements podeu consultar el tutorial oficial de Python: https://docs.python.org/3/tutorial/modules.html

Fins aquí hem parlat de mòduls. Quan desenvolupem una sèrie de mòduls amb unes finalitats més o menys comunes, l'organitzarem com a paquets. Un paquet és un seguit de mòduls organitzats de manera jeràrquica (físicament s'organitzen en una estructura de directoris). Per fer-nos una idea, podem explorar el directori de mòduls i paquets del nostre sistema. Jo ho reprodueixo en un Linux i en Terminal, però també ho podeu fer en Windows i amb l'explorador d'arxius:

Mirem què hi ha en el directori dist-packages. N'hi ha molts, però només reprodueixo uns pocs:

```
aoliverg@aovant:/usr/local/lib/python3.4/dist-packages$ ls
argh
argh-0.26.1-py3.4.egg-info
babel
...
nltk
...
```

entro en l'nltk i miro què hi ha:

```
aoliverg@aovant:/usr/local/lib/python3.4/dist-packages$ cd nltk
aoliverg@aovant:/usr/local/lib/python3.4/dist-packages/nltk$ ls
align            data.py        lazyimport.py   text.py
app              decorators.py  metrics         tgrep.py
book.py          downloader.py  misc            tokenize
ccg              draw           parse           toolbox.py
chat             featstruct.py  probability.py  treeprettyprinter.py
chunk            grammar.py     __pycache__     tree.py
classify         help.py        sem             treetransforms.py
cluster          inference      stem            util.py
collocations.py  __init__.py    tag             VERSION
compat.py        internals.py   tbl             wsd.py
corpus           jsontags.py    test
```

ara entrem a tokenize i mirem el contingut:

```
aoliverg@aovant:/usr/local/lib/python3.4/dist-packages/nltk/tokenize$ ls
api.py       punkt.py     regexp.py  simple.py    texttiling.py  util.py
__init__.py  __pycache__  sexpr.py   stanford.py  treebank.py
aoliverg@aovant:/usr/local/lib/python3.4/dist-packages/nltk/tokenize$
```