Laboratori de LP: Python 1

1 Introducció

Per complementar aquestes notes podeu mirar manuals i cursos (tutorials) com els de la llista `https://wiki.python.org/moin/BeginnersGuide/Programmers`. Per exemple podeu usar `http://www.tutorialspoint.com/python/index.htm`, o llibres com ara "Programming Python", de Mark Lutz. Tingueu en compte que usarem Python3, que no és totalment compatible amb Python2.

Python és un llenguatge de scripting d'alt nivell orientat a objectes. És interpretat i es pot usar interactivament. Un objectiu important de Python és la llegibilitat: s'usen amb freqüència paraules (reservades) en anglès en lloc de símbols de puntuació (com en altres llenguatges) i té menys construccions sintàctiques que altres llenguatges.

Nosaltres l'usarem de forma no-interactiva, escrivint els programes en arxius. Per exemple, podem crear un arxiu `hola.py` simplement amb la instrucció:

```
print("Hola!")
```

i executar-lo per línia de comandes amb

```
$ python3 hola.py
```

Com en tots els scripts en Unix, podem posar en la capçalera amb què s'ha d'executar el codi que va a continuació.

```
#!/usr/bin/python3

print("Hola!")
```

En aquest cas, un cop modificades les propietats de l'arxiu per a que sigui executable amb per exemple

```
$ chmod +x hola.py
```

el podeu cridar directament amb

```
$ ./hola.py
```

Podeu afegir paràmetres a la crida usant el mòdul `sys` i accedir-hi via la llista `sys.argv`.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

import sys

print('El programa', sys.argv[0], 'té', len(sys.argv)-1, 'arguments.')
print("Hola "+sys.argv[1]+"!")
```

Com s'ha vist a l'anterior exemple, també podem indicar el sistema de caràcters que estem usant (s'ha de posar si no estem usant 8-bit ASCII).

1.1 Identificadors

Els identificadors segueixen les regles usuals (una lletra o subratllat, seguit de lletres, digits o subratllats).

La llista de paraules reservades conté `and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `with`, `yield`.

Com a convenció, normalment s'usen majúscules per començar els identificadors de noms classes (hi ha algunes convencions més pels identificadors dins de classes) i minúscules per la resta.

1.2 Indentació i línies de codi

Una de les primeres particularitats que trobem en Python és que no hi ha símbols com les claus { i } per indicar blocs de codi. En Python, un bloc (anomenat *suite*) s'identifica obligatòriament amb indentacions a les línies amb blancs o tabuladors, **però no els barregeu**. Tot el que pertany a un mateix bloc té la mateixa indentació. Si un bloc conté altres blocs, aquest tindran més indentació.

A diferència de Haskell, on també hem usat la indentació de la mateixa manera, en Python no hi ha alternativa.

Per altra banda, en Python les instruccions acaben normalment amb el final de línia. Si volem partir una instrucció en vàries línies, hem d'usar el símbol \. Com ara a,

```
total = x*100 + \
        y*10 + \
        r
```

Amb strings, es poden usar les triples cometes per partir-los en diferents línies. No hi ha diferència entre les simples i les dobles.

```
salutacio1 = "Hola Albert!"
salutacio2 = "Hola Albert!"
salutaciollarga = """Hola Albert! Com va tot? A veure
si quedem un dia."""
print(salutacio1)
```

```
print(salutacio2)
print(salutaciollarga)
```

Las línies comentades comencen per `#`, com en molts llenguatges de scripting.

Es pot posar més d'una instrucció en una línia separant-les amb `;` i les línies de capçalera per condicionals, iteracions o definicions de funcions o classes, comencen per una paraula clau i acaben amb `:` que segueix, a partir de la següent línia amb un bloc (amb la mateixa indentació). Per exemple,

```
if expressio1:
   bloc1
else:
   bloc2
```

o també

```
if expressio1:
   bloc1
elif expressio2:
   bloc2
else:
   bloc3
```

2 Variables i tipus de dades estàndard

A Python, les variables no s'han de declarar explícitament. La declaració es fa implícitament quan s'assigna un valor a variable.

```
#!/usr/bin/python3

comptar = 100     # Assignem un enter
metres = 1000.0   # Assignem un real en coma flotant
nom = "Albert"    # Assignem un string

print(comptar)
print(metres)
print(nom)
```

Es poden fer assignacions múltiples i redefinir variables com ara

```
x = y = z = 0
x, y, z = 0, 5, "Llenguatges"
x, y = y, x
```

Com a tipus estàndard a Python tenim:

- Nombres: `int`, `long`, `float` i `complex`.

- Strings (`str`). Podem accedir a una posició `z[2]`, podem agafar una subcadena `z[3:6]`, un prefix `z[:4]`, un sufix `z[4:]`, concatenar `z+" Programacio"`, repetir `z * 3`, o preguntar la mida `len(z)`.

- Llistes (`list`). Són heterogènies. Mateixes operacions que per strings.

  ```
  z = ["hola",5,"llenguatge",6.63,2]
  ```

  A més de moltes altres operacions predefinides de la classe `list`, com ara `append`, `count`, `pop`, etc.

- Tuples (`tuple`). Són com les llistes però s'escriuen entre parèntesis i no es poden modificar. Són només de lectura.

  ```
  z = ("hola",5,"llenguatge",6.63,2)
  z = (5,)
  ```

  aquest darrer representa una tupla amb un sol element (noteu que `(5)` és simplement l'enter `5`).

- Diccionaris (`dict`). Contenen parells clau-valor i permeten accés directe. Es poden inicialitzar amb diferents parells `clau : valor` separats per comes i entre claus { }. La resta d'assignacions o consultes es fan amb `[ ]`. També es pot demanar la mida.

  ```
  dic = {}
  dic["prim"] = "el primer"
  dic[2] = "el segon"
  print(dic)
  dic = {"nom": "albert","num":37899, "dept": "computer science"}
  print(dic)
  dic["prim"] = "el primer"
  dic[2] = "el segon"
  print(dic)
  ```

- Conjunts (`set`). Admet les operacions usuals de conjunts: `len`, `in`, `not in`, `add`, `remove`, `issubset` (`<=`), `issuperset` (`>=`), `union` (`|`), `intersection` (`&`), `difference` (`-`).

Tenim operacions de conversió:

- `int(x,[base])`, `long(x,[base])`, `float(x)` i `complex(real [,imag])`.

- `chr(x)` i `unichr(x)`, converteix un enter a caràcter (unicode) i `ord(x)`, converteix un caràcter a enter.

- `str(x)`, converteix a string

- `repr(x)`, converteix l'objecte en el string que el representa

- `eval(x)`, avalua un string. Si representa un objecte, com ara una llista, el converteix en aquest objecte. Per exemple, `eval("[2,14,8]")` és la llista `[2,14,8]`

  Alerta amb usar `eval` si es llegeix d'una font desconeguda, ja que pot acabar avaluant "*malware*" i que ens generi un problema. Per exemple,

  `eval("print('2+2')")` escriu `4` i

  `eval("os.system('rm *')")` ens esborrarà tots els fitxers del directori.

- `tuple(x)`, converteix a tupla.
- `list(x)`, converteix a llista.
- `set(x)`, converteix a conjunt.
- `dict(x)`, converteix una llista de parell (clau,valor) a diccionari.

Finalment podem preguntar si un objecte és d'una certa classe o classes:

```
isinstance(x, list)
isinstance(x, (list, tuple))
```

2.1 Llistes per comprensió en Python

Igual que a Haskell podem definir llistes per comprensió. Per exemple, podem definir

```
quadrats1 = [x**2 for x in range(10)]
quadrats2 = [x**2 for x in range(1,10)]
```

on `range(n)` és la llista `[0..n-1]` i `range(m,n)` és la llista `[m..n-1]`. Podem tenir combinacions de `for` i `if`

```
[x for x in L if x<y]
```

També podem tenir combinacions de generadors

```
parells=[(x,y) for x in range(4) for y in range(4) if x!=y]
```

3 Operacions

Com en tots els llenguatges tenim operacions aritmètiques: $+$, $-$, $*$, $/$, $\%$ (mòdul), $**$ (exponencial) i $//$ (quocient de la divisió); operacions relacionals: $==$, $!=$, $<$, $<=$, $>$, $>=$; i operacions booleanes: `and`, `or`, `not` (amb valors `True` i `False`). També hi ha operacions a nivell de bits per enters.

Com ja s'ha vist, tenim l'assignació ($=$) i també totes les assignacions acumulant ($+=$, $-=$, ...) per cada operació aritmètica, però no hi ha ni `++` ni `--`.

3.1 Condicionals

Tenim `if`, `else` (que pot aparèixer o no) i `elif` (que poden aparèixer diversos seguits o cap).

Així, el podem tenir només amb `if`

```
if x in L:
   print(x, "hi és")
```

o bé, si només és un

```
if x in L: print(x, "hi és")
```

El podem tenir amb `if` i `else`

```
if x in L: print(x, "hi és")
else: print(x, "no hi és")
```

o amb `if` i `elif`

```
if x in L: print(x, "hi és")
elif x in L1: print(x, "no hi és a la primera però sí a la segona")
```

o amb `if`, `elif` i `else`

```
if x in L: print(x, "hi és")
elif x in L1: print(x, "no hi és a la primera però sí a la segona")
else:
   print(x, "no hi és a cap llista")
```

3.2 Iteracions

Tenim el `while` i el `for`. El primer és similar a altres llenguatges però el `for` permet iterar sobre els elements d'una seqüència (llista, tupla, conjunt, etc).

```
compt = 0
while compt < 9:
   print("El comptador és:", cont)
   cont = cont + 1
```

Amb el `for` tenim per exemple

```
for x in l:
   if x % 2 == 0:
      s = str(x)+" es parell"
   else:
      s = str(x)+" es senar"
   print(s)
```

Aquest esquema és pot aplicar a qualsevols del tipus estructurats estàndard (llistes, tuples, conjunts o diccionaris), ja que tots tenen l'operació `in`. En el cas de diccionaris itera sobre les claus.

Alternativament podem recorrer sobre els index d'una llista amb un `for` usant el `range`

```
for i in range(len(l)):
   if l[i] % 2 == 0:
      s = str(l[i])+" es parell"
   else:
      s = str(l[i])+" es senar"
   print(s)
```

També podem usar el `break` i el `continue`, amb la semàntica usual.

Tant pel `while` com pel `for` també es pot posar un `else` al final que s'executa quan falla la condició del `while` o quan s'han tractat tots els elements del `for`. Noteu que si es surt amb un `break` o amb un `return` (el veurem després), llavors no s'executa el `else`.

```
for x in llista:
  if (x%elem == 0):
    print(elem, "divideix a", x)
    break
else: print("Cap element divideix a", x)
```

3.3 Entrada/Sortida simple

Per llegir fàcilment dades per teclat podem fer

```
n = input("\n\nIntrodueix un string.")
```

Si no ho assignem simplement espera que s'escrigui alguna cosa.

```
input("\n\nPremeu return per continuar.")
```

Si volem llegir un enter podem fer

```
n = int(input("\n\nIntrodueix un enter."))
```

Si volem llegir una llista podem fer

```
l = eval(input("\n\nIntrodueix una llista:"))
```

però això té els riscos ja esmentats d'usar el `eval`.

Per escriure per pantalla usarem, com ja hem vist el `print` que pot imprimir un o més valors separats per comes. Si n'hi ha més d'un afegeix un blanc entre cada valor.

```
print("Hola "+ nom +"!")
print("Hola", nom, "!")
```

Podem indicar al `print` que ha de posar al final (per defecte és un salt de línia com hem vist).

```
print("Hola",end=" ")
print("Jordi!")
```

4 Definició de funcions

En Python es poden definir fàcilment noves funcions usant la paraula clau `def` i `return`.

```
def functionname(parameters):
   "function_docstring"
   function_suite
   [return [expression]]
```

Com a exemple tenim

```
def divideix(elem, llista):
   "retorna si l'element divideix a algun valor de la llista o no"
   for x in llista:
     if x%elem == 0: return True
   return False
```

Si la funció no torna res, es pot posar el `return` sense res més o no posar `return` i la funció acabarà quan arribi al final.
Es poden posar valors per defecte:

```
def divideix(elem, llista=[]):
   "retorna si l'element divideix a algun valor de la llista o no"
   for x in llista:
     if x%elem == 0: return True
   return False
```

El pas de paràmetres en una crida a funció és pot fer de diferents maneres.

- Amb els paràmetres requerits

  ```
  print(divideix(3, [2,5,8,1]))
  ```

- Amb els paràmetres usats com a paraules claus

  ```
  print(divideix(llista=[2,5,8,1], elem=3))
  ```

- Sense incloure els paràmetres amb valor per defecte

  ```
  print(divideix(3))
  ```

- També es poden tenir funcions amb una longitud variable de paràmetres (no els usarem).

4.1 Funcions anònimes i funcions d'ordre superior

Són lambda-expressions com les vistes a Haskell.

```
lambda [arg1 [,arg2,.....argn]] : expressio
```

També podem tenir funcions d'ordre superior. Com ara el map (que, de fet, és predefinit).

```
def map(fun, llista):
   "retorna si l'aplicació de la funció a tot els elements"
   L = []
   for x in llista:
     L.append(fun(x))
   return L
```

i cridar-la amb

```
inc = lambda arg: arg+1
print(map(inc, [3,6,8,1]))
```

o directament amb

```
print(map(lambda arg: arg*2, [3,6,8,1]))
```

Un altre exemple usant el `if-then-else` com a funció és

```
print(map(lambda arg: 1 if arg==6 else 0, [3,6,8,1]))
```

4.2 Funcions d'ordre superior predefinides

Les més conegudes són: `filter` i `map`. També hi ha el `reduce`, que és semblant al `foldl` del Haskell, però en Python3 no està directament disponible i es troba al mòdul `functools`.

- `filter`. Té dos arguments: `filter(func, seq)`, on `func` té un argument i resultat booleà. A diferència de Python 2, retorna un generador (veure Secció 6), que es pot convertir en llista.

  ```
  list(filter(lambda x: x>3, [3,6,8,1]))
  list(filter(lambda x: x>3, (3,6,8,1)))
  ```

- `reduce`. Pot ser usat amb dos arguments o tres arguments: `reduce(func, seq, [ini])`, on `func` té dos arguments (on el primer és l'acumulat) i `seq` no és buida si no hi ha `ini`, que és el que ens dona el valor inicial. Amb tres arguments és com el `foldl` i amb dos arguments és com els `foldl1`.

  ```
  from functools import reduce
  reduce(lambda acc,y: acc+y, [3,6,8,1])
  reduce(lambda acc,y: acc+y, [3,6,8,1],0)
  ```

fan el mateix, però només el segon funciona per llistes buides.

- `map`. Té dos arguments: `map(func, seq)`, on `func` té un argument, i retorna generador.

  ```
  list(map(lambda x: x*2, [3,6,8,1]))
  ```

5 Visibilitat de les variables

Les variables assignades dins d'una funció (`def` o `lambda`) són sempre locals excepte si es declaren globals. Una variable és local si el primer ús que se'n fa és una assignació (recordeu que no hi ha declaracions). La resta de variables vistes fins ara (noteu que encarar no hem vist classes) són globals.

```
def cerca(x):
   global i
   i = 0
   while i<len(L):
      if x==L[i]: break
      i += 1
   else: i = -1

L = eval(input("\n\nIntrodueix una llista:"))
n = eval(input("\n\nIntrodueix un valor:"))
cerca(n)
print(i)
```

En aquest cas, la variable `i` pot ser usada fora de la funció. Sense la declaració de `global`, no es podria. Per contra `L` és una variable global del programa i es pot usar dins de la funció i fora.

De tota manera, noteu que només podem usar la variable `i` després d'haver cridat a la funció `cerca`. Per tant, encara que la visibilitat de les variables es decideix de forma estàtica, la seva creació és dinàmica i aixi també la possibilitat d'usar-les.

Finalment, per remarcar el fet de que si el primer ús d'una variable és una assignació, llavors es considera la variable declarada com a local, si poseu

```
def cerca(x):
   global i
   i = 0;
   L = []
   while i<len(L):
      if x==L[i]: break
      i += 1
   else: i = -1

L = [3,5,1,8,11]
```

```
n = 3
cerca(n)
print(i)
print(L)
```

i executeu el codi, com que la `L` és local a `cerca`, el resultat mostrat és

```
-1
[3,5,1,8,11]
```

6 Els Generadors i el `yield`

Els generadors són una subclasse d'iteradors, que es construeixen usant la funció `yield`. Per exemple

```
def map( fun, l ):
   "retorna un generador amb l'aplicació de la funció a tot els elements"
   for x in l:
     yield fun(x)
```

En aquest cas `l` pot ser de qualsevol tipus iterable, però el resultat no és una llista, sinó un generador. Així, podem fer

```
for x in map( lambda arg: arg+1, [3,4,5] ):
  print(x, end=" ")
```

però no

```
print(map( lambda arg: arg+1, [3,4,5] ))
```

ja que els generadors no tenen la operació d'imprimir, ni el `len`, ...

No hem de confondre els generadors amb llistes (són tipus diferents). Si volem aplicar operacions de llistes primer hem de fer una conversió.

```
print(list(map( lambda arg: arg+1, [3,4,5] )))
```

7 Exercicis

Feu programes, definint funcions i usant estructures de dades com les descrites en el document, per resoldre els problemes següents.

7.1 Programes amb llistes

Utilitzant com a referència els programes vistos a classe resoleu els exercicis següents:

1. Feu una funció que indiqui quants elements diferents té una llista.

2. Feu una funció que calculi el màxim d'una llista no buida.

3. Feu una funció que calculi la mitjana d'una llista no buida.

4. Feu una funció que aplana una llista de llistes produint una llista d'elements. Per exemple, si rep [[2,6],[8,1,4],[],[1]] torna [2,6,8,1,4,1]. Milloreu la funció per a que pugui tractar llistes amb diferents nivells i si rep [3,[4,[],[5,3]],[2,1]] torni [3, 4, 5, 3, 2, 1]. Pista: feu-la recursiva usant el `isinstance`.

5. Feu un funció que insereix un element en una llista ordenada creixentment.

6. Feu una funció que, donada una llista d'enters, retorni dues llistes, una que conté els parells i una que conté els senars.

7. Feu una funció que retorni la llista de divisors primers d'un enter estrictament positiu.

8. Programeu l'algorisme d'ordenació mergesort. Per això, programeu inicialment la mescla ordenada.

9. Programeu l'algorisme d'ordenació quicksort. Useu funcions d'ordre superior o llistes per comprensió.

7.2 Ús de funcions d'ordre superior predefinides

1. Feu una funció que calculi el producte dels elements d'una llista.

2. Feu una funció que multiplica tots el nombres parells d'una llista.

3. Feu una funció que inverteix els elements d'una llista usant el `reduce` (amb tres arguments).

4. Feu una funció que donada una llista de llistes d'elements l i un element x ens torna la llista que indica quants cops apareix x en cada llista de l.

7.3 Definició de noves funcions d'ordre superior

1. Definiu funcions d'ordre superior que hi ha en Haskell i no hi ha en Python, com ara el `zipWith`, `takeWhile`, `dropWhile`, `foldl`, `foldr`, `scanl`, etc.

2. Feu una funció `countIf` que donada una propietat i una llista, ens retorna el nombre d'elements de la llista que satisfan la propietat. Noteu que aquesta funció d'ordre superior existeix en llenguatges de tractament de fulls de càlcul com ara EXCEL.

3. Feu una funció `inserir` que donada una relació, un element i una llista ens retorna la llista amb l'element inserit segons la relació. Utilitzant aquesta funció i el `reduce`, defineix una funció que donada una relació i una llista, ordeni la llista per inserció.

7.4 Ús de llistes per comprensió

1. Feu una definició nova del `map` usant llistes per comprensió.

2. Feu una definició nova del `filter` usant llistes per comprensió.

3. Feu una funció que donades dues llistes genera la llista dels parells on el primer element és de la primera llista, el segon de la segona i el segon divideix al primer.

4. Feu una funció que, donat un nombre, genera la llista amb els seus factors (no necessàriament primers).

5. Feu una funció que donat un nombre n generi totes les ternes pitagòriques, és a dir, tripletes (a, b, c) on $a^2 + b^2 = c^2$ amb nombres entre 1 i n.

6. Donada una llista d'enters estrictament positius diferents i un nombre positiu retorneu una llista de llistes que conté totes les formes d'obtenir el nombre donat sumant elements de la llista (podeu repetir elements).