Introducció a la programació en Python

Antoni Oliver (a partir de l'obra de Raúl González Duque)

PID_00155230

Universitat Oberta de Catalunya
www.uoc.edu

Aquesta obra és llicencia sota la següent llicència Creative Commons: *Reconeixement - CompartirIgual 3.0* (by-sa): es permet l'ús comercial de l'obra i de les possibles obres derivades, la distribució de les quals s'ha de fer amb una llicència igual a la que regula l'obra original.

Índex

Introducció

En aquest mòdul presentem els fonaments de programació en Python. El mòdul és una adaptació del llibre *Python para todos* de Raúl González Duque (http://mundogeek.net/tutorial-python). S'ha seleccionat el contingut bàsic d'aquest llibre i se n'han adaptat alguns exemples. També es proposen una sèrie d'exercicis d'autoavaluació per a cada apartat, amb les seves solucions corresponents.

Aprendre a programar és important per a qualsevol usuari habitual d'ordinador. Amb uns coneixements bàsics de programació es poden automatitzar una gran quantitat de tasques habituals. No cal ser un gran expert per a realitzar programes que ens facilitin el treball diari. Per a un traductor, filòleg o lingüista, que acostuma a treballar amb una gran quantitat de textos, els coneixements de programació seran de gran valor: li permetran aprofitar millor els seus recursos i estalviar temps en el seu treball.

Hi ha una gran quantitat de llenguatges de programació i l'elecció de Python ha estat meditada. Són nombrosos els factors que han decantat l'elecció, entre els quals podem destacar:

- Python és fàcil d'aprendre i d'utilitzar, però, alhora, és potent i facilita l'elaboració de programes.

- Tot allò necessari per a poder programar en Python és gratuït, des de l'intèrpret fins als editors necessaris, que poden anar des de senzills editors de text fins a complexos entorns de desenvolupament integrats (IDE).

- És un llenguatge molt utilitzat en programari lliure, per la qual cosa podrem trobar nombrosos programes i mòduls que podrem incorporar als nostres propis desenvolupaments.

- És un llenguatge totalment orientat a objectes, tot i que permet treballar amb altres paradigmes de programació.

- És un llenguatge multiplataforma, per la qual cosa els nostres programes podran funcionar fàcilment en diferents sistemes operatius.

Es podrien enumerar moltes més qualitats, però serà el propi lector qui les anirà descobrint a mesura que avanci en els seus coneixements.

Al final del mòdul s'ofereixen alguns llibres, manuals i tutorials de Python que són gratuïts. D'aquesta manera, el lector que s'hagi quedat amb ganes de saber-ne més podrà aprofundir en el coneixement d'aquest llenguatge de programació.

Objectius

En els materials didàctics d'aquest mòdul presentem els continguts i les eines imprescindibles per a assolir els objectius següents:

1) Valorar els coneixements de programació per a un traductor, filòleg o lingüista.

2) Aprendre els conceptes i instruccions bàsiques del llenguatge de programació Python.

3) Ser capaç de solucionar problemes senzills mitjançant programes escrits en Python.

1. Introducció

1.1. Què és Python?

Python és un llenguatge de programació creat per Guido van Rossum a principis dels anys 1990, el nom del qual està inspirat en el grup de còmics anglesos Monty Python. És un llenguatge similar a Perl, però amb una sintaxi molt neta i que afavoreix un codi llegible.

Es tracta d'un llenguatge interpretat o llenguatge *script*, amb tipatge dinàmic, fortament tipat, multiplataforma i orientat a objectes.

1.1.1. Llenguatge interpretat o llenguatge *script*

Un llenguatge interpretat o llenguatge *script* és aquell que s'executa utilitzant un programa intermedi anomenat *intèrpret*, en lloc de compilar el codi a llenguatge màquina que pugui comprendre i executar directament un ordinador, com passa en els llenguatges compilats.

L'avantatge dels llenguatges compilats és que la seva execució és més ràpida. Tanmateix els llenguatges interpretats són més flexibles i més portables.

Python té, no obstant això, moltes de les característiques dels llenguatges compilats, per la qual cosa es podria dir que és semiinterpretat. En Python, com en Java i molts altres llenguatges, el codi font es tradueix a un pseudocodi màquina intermedi anomenat *bytecode* la primera vegada que s'executa, que genera arxius .pyc o .pyo (*bytecode* optimitzat), que són els que s'executaran en ocasions successives.

1.1.2. Tipatge dinàmic

La característica de tipatge dinàmic es refereix a que no cal declarar el tipus de dada que ha de contenir una determinada variable, sinó que el seu tipus es determinarà en temps d'execució segons el tipus del valor al qual s'assigni, i el tipus d'aquesta variable pot canviar si se li assigna un valor d'un altre tipus.

1.1.3. Fortament tipat

No es permet tractar una variable com si fos d'un tipus diferent al que té; és necessari convertir prèviament, de manera explícita, aquesta variable al nou tipus. Per exemple, si tenim una variable que conté un text (variable de tipus cadena o *string*) no podrem tractar-la com un nombre (com ara sumar la cadena "9" i el número 8). En altres llenguatges el tipus de la variable canviaria per a adaptar-se al comportament esperat, malgrat que això és més propens a errors.

1.1.4. Multiplataforma

L'intèrpret de Python és disponible en multitud de plataformes (UNIX, Solaris, Linux, DOS, Windows, OS/2, Mac OS, etc.). Si no utilitzem llibreries específiques de cada plataforma el nostre programa podrà funcionar en tots aquests sistemes sense grans canvis.

1.1.5. Orientat a objectes

L'orientació a objectes és un paradigma de programació en el qual els conceptes del món real rellevants per al nostre problema es traslladen a classes i objectes en el nostre programa. L'execució del programa consisteix en una sèrie d'interaccions entre els objectes.

Python també permet la programació imperativa, la programació funcional i la programació orientada a aspectes.

1.2. Per què Python?

Python és un llenguatge que tothom hauria de conèixer. La seva sintaxi simple, clara i senzilla, el tipatge dinàmic, el gestor de memòria, la gran quantitat de llibreries disponibles i la potència del llenguatge, entre d'altres factors, fan que desenvolupar una aplicació en Python sigui senzill, molt ràpid i, el que és més important, divertit.

La sintaxi de Python és tan senzilla i propera al llenguatge natural que els programes elaborats en Python semblen pseudocodi. Per aquest motiu es tracta d'un dels millors llenguatges per a començar a programar. Python no és adequat, però, per a la programació de baix nivell o per a aplicacions en les quals el rendiment sigui crític.

Alguns casos d'èxit en l'ús de Python són Google, Yahoo, la NASA, l'empresa Industrial Light & Magic i totes les distribucions de Linux, en les quals Python cada vegada representa un tant per cent més elevat dels programes disponibles.

1.3. Instal·lació de Python

Hi ha diverses implementacions diferents de Python: CPython, Jython, IronPython, PyPy, etc.

CPython és la més utilitzada, la més ràpida i la més madura. Quan la gent parla de Python normalment es refereix a aquesta implementació. En aquest cas, tant l'intèrpret com els mòduls estan escrits en C.

Jython és la implementació en Java de Python, mentre que IronPython és la seva contrapartida en C# (.NET). El seu interès rau en el fet que utilitzant aquestes implementacions es poden utilitzar totes les llibreries disponibles per als programadors de Java i .NET.

PyPy, per últim, com haureu endevinat pel nom, es tracta d'una implementació en Python de Python.

ActivePython

Una altra bona distribució, molt recomanable si utilitzeu Windows, és ActivePython d'ActiveState. La podeu descarregar de http://www.activestate.com/Products/activepython/index.mhtml.

CPython està instal·lat per defecte en la major part de les distribucions de Linux i en les últimes versions de Mac OS. Per comprovar si està instal·lat, obriu un terminal i escriviu `python`; si està instal·lat s'iniciarà la consola interactiva de Python i obtindrem alguna cosa semblant a:

```
Python 2.5.1 (r251:54863, May 2 2007, 16:56:35)
[GCC 4.1.2 (Ubuntu 4.1.2-0ubuntu4)] on linux2
Type “help”, “copyright”, “credits” or “license” for more
information.
>>>
```

La primera línia ens indica la versió de Python que tenim instal·lada. Al final podem veure l'indicador d'ordres (>>>) que ens mostra que l'intèrpret està esperant codi per part de l'usuari. Podem sortir escrivint `exit()` o prémer **Control+D**.

Si no mostra una cosa semblant no us preocupeu, instal·lar Python és molt senzill. Podeu descarregar la versió corresponent al vostre sistema operatiu des de la web de Python*. Hi ha instal·ladors per a Windows i Mac OS. Si utilitzeu Linux és molt probable que pogueu instal·lar-lo usant l'eina de gestió de paquets de la distribució, tot i que també podem descarregar l'aplicació compilada des de la web de Python.

***http://www.python.org/download/**

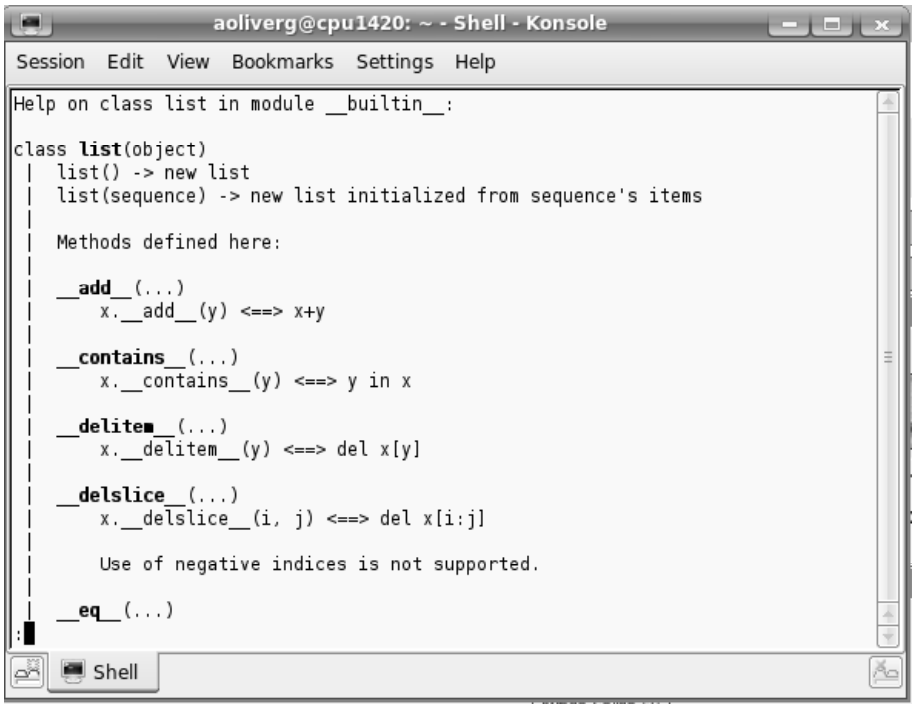
1.4. Eines bàsiques

Hi ha dues maneres d'executar codi Python. Podem escriure línies de codi en l'intèrpret i obtenir una resposta de l'intèrpret per a cada línia (sessió interactiva) o bé podem escriure el codi d'un programa en un arxiu de text i executar-lo.

Per a executar l'intèrpret interactiu simplement s'ha d'escriure `Python` en la línia d'ordres, escriure les instruccions després del símbol >>> i prémer Enter després de cada instrucció. Per accedir a la línia d'ordres sota Linux només cal obrir un terminal. En Windows la línia d'ordres es troba a Inicia > Accessoris > Indicador d'ordres (o alguna cosa semblant, ja que pot canviar lleugerament segons la versió de Windows que utilitzeu). Si no ho trobeu, també podreu accedir-hi fent Inicia > Executa i posant `cmd` a la pantalla que apareix.

L'intèrpret interactiu serà de gran utilitat per a provar instruccions o petits programes abans d'escriure'ls en un editor de text. A més, l'intèrpret interactiu ens podrà donar informació sobre instruccions, classes i tipus de variables. Per exemple, si escrivim `help("list")` en l'intèrpret interactiu, ens apareixerà alguna cosa semblant a la figura 1.

Figura 1. Ajuda per a la classe llista que ofereix l'intèrpret interactiu



Per a escriure els nostres programes en Python només necessitarem un editor de text. Sota Linux una bona opció és Emacs, mentre que sota Windows es pot utilitzar el bloc de notes. A mesura que anem fent programes més complexos serà millor passar a algun tipus d'editor específic per a programació, com comentem al paràgraf següent.

Pythonwin

Si utilitzeu ActivePython per a Windows podreu escriure els programes a l'editor Pythonwin que s'instal·la per defecte.

En el camp d'IDE i d'editors de codi gratuïts, PyDEV* s'alça com a cap de sèrie. PyDEV és un connector per a Eclipse que permet utilitzar aquest IDE multiplataforma per a programar en Python. Compta amb compleció automàtica de codi (amb informació sobre cada element), ressaltat de sintaxi, un depurador gràfic, ressaltat d'errors, explorador de classes, formateig del codi, refactorització, etc. Sens dubte, és l'opció més completa, sobretot si instal·lem les extensions comercials, tot i que necessita una quantitat important de memòria i no és del tot estable.

*http://pydev.sourceforge.net/

Altres opcions gratuïtes a considerar són SPE o Stani's Python Editor*, Eric**, BOA Constructor*** o, fins i tot, Emacs o vim.

*http://sourceforge.net/projectes/spe
**http://die-offenbachs.de/eric/
***http://boa-constructor.sourceforge.net

Si no us importa desemborsar alguns diners, Komodo* i Wing IDE** són també molt bones opcions, amb moltes característiques interessants, com PyDEV, però molt més estables i robusts. A més, si desenvolupeu programari lliure no comercial podeu contactar amb Wing Ware i obtenir, amb una mica de sort, una llicència gratuïta per a Wing IDE Professional.

*http://www.activestate.com/komodo_ide
**http://www.wingware.com

1.5. Exercicis d'autoavaluació

1. Comproveu si teniu instal·lat l'intèrpret de Python al vostre ordinador. Si no l'hi teniu, trieu un distribució i instal·leu-la. Un cop instal·lada torneu a realitzar la comprovació.

2. Instal·leu un editor o IDE específic per a Python i executeu-lo. Fixeu-vos en les opcions que ofereix i escriviu el següent fragment de codi. Passa alguna cosa especial?

```
fav = "macarrons"
if fav == "macarrons":
    print "Tens bon gust!"
    print "Gracies"
```

Accents als programes

Amb Python és perfectament possible fer servir qualsevol caràcter als nostres programes, inclosos els caràcters accentuats. Com que encara no hem explicat com fer-ho, i per evitar problemes, en els primers programes que presentem evitarem fer servir accents i altres caràcters especials.

1.6. Solució als exercicis d'autoavaluació

1. Simplement heu d'obrir un terminal o el símbol del sistema i escriure `Python`. Si està instal·lat es posarà en marxa l'intèrpret interactiu.

2. Si treballeu amb Linux utilitzeu l'aplicació d'instal·lació de programes i busqueu "Python" segurament apareixeran Boa Constructor, Eric Python IDE o PyPe. Instal·leu qualsevol d'aquests. Si treballeu amb Windows i heu escollit la distribució ActivePython tindreu un editor instal·lat per defecte. Si no, podeu instal·lar el Stani's Python Editor d'http://sourceforge.net/projectes/spe.

Quan escriviu el codi segurament es marcarà la sintaxi en colors i possiblement es numeraran les línies. En alguns editors s'activaran opcions de compleció automàtica de codi.

2. El meu primer programa en Python

Com comentàvem en l'apartat anterior, hi ha dues maneres d'executar codi Python: en una sessió interactiva (línia a línia) amb l'intèrpret o bé de la forma habitual, escrivint el codi en un arxiu de codi font i executant-lo.

El primer programa que escriurem en Python és el clàssic "Hola Mon", que en aquest llenguatge és tan simple com:

```
print "Hola Mon"
```

Provarem primer en l'intèrpret. Executeu Python, escriviu la línia anterior i premeu Enter. L'intèrpret respondrà mostrant a la consola el text "Hola Món".

Ara crearem un fitxer de text amb el codi anterior, de manera que poguem distribuir el nostre petit gran programa entre els nostres amics. Obriu el vostre editor de text preferit o bé l'IDE que heu triat i copieu la línia anterior. Deseu el fitxer com a hola.py, per exemple.

Executar aquest programa és tan senzill com indicar a l'intèrpret de Python el nom de l'arxiu que ha d'executar:

```
python hola.py
```

Vegem, però, com simplificar-ho encara més.

Si utilitzeu Windows els arxius .py ja estaran associats a l'intèrpret de Python, per la qual cosa només cal fer doble clic sobre el fitxer per a executar el programa. Tanmateix, com que aquest programa només imprimeix un text a la consola, l'execució és massa ràpida per a poder-la veure. Per a posar-hi remei, afegim una nova línia que esperi l'entrada de dades per part de l'usuari.

```
print "Hola Mon"
raw_input()
```

D'aquesta manera es mostrarà una consola amb el text "Hola Mon" fins que premem Enter.

Si utilitzeu Linux (o un altre Unix) per a aconseguir aquest comportament, és a dir, perquè el sistema operatiu obri l'arxiu .py amb l'intèrpret adequat, cal afegir una nova línia al principi de l'arxiu:

```
#!/usr/bin/python

print "Hola Mon"
raw_input()
```

Aquesta línia es coneix en el món Unix com a *shebang*, *hashbang* o *sharp-bang*. El parell de caràcters `#!` indica al sistema operatiu que aquest *script* s'ha d'executar utilitzant l'intèrpret especificat a continuació. D'això es desprèn, evidentment, que si aquest no és el camí on està instal·lat el nostre intèrpret de Python, caldrà canviar-lo.

Una altra opció és utilitzar el programa **env** (d'*environment*, 'entorn') per a preguntar al sistema la ruta cap a l'intèrpret de Python, de manera que els nostres usuaris no tinguin cap problema si es dóna el cas que el programa no estigués instal·lat en aquesta ruta:

```
#!/usr/bin/env python

print "Hola Mon"
raw_input()
```

Evidentment, a banda d'afegir el *shebang*, haurem de donar permisos d'execució al programa.

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

I llestos. Si fem doble clic el programa s'executarà i mostrarà una consola amb el text "Hola Mon", com en el cas de Windows.

També podríem executar el programa des de la consola com si es tractés d'un executable qualsevol:

```
./hola.py
```

3. Tipus bàsics

En Python els tipus bàsics es divideixen en:

Tipus bàsics

Quan parlem de tipus bàsics ens referim als tipus de variables elementals en Python. En l'apartat següent veurem altres tipus més avançats: les col·leccions.

- Números, com poden ser 3 (enter), 15.57 (de coma flotant) o 7 + 5j (complex).
- Cadenes de text, com "Hola Món".
- Valors booleans: True ('cert') i False ('fals').

Creem ara un parell de variables a tall d'exemple. Una de tipus cadena i una de tipus enter:

```
# aixo es una cadena
c = "Hola Mon"
# i aixo un enter
e = 23
# podem comprovar-ho amb la funcio type
type(c)
type(e)
```

Com veieu, en Python, a diferència de molts altres llenguatges, no es declara el tipus de la variable en el moment de crear-la. En Java, per exemple, cal escriure:

```
String c = "Hola Mon";
int e = 23;
```

Aquest petit exemple també ens ha servit per a presentar els comentaris en línia en Python: cadenes de text que comencen amb el caràcter # i que Python ignora totalment. Hi ha més tipus de comentaris, dels quals parlarem més endavant.

Comentaris

És important introduir comentaris en els nostres programes que ajudin, a nosaltres o a altres usuaris, a comprendre el contingut del programa.

3.1. Números

Com dèiem, en Python es poden representar nombres enters, reals i complexos.

3.1.1. Enters

Els nombres enters són els nombres positius o negatius que no tenen decimals (i també el zero).

En Python es poden representar mitjançant el tipus `int` (d'*integer*, 'enter') o el tipus `long` ('llarg'). L'única diferència és que el tipus `long` permet emmagatzemar números més grans. És aconsellable no utilitzar el tipus `long` a menys que sigui necessari, per tal de no malbaratar memòria. El tipus `long` de Python permet emmagatzemar números de qualsevol precisió, limitats només per la memòria disponible a la màquina.

En assignar un número a una variable aquesta passarà a tenir tipus `int`, a menys que el nombre sigui tan gran per a requerir l'ús del tipus `long`.

3.1.2. Reals

Els nombres reals són els que tenen decimals.

En Python els nombres reals s'expressen mitjançant el tipus `float`. Per a representar un nombre real en Python s'escriu primer la part sencera, seguida d'un punt i, per últim, la part decimal.

```
real = 0.2703
```

També es pot utilitzar la notació científica i afegir una `e` (d'exponent) per a indicar un exponent en base 10. Per exemple:

```
real = 0.1e-3
```

seria equivalent a $0.1 \times 10^{-3} = 0.1 \times 0.001 = 0.0001$

3.1.3. Complexos

Els nombres complexos són aquells que tenen part imaginària.

Si no sabíeu de la seva existència, és probable que mai ho els hagueu de necessitar, per la qual cosa podeu prescindir d'aquest subapartat tranquil·lament. De fet, la major part de llenguatges de programació no tenen aquest tipus de número, malgrat que siguin molt utilitzats per enginyers i científics en general.

En el cas que necessiteu utilitzar nombres complexos, o simplement en tingueu curiositat, us direm que aquest tipus, anomenat `complex` en Python, també s'emmagatzema usant coma flotant, ja que aquests números són una extensió dels nombres reals. En concret, s'emmagatzemen en una estructura de C, composta per dues variables de tipus `double`; una serveix per a emmagatzemar la part real i l'altra, per a la part imaginària.

Els números complexos en Python es representen de la següent forma:

```
complex = 2.1 + 7.8j
```

3.1.4. Operadors aritmètics

Vegem ara què podem fer amb els nostres nombres utilitzant els operadors per defecte. Per a operacions més complexes podem recórrer al mòdul `math`.

Taula 1

Operador	Descripció	Exemple
+	Suma	r = 3 + 2 # r és 5
-	Resta	r = 4 - 7 # r és -3
-	Negació	r = -7 # r és -7
*	Multiplicació	r = 2 * 6 # r és 12
**	Exponenciació	r = 2 ** 6 # r és 64
/	Divisió	r = 3.5 / 2 # r és 1.75
//	Divisió entera	r = 3.5 // 2 # r és 1.0
%	Mòdul	r = 7 % 2 # r és 1

És possible que tingueu dubtes sobre com funciona l'operador mòdul, i quina és la diferència entre la divisió i la divisió entera.

L'operador de mòdul retorna la resta de la divisió entre els dos operands. En l'exemple, 7/2 seria 3, amb 1 de resta i, per tant, el mòdul és 1.

La diferència entre la divisió i la divisió entera no és altra que la que indica el seu nom. A la divisió el resultat que es retorna és un nombre real, mentre que a la divisió entera el resultat que es retorna n'és només la part entera.

No obstant això, cal tenir en compte que si fem servir dos operands enters, Python determinarà que volem que la variable resultat també sigui un enter, per la qual cosa el resultat de, per exemple, 3/2 i 3//2 seria el mateix: 1.

Si volguéssim obtenir els decimals necessitaríem que almenys un dels operands fos un nombre real, cosa que podríem obtenir bé indicant els decimals

```
r = 3.0 / 2
```

o bé utilitzant la funció `float` (no cal que sapigueu què significa el terme *funció* ni que recordeu aquesta forma; ho veurem una mica més endavant):

```
r = float(3) / 2
```

Això és així perquè quan es barregen tipus de nombres diferents, Python converteix tots els operands al tipus més complex d'entre els tipus dels operands.

3.2. Cadenes

Les cadenes no són més que text tancat entre cometes simples ('cadena') o dobles ("cadena"). Dins de les cometes es poden afegir caràcters especials utilitzant \, com ara $\backslash n$, el caràcter de línia nova, o $\backslash t$, el de tabulació.

Una cadena pot estar precedida pel caràcter `u` o pel caràcter `r`, que indiquen, respectivament, que es tracta d'una cadena que utilitza codificació Unicode o d'una cadena *raw* (en anglès 'en brut'). Les cadenes *raw* es distingeixen de les normals en què els caràcters que utilitzen la barra invertida (\) no es substitueixen per les seves contrapartides. Això és especialment útil, per exemple, per a les expressions regulars, com veurem en l'apartat corresponent.

```
unicode = u"äóè"
raw = r"\n"
```

També és possible tancar una cadena entre cometes triples (simples o dobles). D'aquesta manera podrem escriure el text en diverses línies, i en imprimir la cadena, es respectaran els salts de línia que vam introduir sense haver de recórrer al caràcter $\backslash n$, així com les cometes, sense haver d'escapar.

```
triple = """primera línia
això es veurà en una altra línia"""
```

Les cadenes també admeten operadors com `+`, que funciona realitzant una concatenació de les cadenes utilitzades com a operands, i `*`, amb què es repe-

teix la cadena tantes vegades com ho indiqui el número utilitzat com a segon operand.

```
a = “un”
b = “dos”

c = a + b # c és “undos”

c = a * 3 # c és “ununun”
```

Una altra característica interessant de les cadenes és que podem accedir a determinades posicions de la cadena utilitzant índexs (on el 0 indica la primera posició):

```
cadena=“Hola“
print cadena[0] #escriu H
print cadena[2] #escriu l
```

En el proper apartat, al subapartat dedicada a les llistes, veurem totes les possibilitats que ens ofereixen els índexs.

3.3. Booleans

Una variable de tipus booleà només pot tenir dos valors: True (cert) i False (fals). Aquests valors són especialment importants per a les expressions condicionals i els bucles, com veurem més endavant.

En realitat, el tipus `bool` (el tipus dels booleans) és una subclasse del tipus `int`. Pot ser que això us sembli que no té massa sentit si no coneixeu els termes relacionats amb l'orientació a objectes, que veurem més endavant, però tampoc és massa important.

Aquests són els diferents tipus d'operadors amb què podem treballar amb valors booleans, els anomenats *operadors lògics* o *condicionals*:

Taula 2

Operador	Descripció	Exemple
and	es compleix *a* i *b*?	r = True and False # r és False
or	es compleix *a* o *b*?	r = True or False # r és True
not	No *a*	r = not True # r és False

Els valors booleans són, a més a més, el resultat d'expressions que utilitzen operadors relacionals (comparacions entre valors):

Taula 3

Operador	Descripció	Exemple
==	són iguals *a* i *b*?	r = 5 == 3 # r és False
!=	són diferents *a* i *b*?	r = 5 != 3 # r és True
<	és *a* menor que *b*?	r = 5 < 3 # r és False
>	és *a* major que *b*?	r = 5 > 3 # r és True
<=	és *a* menor o igual que *b*?	r = 5 <= 5 # r és True
>=	és a major o igual que b?	r = 5 >= 3 # r és True

3.4. Exercicis d'autoavaluació

1. Escriviu un programa que, a partir del preu sense IVA de tres productes, calculi la suma, l'IVA total i l'import a pagar (és a dir, el total més l'IVA). Trieu el percentatge d'IVA que preferiu (7 % o 16 %).

2. Quina és la resta de la divisió entera de 43 partit per 5?

3. Utilitzeu l'operador + per a cadenes i obtingueu, a partir de l'arrel "cant-", l'acabament d'infinitiu "-ar" i l'acabament de primera persona del singular del present d'indicatiu "-o", les formes corresponents a l'infinitiu i a la primera persona del singular del present d'indicatiu. Considereu les variables: `arrel` (`cant`), `tinf` (`ar`) i `t1p` (`o`).

3.5. Solució als exercicis d'autoavaluació

1. Una possible solució (recordeu que poden haver-hi diverses i que els imports són lliures) és la següent:

```
a=3.21
b=4.12
c=1.15
suma=a+b+c
iva=suma*0.16
total=suma+iva
print suma
print iva
print total
```

2. El resultat és 3. Utilitzarem l'operador mòdul (%):

```
print 43 % 5
```

3. Simplement cal "sumar", és a dir, concatenar l'arrel amb les terminacions:

```
arrel='cant'
tinf='ar'
t1p='o'
print arrel+tinf
print arrel+t1p
```

4. Col·leccions

Al apartat anterior vam veure alguns tipus bàsics, com els números, les cadenes de text i els booleans. En aquest apartat veurem alguns tipus de col·leccions de dades: llistes, tuples i diccionaris.

Els tipus de dades que presentem en aquest apartat es denominen col·leccions perquè poden emmagatzemar més d'un valor sota el mateix nom.

4.1. Llistes

La llista és un tipus de col·lecció ordenada.

Una llista seria equivalent a allò que en altres llenguatges de programació es coneix com a matrius o vectors. Les llistes poden contenir qualsevol tipus de dada: números, cadenes, booleans, etc. i també llistes. Crear una llista és tan senzill com indicar entre claudàtors, i separats per comes, els valors que volem incloure-hi:

```
l = [22, True, "una llista", [1, 2]]
```

Podem accedir a cadascun dels elements de la llista escrivint el nom de la llista i indicant l'índex de l'element entre claudàtors. Tingueu en compte que l'índex del primer element de la llista és 0, no 1:

```
l = [11, False]
la_meva_var = l[0] # la_meva_var val 11
```

Si volem accedir a un element d'una llista inclosa dins d'una altra llista haurem d'utilitzar dues vegades aquest operador, la primera per a indicar a quina posició de la llista exterior volem accedir, i la segona per a seleccionar l'element de la llista interior:

```
l = [“una llista”, [1, 2]]
la_meva_var = l[1][0] # la_meva_var val 1
```

També podem utilitzar aquest operador per a modificar un element de la llista si el posem a la part esquerra d'una assignació:

```
l = [22, True]
l[0] = 99 # Amb aixo l valdra [99, True]
```

L'ús dels claudàtors per a accedir i modificar els elements d'una llista és comú en molts llenguatges, però Python ens proporciona diverses sorpreses molt agradables. Una curiositat sobre l'operador [] de Python és que podem utilitzar també nombres negatius. Si s'utilitza un nombre negatiu com a índex, significa que l'índex comença a comptar des del final i cap a l'esquerra; és a dir, amb [-1] accediria a l'últim element de la llista, amb [-2] al penúltim, amb [-3], a l'avantpenúltim, i així successivament.

Una altra cosa inusual és allò que en Python es coneix com a *slicing* o partició, i que consisteix a ampliar aquest mecanisme per a permetre seleccionar porcions de la llista. Si, en lloc d'un número escrivim dos números, un d'inici i un de final, separats per dos punts (`inici:fi`) Python interpretarà que volem una llista que vagi des de la posició `inici` a la posició `fi`, sense incloure aquesta última. Si escrivim tres números (`inici:fi:salt`) en comptes de dos, el tercer s'utilitza per a determinar cada quantes posicions cal afegir un element a la llista.

```
l = [99, True, “una llista”, [1, 2]]
la_meva_var = l[0:2] # la_meva_var val [99, True]
la_meva_var = l[0:4:2] # la_meva_var val [99, “una llista”]
```

Els números negatius també es poden utilitzar en un *slicing* amb el mateix comportament que s'ha comentat anteriorment. Cal esmentar també que no cal indicar el principi i el final del *slicing*; si aquests s'ometen, s'utilitzaran per defecte les posicions d'inici i final de la llista, respectivament:

```
l = [99, True, “una llista”]
la_meva_var = l[1:] # la_meva_var val [True, “una llista”]
la_meva_var = l[:2] # la_meva_var val [99, True]
la_meva_var = l[:] # la_meva_var val [99, True, “una llista”]
la_meva_var = l[::2] # la_meva_var val [99, “una llista”]
```

També podem utilitzar aquest mecanisme per a modificar la llista:

```
l = [99, True, “una llista”, [1, 2]]
l[0:2] = [0, 1] # l val [0, 1, “una llista”, [1, 2]]
```

Fins i tot podem modificar la mida de la llista si la llista de la part dreta de l'assignació té una mida menor o major que la selecció de la part esquerra de l'assignació:

```
l[0:2] = [False] # l val [False, “una llista”, [1, 2]]
```

En qualsevol cas, les llistes ofereixen mecanismes més còmodes per a ser modificades a través de les funcions de la classe corresponent, tot i que no veurem aquests mecanismes fins més endavant, després d'explicar què són les classes, els objectes i les funcions.

4.2. Tuples

Una tupla és una llista immutable. Una tupla no pot canviar de cap manera un cop creada.

Tot el que hem explicat sobre les llistes s'aplica també a les tuples, a excepció de la forma de definir-les, ja que s'utilitzen parèntesis en lloc de claudàtors.

```
t = (1, 2, True, “python”)
```

En realitat, el constructor de les tuples és la coma, no el parèntesi, però l'intèrpret mostra els parèntesis, i nosaltres els hauríem d'utilitzar per claredat.

```
>>> t = 1, 2, 3
>>> type(t)
type “tuple”
```

A més s'ha de tenir en compte que cal afegir una coma per a tuples d'un sol element, per a diferenciar-les d'un element entre parèntesi.

```
>>> t = (1)
>>> type(t)
type “int”
```

```
>>> t = (1,)
>>> type(t)
type “tuple”
```

Per a referir-nos als elements d'una tupla, com en una llista, es fa servir l'operador []:

```
la_meva_var = t[0] # la_meva_var es 1
la_meva_var = t[0:2] # la_meva_var es (1, 2)
```

Podem utilitzar l'operador [], ja que les tuples, igual que les llistes, formen part d'un tipus d'objectes anomenats *seqüències*. Ara ens agradaria fer un petit incís per a indicar que les cadenes de text també són seqüències, de manera que no us estranyi el fet que poguem fer coses com aquestes:

```
c = “hola mon“
c[0] # h
c[5:] # mon
c[::3] # hao
```

Tornant al tema de les tuples, la seva diferència amb les llistes rau en el fet que les tuples no tenen aquests mecanismes de modificació a través de funcions, tan útils, de què parlàvem al final del subapartat anterior. A més, són immutables, és a dir, els seus valors no es poden modificar un cop creades, i tenen una mida fixa. A canvi d'aquestes limitacions les tuples són més “lleugeres” que les llistes, de manera que si l'ús que volem donar a una col·lecció és molt bàsic, es poden utilitzar tuples en lloc de llistes i, així, estalviar memòria.

4.3. Diccionaris

Els diccionaris, també anomenats *matrius associatives*, reben aquest nom perque són col·leccions que relacionen una clau i un valor.

Per exemple, vegem un diccionari de pel·lícules i directors:

```
d = {’Love Actually’: ’Richard Curtis’,
’Kill Bill’: ’Tarantino’,
’Amélie’: ’Jean-Pierre Jeunet’}
```

El primer valor és la clau i el segon és el valor associat a la clau. Com a clau podem fer servir qualsevol valor immutable: podríem fer servir números, cade-

nes, booleans, tuples, etc., però no llistes ni diccionaris, ja que són mutables. Això és així perquè els diccionaris s'implementen com a taules *hash* i, a l'hora d'introduir un nou parell clau-valor en el diccionari, es calcula el *hash* de la clau per a després trobar la clau ràpidament. Si es modifiqués l'objecte clau després d'haver-se introduït en el diccionari, el seu *hash* també canviaria i no es podria trobar.

La diferència principal entre els diccionaris i les llistes o les tuples és que als valors emmagatzemats en un diccionari s'accedeix no mitjançant seu índex, ja que de fet no tenen ordre, sinó mitjançant la seva clau, utilitzant de nou l'operador [].

```
d['Love Actually'] # retorna 'Richard Curtis'
```

Igual que en llistes i tuples, també es pot utilitzar aquest operador per a reassignar valors.

```
d['Kill Bill'] = 'Quentin Tarantino'
```

Tanmateix, en aquest cas no es pot utilitzar un *slicing*, entre altres coses perquè els diccionaris no són seqüències, sinó *mappings* ('mapatges').

4.4. Exercicis d'autoavaluació

1. Creeu una llista anomenada `frase` que contingui les paraules de la frase "Avui fa un dia molt bonic". Practiqueu amb les operacions descrites al subapartat dedicat a les llistes. Què passa si intenteu escriure `frase[6]`? I `frase[-1]`? Com podeu recuperar la segona i la tercera lletres?

2. Ara creeu una cadena anomenada `frase2` que contingui la mateixa oració. Què passa si intenteu escriure `frase2[6]`? I `frase2[-1]`? Com podeu recuperar la segona i la tercera lletres?

3. Creeu un diccionari que contingui les paraules *house*, *dog*, *car* i *book* amb les seves corresponents traduccions al català i practiqueu les operacions amb diccionaris.

4.5. Solució als exercicis d'autoavaluació

1. Simplement s'ha d'escriure el codi següent (en l'intèrpret interactiu és suficient) i observar els resultats. En un dels casos es produeix un error perquè el valor està fora de rang.

```
frase=['Avui','fa','un','dia','molt','bonic']
print frase[6]
print frase[-1]
print frase[1:3]
```

2. Simplement s'ha d'escriure

```
frase='Avui fa un dia molt bonic'
print frase[6]
print frase[-1]
print frase[1:3]
```

3. Per a crear el diccionari s'ha d'escriure (el nom de la variable pot ser un altre):

```
d={'house':'casa','dog':'gos','car':'cotxe','book':'llibre'}
```

Podem provar coses de l'estil

```
print d
prind d['book']
```

5. Control de flux

En aquest apartat veurem els condicionals i els bucles.

5.1. Sentències condicionals

Si un programa no fos més que una llista d'ordres a executar de manera seqüencial, una per una, no tindria molta utilitat.

Els condicionals ens permeten comprovar condicions i fer que el nostre programa es comporti d'una forma o altra, executar un fragment de codi o un altre, en funció d'aquesta condició.

Aquí és on es manifesta la importància del tipus booleà i dels operadors lògics i relacionals que vam aprendre a l'apartat sobre els tipus bàsics de Python.

5.1.1. if

La manera més simple d'un estament condicional és un `if` (en anglès 'si') seguit de la condició a avaluar, dos punts (:) i, a la següent línia i sagnat, el codi a executar en cas que es compleixi aquesta condició.

```
fav = “macarrons“
if fav == “macarrons“:
    print “Tens bon gust!“
    print “Gracies“
```

Com veieu, és força senzill. Això sí, assegureu-vos que sagneu el codi tal com s'ha fet en l'exemple, és a dir, assegureu-vos de prémer la tecla de tabulació abans de les dues ordres `print`, atès que aquesta és la forma que té Python de saber que la vostra intenció és que les dues instruccions `print` s'executin només en el cas que es compleixi la condició, i no que s'imprimeixi la primera cadena si es compleix la condició i l'altra sempre, cosa que s'expressaria així:

```
if fav == “macarrons“:
    print “Tens bon gust!“
print “Gracies“
```

En altres llenguatges de programació els blocs de codi es determinen tancant-los entre claus, i el sagnat no és més que una bona pràctica perquè sigui més fàcil seguir el flux del programa amb un sol cop d'ull. Tanmateix, com ja hem comentat, en Python es tracta d'una obligació i no d'una elecció. D'aquesta manera s'obliga als programadors a sagnar seu codi perquè sigui més fàcil de llegir.

Recordeu que en Python el sagnat és important i no és una simple qüestió d'estil com en altres llenguatges de programació.

5.1.2. if ... else

Veurem ara un condicional una mica més complicat. Què faríem si volguéssim que s'executessin unes certes ordres en el cas que la condició no es compleixi? Sens dubte podríem afegir un altre `if` que tingués com a condició la negació de la primera:

```
if fav == “macarrons“:
    print “Tens bon gust!“
    print “Gracies“
if fav != “macarrons“:
    print “Doncs a mi m'agraden mes els macarrons“
```

Però el condicional té una segona construcció molt més útil:

```
if fav == “macarrons“:
    print “Tens bon gust!"
    print “Gracies“
else:
    print “Doncs a mi m'agraden mes els macarrons“
```

Veiem que la segona condició es pot substituir per un `else` (en anglès 'en cas contrari'). Si llegim el codi veiem que té força sentit: "si `fav` és igual a `macarrons`, imprimeix això i això, sinó, imprimeix això altre".

5.1.3. if ... elif ... elif ... else

Encara queda una construcció més per veure, que és la que fa ús del `elif`.

```
if numero < 0:
    print 'Negatiu'
elif numero > 0:
    print 'Positiu'
else:
    print 'Zero'
```

`elif` és una contracció d'*`else if`*, per tant, `elif numero > 0` es pot llegir com "sinó, si el número és major que 0". És a dir, primer s'avalua la condició de l'`if`. Si és certa, s'executa el seu codi i es continua executant el codi posterior al condicional; si no es compleix, s'avalua la condició de l'`elif`. Si es compleix la condició de l'`elif` s'executa el seu codi i es continua executant el codi posterior al condicional; si no es compleix i hi ha més d'un `elif` es continua amb el següent en ordre d'aparició. Si no es compleix la condició de l'`if` ni de cap dels `elif`, s'executa el codi de l'`else`.

5.1.4. A if C else B

També hi ha una construcció similar a l'operador `?` d'altres llenguatges, que no és més que una forma compacta d'expressar un `if else`. En aquesta construcció s'avalua el predicat C i es torna A si es compleix o B si no es compleix: `A if C else B`. Vegem-ne un exemple:

```
var = "parell" if (num % 2 == 0) else "senar"
```

I això és tot. Si coneixeu altres llenguatges de programació podríeu esperar que ara us parléssim del `switch`, però en Python no existeix aquesta construcció, que es podria emular amb un simple diccionari, de manera que passem directament als bucles.

5.2. Bucles

Mentre que els condicionals ens permeten executar diferents fragments de codi en funció de certes condicions, els bucles ens permeten executar un mateix fragment de codi un cert nombre de vegades, sempre que es compleixi una determinada condició.

5.2.1. while

El bucle `while` ('mentre') executa un fragment de codi mentre es compleixi una condició.

```
edat = 0
while edat < 18:
    edat = edat + 1
    print "Felicitats, tens " + str(edat)
```

La variable `edat` comença valent 0. Com que la condició que l'edat sigui mes petita que 18 és certa (0 és més petit que 18), s'entra dins del bucle. S'incrementa l'edat en 1 i s'imprimeix el missatge que informa a l'usuari que ha fet

un any. Recordeu que l'operador + per a les cadenes, les concatena. És necessari fer servir la funció `str` (de *string*, 'cadena') per a crear una cadena a partir del número, ja que no podem concatenar números i cadenes. Tanmateix, ja comentarem això més endavant en propers apartats.

Ara es torna a avaluar la condició i 1 continua essent menor que 18, per la qual cosa es torna a executar el codi que augmenta l'edat en un any i imprimeix l'edat a la pantalla. El bucle continuarà funcionant fins que l'edat sigui igual a 18, moment en què la condició deixarà de complir-se i el programa continuaria executant les possibles instruccions següents al bucle.

Ara imaginem que oblidem escriure la instrucció que augmenta l'edat. En aquest cas, mai no s'arribaria a la condició que l'edat fos igual o més gran que 18, sempre seria 0 i el bucle continuaria indefinidament escrivint en pantalla "Felicitats, tens 0". Això és el que es coneix com a bucle infinit.

Tanmateix, hi ha situacions en les quals un bucle infinit és útil. Per exemple, vegem un petit programa que repeteix tot allò que l'usuari digui fins que escrigui "adeu".

```
while True:
    entrada = raw_input(" > ")
    if entrada == "adeu":
        break
    else:
       print entrada
```

Utilitzem la funció `raw_input` per a obtenir allò que l'usuari escriu en pantalla. No cal que sapigueu què és una funció ni com funciona exactament, simplement accepteu per ara que a cada iteració del bucle la variable `entrada` contindrà allò que l'usuari ha escrit fins prémer la tecla Enter.

Comprovem a continuació si el que ha escrit l'usuari era "adeu", cas en què s'executa l'ordre `break`, o si era qualsevol altra cosa, cas en què s'imprimeix en pantalla allò que l'usuari ha escrit. La paraula clau `break` ('trencar') provoca la sortida del bucle. Ara bé, aquest bucle també es podria haver escrit de la manera següent:

```
sortir = False
while not sortir:
    entrada = raw_input()
    if entrada == "adeu":
       sortir = True
    else:
       print entrada
```

però l'exemple anterior ens ha servit per veure com funciona `break`.

Una altra paraula clau que ens podem trobar dins dels bucles és `continue` ('continuar'). Com haureu endevinat, no fa altra cosa que passar directament a la següent iteració del bucle.

```
edat = 0
while edat < 18:
    edat = edat + 1
    if edat % 2 == 0:
        continue
    print “Felicitats, tens “ + str(edat)
```

Com podeu veure, aquesta és una petita modificació del nostre programa de felicitacions. En aquesta ocasió hem afegit un `if` que comprova si l'edat és un nombre parell; en aquest cas saltem a la propera iteració en lloc d'imprimir el missatge. És a dir, amb aquesta modificació el programa només imprimiria felicitacions quan l'edat fos un nombre senar.

5.2.2. for ... in

A aquells que tingueu experiència prèvia amb segons quins llenguatges, aquest bucle us sorprendrà gratament. En Python `for` s'utilitza com a forma genèrica d'iterar sobre una seqüència. I com a tal intenta facilitar-ne l'ús per a aquesta finalitat.

Aquest és l'aspecte d'un bucle `for` en Python:

```
sequencia = [“un”, “dos”, “tres”]
for element in sequencia:
    print element
```

Com hem dit, els `for` s'utilitzen en Python per a recórrer seqüències, de manera que farem servir un tipus seqüència, com la llista, per al nostre exemple.

Llegim la capçalera del bucle com si estigués escrita en llenguatge natural: "per a cada `element` de `sequencia`". I això és exactament el que fa el bucle: per a cada element que tinguem a la seqüència, executa aquestes línies de codi.

El que fa la capçalera del bucle és obtenir el següent element de la seqüència `sequencia` i emmagatzemar-la en una variable que s'anomena `element`. Per aquesta raó, a la primera iteració del bucle, `element` valdrà "un", a la segona, "dos", i a la tercera, "tres". Fàcil i senzill.

5.3. Exercicis d'autoavaluació

1. Escriviu un programa que verifiqui si la primera lletra de la cadena "macarrons" és una "m" o no. Si no comença per "m", el programa ha d'indicar la lletra inicial (per a comprovar-ne el funcionament canvieu el valor de la cadena perquè no comenci per "m").

2. Escriviu el mateix programa però que verifiqui si la primera lletra de la cadena "adéu" és una "a" o no. Si modifiqueu el programa anterior possiblement us trobareu amb un problema. Quin? Cerqueu com es pot solucionar.

3. Escriviu el mateix programa de l'exercici 2 utilitzant la construcció `A if C else B`.

4. Escriviu un programa que admeti entrades a través del teclat fins que la paraula introduïda comenci per "a". Si la paraula introduïda no comença per "a", l'ha d'escriure, i si comença per "a" ha d'escriure "adéu" i finalitzar el programa.

5. Escriviu un programa que prengui la cadena que conté la frase "hola bon dia" i la recorri caràcter a caràcter.

6. Escriviu un programa que prengui la cadena que conté la frase "hola bon dia", la converteixi en una llista que en contingui les paraules i recorri aquesta llista paraula a paraula. Pista: podeu utilitzar el mètode `split` de la classe `str`. Per a veure com funciona aquest mètode podeu utilitzar l'ajuda de l'intèrpret interactiu, entrant-hi i escrivint `help(str.split)`.

5.4. Solucions als exercicis de autoavaluació

1. Una possible solució és:

```
cadena="macarrons"
if cadena[0]=="m":
    print "La primera lletra es una  m\n"
else:
    print "La primera lletra no es una m, sino una "+ cadena[0]
```

2. El programa retorna l'error: `SyntaxError: Non-ASCII character '\XC3' in file p1.py on line 1, but no encoding declared; see http://www.python.org/peps/pep-0263.html`.

A l'enllaç que ofereix trobem la solució.

```
# -*- coding: utf-8 -*-

cadena=“adéu“
if cadena[0]==“a“:
    print “Comença per a“
else:
    print “No comença per a, comença per “+ cadena[0]
```

En aquesta solució hem suposat que la codificació de caràcters que utilitzem és Unicode UTF-8; en cas que utilitzeu una altra heu d'indicar la que correspongui.

3. Es pot escriure de la següent manera:

```
# -*- coding: utf-8 -*-
cadena=“adéu“
print “Comença per a“ if (cadena[0]==“a“) else “No comença per a, comença per “+cadena[0]
```

4. Una possible solució és la següent (recordeu que en indicar la codificació de caràcters heu de posar la que realment utilitzeu):

```
# -*- coding: utf-8 -*-

while True:
    entrada=raw_input(“ >“)
    if entrada[0]!=“a“:
        print entrada
    else:
        print “Adéu“
        break
```

5. Una possible solució és la que es mostra a continuació:

```
# -*- coding: utf-8 -*-

cadena=“hola bon dia“
for caracter in cadena:
        print caracter
```

6. Una possible solució és la següent:

```
# -*- coding: utf-8 -*-

cadena=“hola bon dia“
llista=cadena.split(“ “)
for paraula in llista:
    print paraula
```

6. Funcions

Una funció és un fragment de codi amb un nom associat i que realitza una sèrie de tasques i retorna un valor.

Els fragments de codi que tenen un nom associat i no retornen valors s'acostumen a anomenar *procediments*. En Python no existeixen els procediments, ja que quan el programador no especifica un valor de retorn, la funció torna el valor `None` ('res'), equivalent al `null` de Java. A més d'ajudar-nos a programar i depurar dividint el programa en parts, les funcions també permeten reutilitzar codi.

En Python les funcions es declaren de la forma següent:

Utilitat de les funcions

Les funcions seran útils per a organitzar els nostres programes, amb l'avantatge afegit que ens facilitaran la reutilització de codi.

```
def la_meva_funcio(param1, param2):
    print param1
    print param2
```

És a dir, la paraula clau `def` seguida del nom de la funció i, entre parèntesis, els arguments separats per comes. A continuació, sagnades i després dels dos punts, tindríem les línies de codi que conformen el codi a executar per la funció.

També podem trobar-nos amb una cadena de text com a primera línia del cos de la funció. Aquestes cadenes es coneixen amb el nom de *docstrings* ('cadenes de documentació') i serveixen, com el seu nom indica, per a documentar la funció.

```
def la_meva_funcio(param1, param2):
    """Aquesta funcio imprimeix els dos valors passats
    como a paràmetres"""
    print param1
    print param2
```

Això és el que imprimeix la funció `help` del llenguatge per a proporcionar una ajuda sobre l'ús i utilitat de les funcions. Tots els objectes poden tenir *docstrings*, no només les funcions, com veurem més endavant.

Tornant a la declaració de funcions, és important aclarir que en declarar la funció l'únic que fem és associar un nom al fragment de codi que conforma la funció, de manera que poguem executar aquest codi més tard cridant-lo pel seu nom. És a dir, a l'hora d'escriure aquestes línies no s'executa la funció. Per a cridar la funció (executar el seu codi) s'escriuria:

```
la_meva_funcio("hola", 2)
```

És a dir, el nom de la funció que volem cridar seguit dels valors que volem passar com a paràmetres entre parèntesis. L'associació dels paràmetres i els valors passats a la funció es fa normalment d'esquerra a dreta: com que a `param1` li hem donat el valor "hola" i `param2` val 2, `la_meva_funcio` imprimiria "hola" en una línia i, a continuació, "2".

No obstant això, també és possible modificar l'ordre dels paràmetres si indiquem el nom del paràmetre al qual associem un valor a l'hora de cridar la funció:

```
la_meva_funcio(param2 = 2, param1 = "hola")
```

El nombre de valors que es passen com a paràmetre al cridar la funció ha de coincidir amb el nombre de paràmetres que la funció accepta segons la seva declaració. En cas contrari, Python es queixarà:

```
>>> la_meva_funcio("hola")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: la_meva_funcio() takes exactly 2 arguments (1 given)
```

Tot i així, també és possible definir funcions amb un nombre variable d'arguments, o bé assignar valors per defecte als paràmetres per als casos en què no s'indiqui cap valor per a aquest paràmetre quan es crida la funció. Els valors per defecte per als paràmetres es defineixen situant un signe igual després del nom del paràmetre i, a continuació, el valor per defecte:

```
def imprimir(text, vegades = 1):
    print vegades * text
```

En l'exemple anterior, si no indiquem un valor per al segon paràmetre, s'imprimirà una sola vegada la cadena que li passem com a primer paràmetre:

```
>>> imprimir("hola")
hola
```

En canvi, si se li indica un altre valor, serà aquest el que s'utilitzi:

```
>>> imprimir("hola", 2)
holahola
```

Per a definir funcions amb un nombre variable d'arguments posem un últim argument per a la funció, el nom del qual ha d'anar precedit d'un signe *:

```
def diversos(param1, param2, *altres):
    for val in altres:
        print val

diversos(1, 2)
diversos(1, 2, 3)
diversos(1, 2, 3, 4)
```

Aquesta sintaxi funciona creant una tupla (que s'anomena `altres` en l'exemple) en la qual s'emmagatzemen els valors de tots els paràmetres addicionals passats com a argument. Per a la primera crida, `diversos(1, 2)`, la tupla `altres` estaria buida, atès que no s'han passat més paràmetres que els dos definits per defecte, per tant no s'imprimiria res. En la segona crida `altres` valdria (3,) i en la tercera (3, 4).

També es pot precedir el nom de l'últim paràmetre amb **; en aquest cas, en lloc d'una tupla es fa servir un diccionari. Les claus d'aquest diccionari serien els noms dels paràmetres indicats al cridar a la funció i els valors del diccionari, els valors associats a aquests paràmetres.

En el següent exemple s'utilitza la funció `items` dels diccionaris, que retorna una llista amb els seus elements, per a imprimir els paràmetres que conté el diccionari.

```
def diversos(param1, param2, **altres):
    for i in altres.items():
        print i

diversos(1, 2, tercer = 3)
```

Si coneixeu un altre llenguatge de programació segurament us preguntareu si en Python, quan passem una variable com a argument d'una funció, es passa per referència o per valor. El pas per referència passa com a argument una referència o punter a la variable, és a dir, passa la direcció de memòria on es troba el contingut de la variable, i no el seu contingut. En el pas per valor el que es passa com a argument és el valor que conté la variable.

La diferència entre ambdós rau en el fet que en el pas per valor els canvis que es facin sobre el paràmetre no es veuen fora de la funció, atès que els arguments de la funció són variables locals a la funció, que contenen els valors indicats per les variables que es van passar com a argument. És a dir, en realitat allò que es passa a la funció són còpies dels valors i no les variables en si.

Si volguéssim modificar el valor d'un dels arguments i que aquests canvis es reflectissin fora de la funció hauríem de passar el paràmetre per referència.

En Python també s'utilitza el pas per valor de referències a objectes, com en Java, tot i que en el cas de Python, a diferència de Java, tot és un objecte (per a ser exactes, el que passa en realitat és que a l'objecte se li assigna una altra etiqueta o nom a l'espai de noms local de la funció).

Tanmateix, no tots els canvis que fem als paràmetres dins d'una funció Python es reflectiran fora d'aquesta, ja que cal tenir en compte que en Python existeixen objectes immutables, com les tuples, de manera que si intentem modificar una tupla passada com a paràmetre, el que passaria en realitat és que es crearia una nova instància; per tant, els canvis no es veurien fora de la funció.

Vegem un petit programa per a demostrar-ho. En aquest exemple es fa ús del mètode `append` de les llistes. Un mètode no és més que una funció que pertany a un objecte, en aquest cas a una llista; en concret, `append` serveix per a afegir un element a una llista.

```
def f(x, y):
    x = x + 3
    y.append(23)
    print x, y

x = 22
y = [22]
f(x, y)
print x, y
```

El resultat de l'execució d'aquest programa seria

```
25 [22, 23]
22 [22, 23]
```

Com veiem, la variable *x* no conserva els canvis un cop sortim de la funció perquè els enters són immutables en Python. Tanmateix, la variable *i* sí els conserva, perquè les llistes són mutables.

En resum: els valors mutables es comporten com a pas per referència, i els immutables com a pas per valor.

Amb això finalitzem els aspectes relacionats amb els paràmetres de les funcions. Vegem finalment com retornar valors, procés per al qual s'utilitza la paraula clau `return`:

```
def sumar(x, y):
    return x + y

print sumar(3, 2)
```

Com veiem, aquesta funció tan senzilla no fa altra cosa que sumar els valors passats com a paràmetre i retornar el resultat com a valor de retorn.

També podríem passar diversos valors per a retornar a `return`.

```
def f(x, y):
 return x * 2, y * 2

a, b = f(1, 2)
```

Això no vol dir que les funcions Python puguin tornar diversos valors; el que passa en realitat és que Python crea una tupla amb els valors a retornar, i aquesta tupla és l'única variable que es retorna.

6.1. Exercicis d'autoavaluació

1. Escriviu una funció que retorni la primera lletra de la paraula que se li passi com a paràmetre.

2. Modifiqueu la funció anterior perquè retorni la lletra indicada per un segon paràmetre.

3. Escriviu una funció que, a partir d'una cadena que contingui una frase, retorni una llista que contingui les paraules de la frase (considerem que les paraules d'una frase se separen per espais en blanc).

6.2. Solució als exercicis d'autoavaluació

1. Es pot fer simplement:

```
def primeralletra(paraula):
    return paraula[0]

paraula="hola"
a=primeralletra(paraula)
print a
```

2. És una lleugera modificació de l'anterior:

```
def nlletra(paraula,n):
    return paraula[n]

paraula="hola"
a=nlletra(paraula,1)
print a
```

3. Aprofitem el mètode `split` de les cadenes:

```
 # -*- coding: utf-8 -*-

def segmenta(frase):
    paraules=frase.split(" ")
    return paraules

frase="Hola bon dia avui fa sol"
paraules=segmenta(frase)

for paraula in paraules:
    print paraula
```

7. 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.

La programació orientada a objectes (POO, o OOP segons les seves sigles en anglès) és un paradigma de programació en el qual els conceptes del món real rellevants per al nostre problema es modelen a través de classes i objectes, i en el qual el nostre programa consisteix en una sèrie d'interaccions entre aquests objectes.

7.1. Classes i objectes

Per a comprendre aquest paradigma primer hem de comprendre què és una classe i què és un objecte.

Un objecte és una entitat que agrupa un estat i una funcionalitat relacionades. L'estat de l'objecte es defineix a través de variables anomenades *atributs*, mentre que la funcionalitat es modela a través de funcions que reben el nom de *mètodes* de l'objecte.

Un exemple d'objecte podria ser un cotxe, en el qual tindríem atributs com la marca, el nombre de portes o el tipus de carburant i mètodes com arrencar i parar, o bé qualsevol altra combinació d'atributs i mètodes segons els aspectes que fossin rellevants per al nostre programa.

Una classe, d'altra banda, no és més que una plantilla genèrica que instancia els objectes; aquesta plantilla defineix quins atributs i mètodes tindran els objectes d'aquesta classe.

Tornant al nostre exemple: en el món real existeix un conjunt d'objectes que anomenem *cotxes* i que tenen un conjunt d'atributs comuns i un comportament comú; això és el que anomenem *classe*. Tanmateix, el meu cotxe no és igual que el cotxe del meu veí, i encara que pertanyen a la mateixa classe d'objectes, són objectes diferents.

En Python les classes es defineixen mitjançant la paraula clau `class` seguida del nom de la classe, dos punts (:) i, a continuació, sagnat, el cos de la classe. Com en el cas de les funcions, si la primera línia del cos és una cadena de text, aquesta serà la cadena de documentació de la classe o *docstring*.

```
class Cotxe:

    """Abstraccio dels objectes cotxe."""

    def __init__(self, benzina):
        self.benzina = benzina
        print "Tenim", benzina, "litres"

    def arrencar(self):
        if self.benzina > 0:
            print "Arrenca"
        else:
            print "No arrenca"

    def conduir(self):
        if self.benzina > 0:
            self.benzina -= 1
            print "Queden", self.benzina, "litres"
        else:
            print "No es mou"
```

El primer que crida l'atenció en l'exemple anterior és el nom tan curiós que té el mètode `__init__`. Aquest nom és una convenció i no un caprici. El mètode `__init__`, amb una doble barra baixa al principi i final del nom, s'executa immediatament després de crear un nou objecte a partir de la classe, procés que es coneix amb el nom d'*instanciació*. El mètode `__init__` serveix, com suggereix el seu nom, per a realitzar qualsevol procés d'inicialització que sigui necessari.

Com veiem, el primer paràmetre de `__init__` i de la resta de mètodes de la classe és sempre `self`. Aquesta és una idea inspirada en Modula-3 i serveix per a referir-se a l'objecte actual. Aquest mecanisme és necessari per a poder accedir als atributs i mètodes de l'objecte i diferenciar, per exemple, una variable local `la_meva_var` d'un atribut de l'objecte `self.la_meva_var`.

Si tornem al mètode `__init__` de la nostra classe `Cotxe` veurem com s'utilitza `self` per a assignar l'atribut `benzina` de l'objecte (`self.benzina`). El paràmetre `benzina` es destrueix al final de la funció, mentre que l'atribut `benzina` es conserva (i s'hi pot accedir) mentre l'objecte existeixi.

Per a crear un objecte s'escriu el nom de la classe seguit de qualsevol paràmetre que sigui necessari entre parèntesis. Aquests paràmetres són els que es passaran al mètode `__init__`, que, com dèiem, és el mètode que es crida quan s'instancia la classe.

```
el_meu_cotxe = Cotxe(3)
```

Us preguntareu llavors com és possible que a l'hora de crear el nostre primer objecte passem un sol paràmetre a `__init__`, el número 3, quan la definició de la funció indica clarament que necessita dos paràmetres (`self` i `benzina`). Això és així perquè Python passa el primer argument (la referència a l'objecte que es crea) *automàticament*.

Ara que ja hem creat el nostre objecte, podem accedir als seus atributs i mètodes mitjançant la sintaxi `objecte.atribut` i `objecte.mètode()`:

```
>>> print el_meu_cotxe.benzina
3
>>> el_meu_cotxe.arrencar()
Arrenca
>>> el_meu_cotxe.conduir()
Queden 2 litres
>>> el_meu_cotxe.conduir()
Queden 1 litres
>>> el_meu_cotxe.conduir()
Queden 0 litres
>>> el_meu_cotxe.conduir()
No es mou
>>> el_meu_cotxe.arrencar()
No arrenca
>>> print el_meu_cotxe.benzina
0
```

Com a últim apunt cal recordar que en Python, com ja s'ha comentat en repetides ocasions, tot són objectes. Les cadenes, per exemple, tenen mètodes com ara `upper()`, que retorna el text en majúscules, o `count(sub)`, que retorna el nombre de vegades que es troba la cadena `sub` en el text.

7.2. Herència

Hi ha tres conceptes que són bàsics per a qualsevol llenguatge de programació orientat a objectes: l'encapsulat, l'herència i el polimorfisme.

En un llenguatge orientat a objectes quan fem que una classe (subclasse) hereti d'una altra classe (superclasse) estem fent que la subclasse contingui tots els atributs i mètodes que tenia la superclasse. L'acte d'heretar d'una classe també s'anomena sovint *estendre una classe*.

Suposem que volem modelar els instruments musicals d'una banda. Tindrem llavors una classe `Guitarra`, una classe `Bateria`, una classe `Baix`, etc. Cadascuna d'aquestes classes tindrà una sèrie d'atributs i mètodes, però passa que, pel sol fet de ser instruments musicals, aquestes classes compartiran molts dels seus atributs i mètodes; un exemple seria el mètode `tocar()`.

És més senzill crear un tipus d'objecte `Instrument` amb els atributs i mètodes comuns i indicar al programa que `Guitarra`, `Bateria` i `Baix` són tipus d'instruments, fent que heretin d'`Instrument`.

Per a indicar que una classe hereta d'una altra es col·loca el nom de la classe de la qual s'hereta entre parèntesis després del nom de la classe:

```
class Instrument:
    def __init__(self, preu):
        self.preu = preu
    def tocar(self):
        print "Estem tocant musica"
    def trencar(self):
        print "Aixo ho pagues tu"
        print "Son", self.preu, "$$$"

class Bateria(Instrument):
    pass

class Guitarra(Instrument):
    pass
```

Com que `Bateria` i `Guitarra` hereten d'`Instrument`, ambdós tenen un mètode `tocar()` i un mètode `trencar()`, i s'inicialitzen passant un paràmetre `preu`. Però, què passaria si volguéssim especificar un nou paràmetre `tipus_corda` a l'hora de crear un objecte `Guitarra`? N'hi hauria prou amb escriure un nou mètode `__init__` per a la classe `Guitarra` que s'executaria

en lloc del `__init__` d'`Instrument`. Això és el que es coneix com a *sobreescriure mètodes*.

Ara bé, pot passar que en alguns casos necessitem sobreescriure un mètode de la classe pare, però que en aquest mètode volguem executar el mètode de la classe pare perquè el nostre mètode nou només necessita un parell d'instruccions noves. En aquest cas faríem servir la sintaxi `SuperClasse.metode(self, args)` per a cridar el mètode d'igual nom de la classe pare. Per exemple, per a cridar el mètode `__init__` d'`Instrument` des de `Guitarra` faríem servir `Instrument.__init__(self, preu)`.

Observeu que en aquest cas sí que cal especificar el paràmetre `self`.

7.3. Herència múltiple

En Python, a diferència d'altres llenguatges com Java o C#, es permet l'herència múltiple, és a dir, una classe pot heretar de diverses classes a la vegada. Així, podríem tenir una classe `Cocodril` heretada de la classe `Terrestre`, amb mètodes com `caminar()` i atributs com `velocitat_caminar`, i de la classe `Aquatic`, amb mètodes com `nedar()` i atributs com `velocitat_nedar`. N'hi ha prou amb enumerar les classes de què s'hereta separades per comes:

```
class Cocodril(Terrestre, Aquatic):
    pass
```

En el cas que alguna de les classes pare tingués mètodes amb el mateix nom i número de paràmetres, les classes sobreescriurien la implementació dels mètodes de les classes situats més a la seva dreta en la definició.

En el següent exemple, com que `Terrestre` es troba més a l'esquerra, seria la definició de `desplacar` d'aquesta classe la que quedaria i, per tant, si cridem el mètode `desplacar` d'un objecte de tipus `Cocodril` el que s'imprimiria seria "L'animal camina".

```
class Terrestre:
    def desplacar(self):
    print "L'animal camina"

class Aquatic:
    def desplacar(self):
    print "L'animal neda"

class Cocodril(Terrestre, Aquatic):
    pass

c = Cocodril()
c.desplacar()
```

7.4. Polimorfisme

La paraula *polimorfisme*, del grec *poly morphos* ('diverses formes'), es refereix a l'habilitat d'objectes de diferents classes de respondre al mateix missatge. Això es pot aconseguir a través de l'herència: un objecte d'una classe derivada és, alhora, un objecte de la classe pare, de manera que allà on es requereix un objecte de la classe pare també es pot utilitzar un de la classe filla.

Python, en ser de tipatge dinàmic, no imposa restriccions als tipus que es poden passar a una funció, més enllà del fet que l'objecte es comporti com s'espera: si s'ha de cridar un mètode `f()` de l' objecte passat com a paràmetre, per exemple, és clar que l'objecte haurà de comptar amb aquest mètode. Per aquest motiu, a diferència de llenguatges de tipatge estàtic com Java o C++, el polimorfisme en Python no és de gran importància.

En ocasions també es fa servir el terme *polimorfisme* per a referir-se a la sobrecàrrega de mètodes, terme que es defineix com la capacitat del llenguatge per a determinar quin mètode cal executar d'entre diversos mètodes amb el mateix nom, segons el tipus o nombre de paràmetres que es passen. En Python no existeix la sobrecàrrega de mètodes (l'últim mètode sobreescriuria la implementació dels anteriors), tot i que es pot aconseguir un comportament similar recorrent a funcions amb valors per defecte per als paràmetres o a la sintaxi `*params` o `**params`, explicada a l'apartat sobre les funcions en Python, o bé fent servir decoradors.

7.5. Encapsulació

L'encapsulació es refereix a impedir l'accés a determinats mètodes i atributs dels objectes establint així què és el que es pot utilitzar des de l'exterior de la classe.

Això s'aconsegueix en altres llenguatges de programació, com Java, utilitzant modificadors d'accés que defineixen si qualsevol pot accedir a aquesta funció o variable (`public`) o si l'accés està restringit a la pròpia classe (`private`).

En Python no existeixen els modificadors d'accés, i el que se sol fer és que l'accés a una variable o funció és determinat pel seu nom: si el nom comença amb dos guions baixos (i no acaba també amb dos guions baixos) es tracta d'una variable o funció privada; en cas contrari, és pública. Els mètodes el nom dels quals comença i acaba amb dos guions baixos són mètodes especials que Python crida automàticament en certes circumstàncies.

En el següent exemple només s'imprimirà la cadena corresponent al mètode `public()`, mentre que en intentar cridar el mètode `__privat()` Python llençarà una excepció per a queixar-se de que no existeix (evidentment existeix, però no el podem veure perquè és privat).

```
class Exemple:
    def public(self):
        print “Public“
    def __privat(self):
        print “Privat“

ex = Exemple()
ex.public()
ex.__privat()
```

Aquest mecanisme es basa en que els noms que comencen amb un doble guió baix canvien el seu nom per a incloure el nom de la classe (característica que es coneix amb el nom de *name mangling*). Això implica que el mètode o atribut no és realment privat, i podem accedir-hi mitjançant una petita trampa:

```
ex._Exemple__privat()
```

A vegades també pot passar que volguem permetre l'accés a algun atribut del nostre objecte, però d'una forma controlada. Per a això podem escriure mètodes que tenen aquest únic propòsit, mètodes que normalment, per convenció, s'anomenen `getVariable` i `setVariable`; per això es coneixen també amb el nom de *getters* i *setters*.

```
class Data():
    def __init__(self):
        self.__dia = 1
    def getDia(self):
        return self.__dia
    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print “Error”

la_meva_data = Data()
la_meva_data.setDia(33)
```

Això es podria simplificar mitjançant propietats, que abstreuen l'usuari del fet que s'estan utilitzant mètodes d'amagat per obtenir i modificar els valors de l'atribut:

```
class Data(object):
    def __init__(self):
        self.__dia = 1
```

```
    def getDia(self):
        return self.__dia
    def setDia(self, dia):
        if dia > 0 and dia < 31:
            self.__dia = dia
        else:
            print “Error“
     dia = property(getDia, setDia)

la_meva_data = Data()
la_meva_data.dia = 33
```

7.6. Classes de “nou estil”

A l'exemple anterior us haurà cridat l'atenció el fet que la classe `Data` derivi d'`object`. El motiu és que per poder fer servir propietats, la classe ha de ser de “nou estil”, classes enriquides que es van introduir a Python 2.2 i que seran l'estàndard a Python 3.0, però que encara conviuen amb les classes “clàssiques” per raons de retrocompatibilitat. A més a més de les propietats, les classes de nou estil afegeixen altres funcionalitats, com descriptors o mètodes estàtics.

Perquè una classe sigui de nou estil és necessari, per ara, que estengui una classe de nou estil. En el cas que no calgui heretar el comportament o l'estat de cap mena, com en el nostre exemple anterior, es pot heretar d'`object`, que és un objecte buit que serveix com a base per a totes les classes de nou estil.

La diferència principal entre les classes antigues i les de nou estil consisteix en què a l'hora de crear una nova classe, abans no es definia realment un nou tipus, sinó que tots els objectes creats a partir de classes, fossin aquestes les classes que fossin, eren de tipus `instance`.

7.7. Mètodes especials

Ja vam veure al principi de l'apartat l'ús del mètode `__init__`. Existeixen altres mètodes amb significats especials, els noms dels quals sempre comencen i acaben amb dos guions baixos. A continuació podeu veure'n alguns especialment útils.

```
__init__(self, args)
```

Mètode per a realitzar tasques d'inicialització que es crida després de crear l'objecte.

```
__new__(cls, args)
```

Mètode exclusiu de les classes de nou estil que s'executa abans de `__init__` i que s'encarrega de construir i retornar l'objecte. És equivalent als constructors de C++ o Java. Es tracta d'un mètode estàtic, és a dir, que existeix amb independència de les instàncies de la classe. És un mètode de classe, no d'objecte i, per tant, el primer paràmetre no és `self`, sinó la pròpia classe: `cls`.

`__del__(self)`

Mètode que es crida quan s'esborra l'objecte. També anomenat *destructor*, s'utilitza per a realitzar tasques de neteja.

`__str__(self)`

Mètode que es crida per a crear una cadena de text que representi el nostre objecte. S'utilitza quan fem servir `print` per a mostrar el nostre objecte o quan fem servir la funció `str(obj)` per a crear una cadena a partir del nostre objecte.

`__cmp__(self, altre)`

Mètode que es crida quan es fan servir els operadors de comparació per a comprovar si el nostre objecte és més petit, més gran o igual que l'objecte que es passa com a paràmetre. Ha de tornar un número negatiu si l'objecte és menor, zero si són iguals i un número positiu si l'objecte és més gran. Si no es defineix aquest mètode i s'intenta comparar l'objecte mitjançant els operadors <, <=, > o >=, es llençarà una excepció. Si es fan servir els operadors == o != per a comprovar si dos objectes són iguals, es comprova si són el mateix objecte (si tenen el mateix `id`).

`__len__(self)`

Mètode que es crida per a comprovar la longitud de l'objecte. S'utilitza, per exemple, quan es crida la funció `len(obj)` sobre el nostre objecte. Com és de suposar, el mètode ha de retornar la longitud de l'objecte.

Hi ha molts altres mètodes especials que permeten, entre altres coses, utilitzar el mecanisme de *slicing* sobre el nostre objecte, utilitzar els operadors aritmètics o usar la sintaxi de diccionaris, però un estudi exhaustiu de tots els mètodes queda fora del propòsit de l'apartat.

7.8. Exercicis d'autoavaluació

1. Creeu una classe que serveixi per a emmagatzemar informació sobre un *token*. Aquesta informació serà la forma, el lema i l'etiqueta morfosintàctica. Escriviu els mètodes `t_forma`, `t_lema` i `t_etiqueta` que retornin la forma, el lema i l'etiqueta, respectivament.

2. Creeu un nou mètode `t_categoria` que retorni "substantiu" si l'etiqueta comença per "N", "verb" si comença per "V", "adjectiu" si comença per "A" i "altra" si comença per qualsevol altra lletra.

3. Creeu un mètode que retorni la pseudoarrel, és a dir la part comuna entre forma i lema. Per exemple de "nens-nen" és "nen", de "canto-cantar" és "cant" i de "cantava-cantar" és "canta". No és una arrel autèntica, simplement la part comuna.

4. Creeu un mètode que retorni la pseudoterminació, és a dir, de "nens-nen" retorni "s", de "cant-cantar" retorni "ar" i de "cantava-cantar" retorni "va". No és una terminació real. Podeu aprofitar en part el mètode de l'exercici anterior.

7.9. Solució als exercicis d'autoavaluació

1. Una possible solució és la següent:

```
# -*- coding: utf-8 -*-

class Token:
    def __init__(self, forma, lema, etiqueta):
        self.forma=forma
        self.lema=lema
        self.etiqueta=etiqueta

    def t_forma(self):
        return self.forma

    def t_lema(self):
        return self.lema

    def t_etiqueta(self):
        return self.etiqueta


token1=Token("nens","nen","NCMS")
print token1.t_forma()
```

2. Indiquem només el codi del nou mètode:

```
def t_categoria(self):
    etiqueta=self.t_etiqueta()
    if (etiqueta[0] == "V"):
```

```
        return “Verb“
    elif (etiqueta[0] == “N“):
        return “Substantiu“
    elif (etiqueta[0] == “A“):
        return “Adjectiu“
    else:
        return “Altra“
```

3. Indiquem només el codi del nou mètode:

```
def t_pseudoarrel(self):
    cont=0;
    lema=self.t_lema()
    forma=self.t_forma()
    for i in lema:
        if (forma[cont]==i):
            cont=cont+1
        else:
            break
    return forma[0:cont]
```

4. Indiquem només el codi del nou mètode:

```
def t_pseudoterminacio(self):
    cont=0;
    lema=self.t_lema()
    forma=self.t_forma()
    for i in lema:
        if (forma[cont]==i):
            cont=cont+1
        else:
             break
    return forma[cont:]
```

8. Revisitant objectes

En els apartats dedicats als tipus simples i les col·leccions vèiem per primera vegada alguns dels objectes del llenguatge Python: nombres, booleans, cadenes de text, diccionaris, llistes i tuples.

Ara que sabem què són les classes, els objectes, les funcions i els mètodes és el moment de revisitar aquests objectes per a descobrir-ne el veritable potencial.

Veurem a continuació alguns mètodes útils d'aquests objectes. Evidentment, no cal memoritzar-los, però sí, almenys, recordar que existeixen per a quan siguin necessaris.

8.1. Diccionaris

`D.get(k[,d])`

Busca el valor de la clau `k` al diccionari. És equivalent a utilitzar `D[k]` però en utilitzar aquest mètode podem indicar un valor a retornar per defecte, `d`, si no es troba la clau, mentre que amb la sintaxi `D[k]`, si no existeix la clau es llença una excepció.

help

Recordeu que des de l'intèrpret interactiu podeu obtenir informació sobre els mètodes disponibles per a les diferents classes. Així `help(dict)` mostrarà la informació sobre els mètodes dels diccionaris, i igual passarà amb `str`, `tuple`, `list`, `int`, etc.

`D.has_key(k)`

Comprova si el diccionari té la clau `k`. És equivalent a la sintaxi `k in D`.

`D.items()`

Retorna una llista de tuples amb parells clau-valor.

`D.keys()`

Retorna una llista de les claus del diccionari.

`D.pop(k[,d])`

Esborra la clau `k` del diccionari i retorna el seu valor. Si no es troba aquesta clau es torna `d` si es va especificar el paràmetre o bé es llença una excepció.

`D.values()`

Retorna una llista dels valors del diccionari.

8.2. Cadenes

`S.count(sub[,start[,end]])`

Retorna el nombre de vegades que es troba `sub` a la cadena. Els paràmetres opcionals `start` i `end` defineixen la subcadena amb què es farà la cerca.

`S.find(sub[,start[,end]])`

Retorna la posició en què es va trobar per primera vegada `sub` en la cadena o -1 si no s'ha trobat.

`S.join(sequence)`

Mètode que retorna una cadena resultant de concatenar les cadenes de la seqüència `sequence` separades per la cadena sobre la qual es crida el mètode.

`S.partition(set)`

Cerca el separador `set` a la cadena i retorna una tupla amb la subcadena fins a aquest separador, el separador en si i la subcadena del separador fins al final de la cadena. Si no es troba el separador, la tupla contindrà la pròpia subcadena i dues cadenes buides.

`S.replace(old,new[,count])`

Retorna una cadena en què s'han substituït totes les ocurrències de la cadena `old` per la cadena `new`. Si s'especifica el paràmetre `count`, indica el nombre màxim d'ocurrències a reemplaçar.

`S.split([set[,maxsplit]])`

Retorna una llista que conté les subcadenes en què es divideix la nostra cadena en dividir-la pel delimitador `set`. En el cas que no s'especifiqui `set`, s'usen espais. Si s'especifica `maxsplit`, indica el nombre màxim de particions que es realitzaran.

8.3. Llistes

`L.append(object)`

Afegeix un objecte al final de la llista.

`L.count(value)`

Retorna el nombre de vegades que es troba `value` a la llista.

`L.extend(iterable)`

Afegeix els elements d'`iterable` a la llista.

`L.index(value[,start[,stop]])`

Retorna la posició en què es va trobar la primera ocurrència de `value`. Si s'especifiquen `start` i `stop` defineixen les posicions d'inici i final d'una subllista on es fa la cerca.

`L.insert(index,object)`

Insereix l'objecte `object` a la posició `index`.

`L.pop([index])`

Torna el valor de la posició `index` i l'elimina de la llista. Si no s'especifica la posició, s'utilitza l'últim element de la llista.

`L.remove(value)`

Elimina la primera ocurrència de `value` a la llista.

`L.reverse ()`

Inverteix la llista. Aquesta funció treballa sobre la pròpia llista des de la qual s'invoca el mètode, no sobre una còpia.

`L.sort(cmp=None,key=None,reverse=False)`

Ordena la llista. Si s'especifica `cmp`, aquest ha de ser una funció que prengui com a paràmetre dos valors `x` i `y` de la llista i retorni -1 si `x` és menor que `y`, 0 si són iguals i 1 si `x` és major que `y`. El paràmetre `reverse` és un booleà que indica si s'ha d'ordenar la llista de manera inversa, que seria equivalent a cridar primer `L.sort()` i després `L.reverse()`. Finalment, si s'especifica, el paràmetre `key` ha de ser una funció que prengui un element de la llista i retorni una clau per a utilitzar a l'hora de comparar, en lloc de l'element en si.

8.4. Exercicis d'autoavaluació

1. Escriviu un programa que divideixi una frase en espais en blanc i retorni una cadena amb les paraules ordenades alfabèticament.

2. Modifiqueu el programa anterior perquè la llista estigui ordenada a la inversa.

3. Escriviu un programa que, a partir d'una cadena que contingui una frase, retorni les paraules ordenades alfabèticament i indiqui quantes vegades apareix cada paraula a la frase.

8.5. Solució als exercicis d'autoavaluació

1. Una possible solució és la següent:

```
# -*- coding: utf-8 -*-

frase="avui fa un dia molt maco però demà plourà"
llista=frase.split()
llista.sort()
print llista
```

2. Podem utilitzar l'opció `reverse=True` de la funció `sort`:

```
# -*- coding: utf-8 -*-

frase="avui fa un dia molt maco però demà plourà"
llista=frase.split()
llista.sort(reverse=True)
print llista
```

3. Una possible solució és la següent:

```
# -*- coding: utf-8 -*-

frase="aquesta frase té aquesta paraula repetida, i aquesta
paraula també està repetida i té una altra paraula més"
llista=frase.split()
llista.sort()
for paraula in llista:
    print paraula , llista.count(paraula)
```

Fixeu-vos que aquesta solució escriu les paraules repetides diverses vegades. Per fer que cada paraula només s'escrigui una vegada podem fer el següent:

```
# -*- coding: utf-8 -*-

frase="aquesta frase té aquesta paraula repetida, i aquesta
paraula també està repetida i té una altra paraula més"
```

```
llista=frase.split()
diccionari={}
for paraula in llista:
    diccionari[paraula]=lista.count(paraula)
claus=diccionari.keys()
claus.sort()
for paraula in claus:
    print paraula,diccionari[paraula]
```

9. Excepcions

Les excepcions són errors detectats per Python durant l'execució d'un programa.

Quan l'intèrpret es troba amb una situació excepcional, com intentar dividir un número entre 0 o intentar accedir a un arxiu que no existeix, genera o llença una excepció i informa a l'usuari que hi ha algun problema.

Si l'excepció no es captura, el flux d'execució s'interromp i es mostra a la consola la informació associada a l'excepció, de manera que el programador pugui solucionar el problema.

Vegem un petit programa que llençaria una excepció en intentar dividir 1 entre 0.

```
def divisio(a, b):
   return a/b

def calcular():
    divisio(1, 0)

calcular()
```

Si l'executem obtindrem el missatge d'error:

```
$ python exemple.py

Traceback (most recent call last):

File "exemple.py", line 7, in

calcular()

File "exemple.py", line 5, in calcular

divisio(1, 0)
```

```
File “exemple.py“, line 2, in divisio

a/b

ZeroDivisionError: integer division or modulo by zero
```

El primer que es mostra és el traçat de pila o *traceback*, que consisteix en una llista amb les crides que ha provocat l'excepció. Com veiem en el traçat de pila, l'error ha estat provocat per la crida a `calcular()` de la línia 7 que, al seu torn, crida a `divisio(1, 0)` a la línia 5 i, en última instància, per l'execució de la sentència `a/b` de la línia 2 de `divisio`.

A continuació veiem el tipus de l'excepció, `ZeroDivisionError`, juntament amb una descripció de l'error: `integer division or modulo by zero` (és a dir, 'mòdul o divisió entera entre zero').

En Python es fa servir la construcció `try-except` per a capturar i tractar les excepcions. El bloc `try` ('intentar') defineix el fragment de codi en què creiem que es podria produir una excepció. El bloc `except` ('excepció') permet indicar el tractament que es durà a terme si es produeix aquesta excepció. Moltes vegades el tractament de l'excepció consistirà simplement en imprimir un missatge més amigable per a l'usuari, altres vegades ens interessarà registrar l'error i també podrem establir una estratègia de resolució del problema.

En el següent exemple intentem crear un objecte `f` de tipus fitxer. Si no existeix l'arxiu passat com a paràmetre, es llença una excepció de tipus `IOError`, que capturem gràcies al nostre `try-except`.

```
try:
    f = file(“arxiu.txt“)
except:
    print “L'arxiu no existeix”
```

Python permet utilitzar diversos `except` per a un sol bloc `try`, de manera que poguem donar un tractament diferent a l'excepció depenent del tipus d'excepció de què es tracti. Això és una bona pràctica, i és tan senzill com indicar el nom del tipus d'excepció a continuació de l'`except`.

```
try:
    num = int(“3a”)
except NameError:
    print “La variable no existeix“
except ValueError:
    print “El valor no es un numero“
```

Quan es llença una excepció al bloc `try`, es busca en cadascuna de les clàusules `except` un controlador adequat per al tipus d'error que s'ha produït. En cas que no es trobi, es propaga l'excepció.

A més, podem fer que un mateix `except` serveixi per a tractar més d'una excepció utilitzant una tupla per a llistar els tipus d'error que volem que tracti el bloc:

```
try:
    num = int("3a")
except (NameError, ValueError):
    print "S'ha produït un error"
```

La construcció `try-except` pot comptar, a més, amb una clàusula `else`, que defineix un fragment de codi a executar només si no s'ha produït cap excepció al `try`.

```
try:
    num = 33
except:
    print "S'ha produit un error!"
else:
    print "Tot ha funcionat correctament"
```

També hi ha una clàusula `finally` que s'executa sempre, es produeixi o no una excepció. Aquesta clàusula se sol utilitzar, entre altres coses, per a tasques de neteja.

```
try:
    z = x/y
except ZeroDivisionError:
    print "Divisio entre zero"
finally:
    print "Netejant"
```

També és interessant comentar que com a programadors podem crear i llençar les nostres pròpies excepcions. Només és necessari crear una classe que hereti d'*`Exception`* o de qualsevol de les seves filles i llençar-les amb `raise`.

```
class El_Meu_Error(Exception):
    def __init__(self, valor):
        self.valor = valor
    def __str__(self):
        return "Error " + str(self.valor)

try:
    if resultat > 20:
        raise El_Meu_Error(33)
except El_Meu_Error, e:
    print e
```

Per últim, a continuació es llisten, com a referència, les excepcions disponibles per defecte, així com la classe de la qual deriva cada una d'elles entre parèntesis.

- **BaseException**: Classe de la qual hereten totes les excepcions.
- **Exception(BaseException)**: Superclasse de totes les excepcions que no siguin de sortida.
- **GeneratorExit(Exception)**: Es demana que se surti d'un generador.
- **StandardError(Exception)**: Classe base per a totes les excepcions que no tinguin a veure amb sortir de l'intèrpret.
- **ArithmeticError(StandardError)**: Classe base per als errors aritmètics.
- **FloatingPointError(ArithmeticError)**: Error en una operació de coma flotant.
- **OverflowError(ArithmeticError)**: Resultat massa gran per a poder-lo representar.
- **ZeroDivisionError(ArithmeticError)**: Es llença quan el segon argument d'una operació de divisió o mòdul és 0.
- **AssertionError(StandardError)**: Ha fallat la condició d'un estament `assert`.
- **AttributeError(StandardError)**: No s'ha trobat l'atribut.
- **EOFError(StandardError)**: S'ha intentat llegir més enllà del final de fitxer.
- **EnvironmentError(StandardError)**: Classe pare dels errors relacionats amb l'entrada/sortida.
- **IOError(EnvironmentError)**: Error en una operació d'entrada/sortida.
- **OSError(EnvironmentError)**: Error en una crida a sistema.
- **WindowsError(OSError)**: Error en una crida a sistema en Windows.
- **ImportError(StandardError)**: No s'ha trobat el mòdul o l'element del mòdul que es volia importar.
- **LookupError(StandardError)**: Classe pare dels errors d'accés.
- **IndexError(LookupError)**: L'índex de la seqüència és fora del rang possible.
- **KeyError(LookupError)**: La clau no existeix.
- **MemoryError(StandardError)**: No queda prou memòria.
- **NameError(StandardError)**: No s'ha trobat cap element amb aquest nom.
- **UnboundLocalError(NameError)**: El nom no està associat a cap variable.
- **ReferenceError(StandardError)**: L'objecte no té cap referència forta apuntant cap a ell.
- **RuntimeError(StandardError)**: Error en temps d'execució no especificat.
- **NotImplementedError(RuntimeError)**: Aquest mètode o funció no està implementat.
- **SyntaxError(StandardError)**: Classe pare per als errors sintàctics.
- **IndentationError(SyntaxError)**: Error en el sagnat de l'arxiu.
- **TabError(IndentationError)**: Error degut a la barreja d'espais i tabuladors.
- **SystemError(StandardError)**: Error intern de l'intèrpret.
- **TypeError(StandardError)**: Tipus d'argument no apropiat.
- **ValueError(StandardError)**: Valor de l'argument no apropiat.
- **UnicodeError(ValueError)**: Classe pare per als errors relacionats amb Unicode.

- **UnicodeDecodeError(UnicodeError)**: Error de decodificació d'Unicode.
- **UnicodeEncodeError(UnicodeError)**: Error de codificació d'Unicode.
- **UnicodeTranslateError (UnicodeError)**: Error de traducció d'Unicode.
- **StopIteration(Exception)**: S'utilitza per indicar el final de l'iterador.
- **Warning(Exception)**: Classe pare per als avisos.
- **DeprecationWarning(Warning)**: Classe pare per a avisos sobre característiques obsoletes.
- **FutureWarning(Warning)**: Avís. La semàntica de la construcció canviarà en un futur.
- **ImportWarning(Warning)**: Avís sobre possibles errors a l'hora d'importar.
- **PendingDeprecationWarning(Warning)**: Avís sobre característiques que es marquen com a obsoletes en un futur proper.
- **RuntimeWarning(Warning)**: Avís sobre comportaments dubtosos en temps d'execució.
- **SyntaxWarning(Warning)**: Avís sobre sintaxi dubtosa.
- **UnicodeWarning(Warning)**: Avís sobre problemes relacionats amb Unicode, sobretot amb problemes de conversió.
- **UserWarning(Warning)**: Classe pare per a avisos creats pel programador.
- **KeyboardInterrupt(BaseException)**: L'usuari ha interromput el programa.
- **SystemExit(BaseException)**: Petició de l'intèrpret per a acabar l'execució.

10. Mòduls i paquets

10.1. Mòduls

Per a facilitar el manteniment i la lectura dels programes massa llargs, podem dividir-los en mòduls, agrupant elements relacionats. Els mòduls són entitats que permeten una organització i divisió lògica del nostre codi. Els fitxers són la seva contrapartida física: cada arxiu Python emmagatzemat en disc equival a un mòdul.

Escrivim el nostre primer mòdul creant un petit fitxer modul.py amb el següent contingut:

```
def la_meva_funcio():
    print “una funcio“

class La_Meva_Classe:
    def __init__(self):
        print “una classe“

print “un modul“
```

Si volguéssim fer servir la funcionalitat definida en aquest mòdul al nostre programa, hauríem d'importar-lo. Per a importar un mòdul es fa servir la paraula clau `import` seguida del nom del mòdul, que és el nom de l'arxiu sense l'extensió. Com a exemple podem crear un arxiu programa.py en el mateix directori on guardem l'arxiu del mòdul (això és important, perquè si no es troba en el mateix directori, Pyhton no podrà trobar-lo).

```
import modul
modul.la_meva_funcio()
```

L'`import` no només fa que tinguem disponible tot el que està definit dins del mòdul, sinó que també executa el codi del mòdul. Per aquesta raó el nostre programa, a més d'imprimir el text “una funcio” al cridar `la _meva_funcio`, també imprimirà el text “un mòdul”, a causa del `print` del mòdul importat. No s'imprimiria, però, el text “una classe”, ja que en el mòdul només es defineix la classe, però no s'instancia.

La clàusula `import` també permet importar diversos mòduls en la mateixa línia. En el següent exemple podem veure com s'importa amb una sola clàusula `import` els mòduls de la distribució per defecte de Python: `os`, que engloba funcionalitats relatives al sistema operatiu; `sys`, amb funcionalitats relacionades amb el propi intèrpret de Python, i `time`, en què s'emmagatzemen funcions per a manipular dates i hores.

```
import os, sys, time

print time.asctime()
```

Sens dubte us haureu fixat en un detall important d'aquest exemple i de l'anterior, i és que, com veiem, cal precedir el nom dels objectes que importem d'un mòdul amb el nom del mòdul al qual pertanyen, o el que és el mateix, l'espai de noms en què es troben. Això permet que no sobreescrivim accidentalment algun altre objecte que tingui el mateix nom en importar un altre mòdul.

Tanmateix, és possible utilitzar la construcció `from-import` per a estalviarnos haver d'indicar el nom del mòdul abans de l'objecte que ens interessa. D'aquesta manera, s'importa l'objecte o els objectes que indiquem a l'espai de noms actual.

```
from time import asctime

print asctime()
```

Encara que es considera una mala pràctica, també és possible importar tots els noms del mòdul a l'espai de noms actual usant el caràcter *****:

```
from time import *
```

Ara bé, recordareu que a l'hora de crear el nostre primer mòdul es va insistir en desar-lo al mateix directori que el programa que l'importava. Llavors, com podem importar els mòduls `os`, `sys` o `time` si no es troben els arxius os.py, sys.py i time.py en el mateix directori?

A l'hora d'importar un mòdul, Python recorre tots els directoris indicats en la variable d'entorn `PYTHONPATH` a la recerca d'un fitxer amb el nom adequat. El valor de la variable `PYTHONPATH` es pot consultar des de Python mitjançant `sys.path`.

```
>>> import sys

>>> sys.path
```

D'aquesta manera, perquè el nostre mòdul estigui disponible per a tots els programes del sistema n'hi hauria prou amb copiar-lo a un dels directoris indicats a PYTHONPATH.

En el cas que Python no trobi cap mòdul amb el nom especificat, es llençaria una excepció de tipus ImportError.

Per últim, és interessant comentar que en Python els mòduls també són objectes, concretament del tipus module. Això significa que poden tenir atributs i mètodes. Un dels seus atributs, __name__, es fa servir sovint per a incloure codi executable en un mòdul però que aquest codi només s'executi si es crida el mòdul com a programa, i no quan l'importem. Per a aconseguir-ho només cal saber que quan s'executa el mòdul directament, __name__ té com a valor __main__, mentre que quan s'importa, el nom de __name_ és el nom del mòdul.

```
print "Es mostra sempre"

if __name__ == "__main__":
    print "Es mostra si no es importacio"
```

Un altre atribut interessant és __doc__, que, com en el cas de funcions i classes, serveix a manera de documentació de l'objecte (*docstring* o cadena de documentació). El seu valor és el de la primera línia del cos del mòdul, en el cas que aquesta sigui una cadena de text; en cas contrari, valdrà None.

10.2. Paquets

Els mòduls serveixen per a organitzar el codi i els paquets serveixen per organitzar els mòduls. Els paquets són tipus especials de mòduls (tots dos són de tipus module) que permeten agrupar mòduls relacionats. Mentre que els mòduls es corresponen a nivell físic amb els arxius, els paquets es representen mitjançant directoris.

Mòduls i paquets

Els paquets són tipus especials de mòduls que permeten agrupar mòduls relacionats.

En una aplicació qualsevol podríem tenir, per exemple, un paquet iu per a la interfície o un paquet bbdd per a la persistència a la base de dades.

Per fer que Python tracti un directori com a paquet és necessari crear un arxiu __init__.py en aquesta carpeta. En aquest fitxer es poden definir elements que pertanyin a aquest paquet, com una constant DRIVER per al paquet bbdd, tot i que habitualment es tractarà d'un arxiu buit. Per a fer que un cert mòdul es trobi dins d'un paquet, n'hi ha prou de copiar l'arxiu que defineix el mòdul al directori del paquet.

Igual que passava amb els mòduls, per a importar paquets també s'utilitza `import` i `from-import` i el caràcter `.` per a separar paquets, subpaquets i mòduls.

```
import paq.subpaq.modul

paq.subpaq.modul.func()
```

PyPI

Per a trobar algun mòdul o paquet que cobreixi una certa necessitat, podeu consultar la llista de PyPI (*Python Package Index*) a http://pypi.python.org/, que compta, a l'hora d'escriure aquestes línies, amb més de 4.000 paquets diferents.

10.3. Exercicis d'autoavaluació

1. Creeu un mòdul a partir dels exercicis de l'apartat 7. Importeu el mòdul en un programa i proveu si funciona.

2. A partir del següent mòdul*, creeu un programa que demani la llengua de partida, la d'arribada i la frase a traduir i retorni la frase traduïda. Les llengües s'han de donar amb els codis (en, es, ca, fr, etc.). Per a poder realitzar aquest exercici haureu de llegir el primer subapartat de l'apartat següent.

***El mòdul es pot descarregar de http://www.technobabble.dk/2008/may/25/translate-strings-using-google-translate/.**

El mòdul a utilitzar és:

```
"""
translate.py

Translates strings using Google Translate

All input and output is in unicode.
"""

__all__ = ('source_languages', 'target_languages', 'translate')

import sys
import urllib2
import urllib

from BeautifulSoup import BeautifulSoup

opener = urllib2.build_opener()
opener.addheaders = [('User-agent', 'translate.py/0.1')]

# lookup supported languages

translate_page = opener.open("http://translate.google.com/translate_t")
translate_soup = BeautifulSoup(translate_page)

source_languages = {}
target_languages = {}
```

```
for language in translate_soup("select", id="old_sl")[0].childGenerator():
    if language['value'] != 'auto':
        source_languages[language['value']] = language.string

for language in translate_soup("select", id="old_tl")[0].childGenerator():
    if language['value'] != 'auto':
        target_languages[language['value']] = language.string

def translate(sl, tl, text):

    """ Translates a given text from source language (sl) to
        target language (tl) """

    assert sl in source_languages, "Unknown source language."
    assert tl in target_languages, "Unknown taret language."

    assert type(text) == type(u''), "Expects input to be unicode."

    # Do a POST to google

    # I suspect "ie" to be Input Encoding.
    # I have no idea what "hl" is.

    translated_page = opener.open(
        "http://translate.google.com/translate_t?" +
        urllib.urlencode({'sl': sl, 'tl': tl}),
        data=urllib.urlencode({'hl': 'en',
                               'ie': 'UTF8',
                               'text': text.encode('utf-8'),
                               'sl': sl, 'tl': tl})
    )

    translated_soup = BeautifulSoup(translated_page)

    return translated_soup('div', id='result_box')[0].string
```

Aquest mòdul s'ha de desar com a translate.py.

3. Modifiqueu el mòdul translate.py perquè pogueu demanar-li les llengües de partida i d'arribada disponibles.

10.4. Solució als exercicis d'autoavaluació

1. Si hem anomenat token.py al mòdul (el nom pot ser un altre), una possible solució pot ser:

```
# -*- coding: utf-8 -*-
import token

token1=token.Token(“nens“,“nen“,“NCMS“)

print token1.d_pseudoarrel()
print token1.d_pseudoterminacio()
```

2. Una possible solució és la següent (recordeu que perquè funcioni heu d'estar connectat a Internet):

```
import translate

sl=raw_input('Entra la llengua de partida: ')
tl=raw_input('Entra la llengua d'arribada: ')
frase_sl=raw_input('Entra la frase a traduir: ')

frase_tl=translate.translate(sl,tl,unicode(frase_sl,'utf-8'))
print frase_tl
```

3. A tranlate.py cal afegir (per exemple, abans de `def translate`):

```
def sl_codes():
    return source_languages

def tl_codes():
    return target_languages
```

I el programa que utilitza aquestes funcions pot ser com segueix:

```
import translate

sl=translate.sl_codes()
tl=translate.tl_codes()

print “SOURCE LANGUAGES“
for clau in sl.keys():
   print clau+“ “+sl[clau]

print “TARGET LANGUAGES“
for clau in tl.keys():
   print clau+“ “+tl[clau]
```

11. Entrada/Sortida i fitxers

Els nostres programes serien de molt poca utilitat si no fossin capaços d'interaccionar amb l'usuari. En apartats anteriors vam veure, de passada, l'ús de la paraula clau `print` per a mostrar missatges a la pantalla.

En aquest apartat, a més de descriure més detalladament l'ús de `print` per a mostrar missatges a l'usuari, aprendrem a utilitzar les funcions `input` i `raw_input` per a demanar informació, així com els arguments de la línia d'ordres i, finalment, l'entrada/sortida de fitxers.

11.1. Entrada estàndard

La forma més senzilla d'obtenir informació per part de l'usuari és mitjançant la funció `raw_input`. Aquesta funció pren com a paràmetre una cadena que s'utilitza com a indicador (és a dir, com a text que es mostrarà a l'usuari demanant l'entrada) i retorna una cadena amb els caràcters introduïts per l'usuari fins que prem la tecla Enter. Vegem un petit exemple:

```
nom = raw_input(“Com et dius? ")

print “Encantat, “ + nom
```

Si necessitéssim un número enter com a entrada en comptes d'una cadena, per exemple, podríem utilitzar la funció `int` per a convertir la cadena a enter, tot i que seria convenient tenir en compte que pot llençar una excepció si l'usuari no introdueix un número.

```
try:
    edat = raw_input("Quants anys tens? ")
    dies = int(edat) * 365
    print “Has viscut “ + str(dies) + “ dies”
except ValueError:
    print “Aixo no es un numero”
```

La funció `input` és una mica més complicada. El que fa aquesta funció és utilitzar `raw_input` per a llegir una cadena de l'entrada estàndard i després passa a avaluar-la com si fos codi Python; per tant s'ha d'anar amb compte amb `input`.

11.2. Paràmetres de la línia d'ordres

A més de l'ús d'`input` i `raw_input`, el programador Python compta amb altres mètodes per a obtenir dades de l'usuari. Un d'ells és l'ús de paràmetres a l'hora de cridar el programa amb la línia d'ordres. Per exemple:

```
python editor.py hola.txt
```

En aquest cas `hola.txt` seria el paràmetre, al qual es pot accedir a través de la llista `sys.argv`, encara que, com és de suposar, abans de poder utilitzar aquesta variable hem d'importar el mòdul `sys`.

`sys.argv[0]` conté sempre el nom del programa tal com l'ha executat l'usuari, `sys.argv[1]`, si existeix, seria el primer paràmetre; `sys.argv[2]`, el segon, i així successivament.

```
import sys
if(len(sys.argv) > 1):
    print “Obrint “ + sys.argv[1]
else:
    print “Has d'indicar el nom de l'arxiu”
```

Hi ha mòduls, com `optparse`, que faciliten el treball amb els arguments de la línia d'ordres, però explicar el seu ús queda fora de l'objectiu d'aquest apartat.

11.3. Sortida estàndard

La manera més senzilla de mostrar una cosa a la sortida estàndard és mitjançant l'ús de la sentència `print`, com ja hem vist moltes vegades en exemples anteriors. En la seva forma més bàsica, a la paraula clau `print` segueix una cadena, que es mostrarà com a sortida quan s'executi la instrucció.

```
>>> print “Hola mon”

Hola mon
```

Després d'imprimir la cadena passada com a paràmetre, el cursor se situa a la següent línia de la pantalla, per la qual cosa el `print` de Python funciona igual que el `println` de Java.

En algunes funcions equivalents d'altres llenguatges de programació cal afegir un caràcter de nova línia per a indicar explícitament que volem passar a la

línia següent. Aquest és el cas de la funció `printf` de C o la mateixa funció `print` de Java.

Ja vam explicar l'ús d'aquests caràcters especials durant l'explicació del tipus cadena en l'apartat sobre els tipus bàsics de Python. La sentència següent, per exemple, imprimiria la paraula "Hola" seguida d'una línia buida (dos caràcters de nova línia, $\backslash n$) i, a continuació, la paraula "mon" amb un sagnat (un caràcter tabulador, $\backslash t$).

```
print "Hola\n\n\tmon"
```

Perquè la impressió següent es realitzi en la mateixa línia hauríem de posar una coma al final de la sentència. Comparem el resultat d'aquest codi:

```
>>> for i in range(3):
>>> ...print i,
0 1 2
```

amb el d'aquest altre, en el qual no s'utilitza una coma al final de la sentència:

```
>>> for i in range(3):
>>> ...print i
0
1
2
```

Aquest mecanisme de col·locar una coma al final de la sentència funciona ja que és el símbol que s'utilitza per a separar cadenes que volem imprimir en la mateixa línia.

```
>>> print "Hola", mon
Hola mon
```

En això es diferencia de l'ús de l'operador + per a concatenar les cadenes, ja que quan utilitzem les comes, `print` introdueix automàticament un espai per

a separar cada una de les cadenes. Aquest no és el cas quan s'utilitza l'operador +, ja que el que li arriba a `print` és un sol argument: una cadena ja concatenada.

```
>>> print “Hola” + “mon”

Holamon
```

A més, en utilitzar l'operador + hauríem de convertir abans cada argument en una cadena, si és que no ho són, ja que no és possible concatenar cadenes i altres tipus, mentre que quan es fa servir el primer mètode no és necessària la conversió.

```
>>> print “Val”, 3, “euros”

Val 3 euros

>>> print “Val” + 3 + “euros”

<type ‘exceptions.TypeError’>: cannot concatenate ‘str’ and ‘int’ objects
```

La sentència `print`, o més aviat les cadenes que imprimeix, permeten també utilitzar tècniques avançades de formatació, de manera similar al `sprintf` de C. Vegem-ne un exemple bastant simple:

```
print “Hola %s” % “mon”

print “%s %s” % (“Hola”, “mon”)
```

El que fa la primera línia és introduir els valors a la dreta del símbol % (la cadena “mon”) en les posicions indicades pels especificadors de conversió de la cadena a l'esquerra del símbol %, després de convertir al tipus adequat.

A la segona línia, veiem com es pot passar més d'un valor a substituir, mitjançant una tupla.

En aquest exemple només tenim un especificador de conversió: %s.

Els especificadors més senzills estan formats pel símbol % seguit d'una lletra que indica el tipus amb què formatar el valor (vegeu la taula 4).

Es pot introduir un número entre el % i el caràcter que indica el tipus a què volem formatar, de manera que indiqui el nombre mínim de caràcters que volem que ocupi la cadena. Si la mida de la cadena resultant és menor que aquest

Taula 4. Especificadors per al formateig de la sortida

Especificador	Format
%s	Cadena
%d	Enter
%o	Octal
%x	Hexadecimal
%f	Real

número, s'afegiran espais a l'esquerra de la cadena. En el cas que el nombre sigui negatiu, es farà exactament el mateix, només que els espais s'afegiran a la dreta de la cadena.

```
>>> print "%10s mon" % "Hola"

______Hola mon

>>> print "%-10s mon" % "Hola"

Hola_______mon
```

En el cas dels reals és possible indicar la precisió que volem utilitzar precedint la `f` d'un punt seguit del nombre de decimals que volem mostrar:

```
>>> from math import pi

>>> print "%.4f" % pi

3.1416
```

La mateixa sintaxi es pot utilitzar per a indicar el nombre de caràcters de la cadena que volem mostrar:

```
>>> print "%.4s" % "hola mom"

hola
```

11.4. Arxius

Els fitxers en Python són objectes de tipus `file` creats mitjançant la funció `open` ('obrir'). Aquesta funció pren com a paràmetres una cadena amb la ruta al fitxer que volem obrir, que pot ser relativa o absoluta, una cadena opcional indicant el mode d'accés (si no s'especifica, s'accedeix en mode de lectura) i, finalment, un enter opcional per a especificar una mida de memòria intermèdia diferent de la utilitzada per defecte.

El mode d'accés pot ser qualsevol combinació lògica dels següents modes:

- "r": *read*, lectura. Obre el fitxer en mode de lectura. L'arxiu ha d'existir prèviament, en cas contrari es llençarà una excepció de tipus `IOError`.
- "w": *write*, escriptura. Obre el fitxer en mode escriptura. Si l'arxiu no existeix es crea. Si existeix, sobreescriu el contingut.
- "a": *append*, afegeix. Obre el fitxer en mode escriptura. Es diferencia del mode "w" en què, en aquest cas, no es sobreescriu el contingut de l'arxiu, sinó que es comença a escriure al final de l'arxiu.
- "b": *binary*, binari.
- "+": permet lectura i escriptura simultànies.
- "U": *universal newline*, salts de línia universals. Permet treballar amb arxius que tinguin un format per als salts de línia que no coincideixi amb el de la plataforma actual (a Windows s'utilitza el caràcter CR LF, a Unix, LF, i a Mac, OS CR).

```
f = open("arxiu.txt", "w")
```

Després de crear l'objecte que representa el nostre arxiu mitjançant la funció `open` podrem realitzar les operacions de lectura/escriptura pertinents utilitzant els mètodes de l'objecte que veurem en les subapartats següents.

Un cop acabem de treballar amb l'arxiu hem de tancar-lo utilitzant el mètode `close`.

11.4.1. Lectura d'arxius

Per a la lectura d'arxius s'utilitzen els mètodes `read`, `readline` i `readlines`.

El mètode `read` retorna una cadena amb el contingut del fitxer o bé el contingut dels primers *n* octets, si s'especifica la mida màxima a llegir.

```
complet = f.read()

part = f2.read(512)
```

El mètode `readline` serveix per a llegir les línies del fitxer una per una. És a dir, cada vegada que es crida aquest mètode, es retorna el contingut de l'arxiu des del punter fins que es troba un caràcter de nova línia, incloent-lo.

```
while True:
    linia = f.readline()
    if not linia: break
        print linia
```

Per acabar, `readlines`, funciona llegint totes les línies del fitxer i retornant una llista amb les línies llegides.

11.4.2. Escriptura d'arxius

Per a l'escriptura d'arxius s'utilitzen els mètodes `write` i `writelines`. Mentre que el primer funciona escrivint en el fitxer una cadena de text que pren com a paràmetre, el segon pren com a paràmetre una llista de cadenes de text que indiquen les línies que volem escriure en el fitxer.

11.4.3. Moure el punter de lectura/escriptura

Hi ha situacions en què ens pot interessar moure el punter de lectura/escriptura a una posició determinada de l'arxiu. Per exemple, si volem començar a escriure en una posició determinada i no al final o al principi de l'arxiu.

Per a això s'utilitza el mètode `seek`, que pren com a paràmetre un número positiu o negatiu que s'utilitzarà com a desplaçament. També és possible utilitzar un segon paràmetre per a indicar des d'on volem que es faci el desplaçament: 0 indicarà que el desplaçament es refereix al principi del fitxer (comportament per defecte), 1 es refereix a la posició actual, i 2, al final del fitxer.

Per a determinar la posició en què es troba actualment el punter s'utilitza el mètode `tell()`, que retorna un enter que indica la distància en bytes des del principi del fitxer.

11.5. Exercicis d'autoavaluació

1. Escriviu un programa que demani el nom d'un arxiu i escrigui el seu contingut a la pantalla. Si l'arxiu no existeix, ha de donar un missatge d'error.

2. Modifiqueu el programa perquè retorni una llista de totes les paraules que apareixen a l'arxiu i la seva freqüència d'aparició.

3. Modifiqueu el programa perquè la sortida estigui ordenada alfabèticament.

4. Modifiqueu el programa perquè la llista de paraules les doni ordenades de més a menys freqüència d'aparició.

11.6. Solució als exercicis d'autoavaluació

1. Una possible solució és la següent:

```
nom=raw_input('Escriu el nom de l'arxiu: ')

try:
    arxiu=open(nom,"r")
    while True:
        linia=arxiu.readline()
        if not linia:break
        print linia
except:
    print 'No he pogut obrir l'arxiu'
```

2. Una possible solució és la següent:

```
nom=raw_input('Escriu el nom de l'arxiu: ')
diccionari={}
try:
    arxiu=open(nom,"r")
    while True:
        linia=arxiu.readline()
        if not linia:break
        for paraula in linia.split(" "):
            if diccionari.has_key(paraula):
                diccionari[paraula]+=1
            else:
                diccionari[paraula]=1
except:
    print "No he pogut obrir l'arxiu"

for paraula in diccionari.keys():
    print diccionari[paraula], paraula
```

3. Una possible solució és la següent:

```
nom=raw_input('Escriu el nom de l'arxiu: ')
diccionari={}
try:
    arxiu=open(nom,"r")
    while True:
        linia=arxiu.readline()
        if not linia:break
```

```
        for paraula in linia.split(“ “):
            if diccionari.has_key(paraula):
                diccionari[paraula]+=1
            else:
                diccionari[paraula]=1
except:
    print “No he pogut obrir l’arxiu“

llista=diccionari.keys()
llista.sort()
for paraula in llista:
    print diccionari[paraula], paraula
```

4. Una possible solució és la següent:

```
nom=raw_input(’Escriu el nom de l’arxiu: ’)
diccionari={}
try:
    arxiu=open(nom,“r“)
    while True:
        linia=arxiu.readline()
        if not linia:break
        for paraula in linia.split(“ “):
            if diccionari.has_key(paraula):
                diccionari[paraula]+=1
            else:
                diccionari[paraula]=1
except:
    print “No he pogut obrir l’arxiu“

llista=diccionari.keys()
llista.sort(lambda x, y: diccionari[y]-diccionari[x])
for paraula in llista:
    print diccionari[paraula], paraula
```

12. Expressions regulars

Les expressions regulars, també anomenades *regex* o *regexp*, són patrons que descriuen conjunts de cadenes de caràcters.

Una cosa semblant seria escriure a la línia d'ordres de Windows:

```
dir *.exe
```

`*.exe` seria una expressió regular que descriuria totes les cadenes de caràcters que comencen amb qualsevol cosa seguida de .exe, és a dir, tots els arxius .exe.

El treball amb expressions regulars en Python es realitza mitjançant el mòdul `re`, que data de Python 1.5 i que proporciona una sintaxi per a la creació de patrons similar a la de Perl. En Python 1.6 el mòdul es va reescriure per a incloure el suport de cadenes d'Unicode i millorar-ne el rendiment.

El mòdul `re` conté funcions per a cercar patrons dins d'una cadena (`search`), verificar si una cadena s'ajusta a un determinat criteri descrit mitjançant un patró (`match`), dividir la cadena fent servir les ocurrències del patró com a punt de tall (`split`) o per a substituir totes les ocurrències del patró per una altra cadena (`sub`). Veurem aquestes funcions i alguna altra en el proper subapartat, però per ara, aprendrem alguna cosa més sobre la sintaxi de les expressions regulars.

12.1. Patrons

L'expressió regular més senzilla és una cadena simple, que descriu un conjunt format només per aquesta mateixa cadena. Per exemple, veiem com la cadena "python" coincideix amb l'expressió regular "python" usant la funció `match`:

```
import re

if re.match("python", "python"):
    print "cert"
```

Si volguéssim comprovar si la cadena és “python”, “Jython”, “cython” o qualsevol altra cosa que acabi amb “ython”, podríem utilitzar el caràcter comodí, el punt ‘.’:

```
re.match(“.ython”, “python”)

re.match(“.ython”, “jython”)
```

L’expressió regular “.ython” descriuria totes les cadenes que consisteixen en un caràcter qualsevol, menys el de nova línia, seguit de “ython”. Un caràcter qualsevol i només un; no zero ni dos ni tres.

En el cas que necessitéssim el caràcter ‘.’ a l’expressió regular, o qualsevol altre dels caràcters especials que veurem a continuació, hauríem d’escapar utilitzant la barra invertida.

Per a comprovar si la cadena està formada per 3 caràcters seguits d’un punt, per exemple, podríem utilitzar el codi següent:

```
re.match(“...\.”, “abc.”)
```

Si necessitéssim una expressió que només resultés certa per a les cadenes “python”, “Jython” i “cython” però per a cap altra, podríem utilitzar el caràcter ‘|’ per expressar alternativa, escrivint els tres subpatrons complets:

```
re.match(“python|jython|cython”, “python”)
```

o bé tan sols la part que pugui canviar, tancada entre parèntesis, formant el que es coneix com a *grup*. Els grups tenen una gran importància a l’hora de treballar amb expressions regulars i aquest no és el seu únic ús, com veurem en el següent subapartat.

```
re.match(“(p|j|c)ython”, “python”)
```

Una altra opció consistiria a tancar els caràcters ‘p’, ‘j’ i ‘c ’entre claudàtors per a formar una classe de caràcters, indicant que en aquesta posició es pot situar qualsevol dels caràcters de la classe.

```
re.match(“[pjc]ython”, “python”)
```

I si volguéssim comprovar si la cadena és “python”, “python1”, “python2”, ..., “Python9”? En lloc d’haver de tancar els 10 dígits dins dels claudàtors podem

utilitzar el guió, que serveix per a indicar rangs. Per exemple "a-d" indicaria totes les lletres minúscules de l''a' a la 'd' i "0-9" serien tots els números de 0 a 9, ambdós inclosos.

```
re.match("python[0-9]", "python0")
```

Si volguéssim, per exemple, que l'últim caràcter fos un dígit o una lletra, simplement s'escriurien dins dels claudàtors tots els criteris, un darrere l'altre.

```
re.match("python[0-9a-zA-Z]", "pythonp")
```

Cal advertir que dins de les classes de caràcters, els caràcters especials no necessiten ser escapats. Per comprovar si la cadena és "python." o "python," s'escriuria:

```
re.match("python[.,]", "python.")
```

i no

```
re.match("python[\.,]", "python.")
```

ja que en aquest últim cas estaríem comprovant si la cadena és "python.", "python," o "python\".

Els conjunts de caràcters també es poden negar utilitzant el símbol '^'. L'expressió `python[^0-9a-z]`, per exemple, indicaria que ens interessen les cadenes que comencen per "python" i que tenen com a últim caràcter una cosa que no és ni una lletra minúscula ni un número.

```
re.match("python[^0-9a-z]", "python+")
```

L'ús de [0-9] per a referir-se a un dígit no és molt comú, ja que, com que la comprovació que un caràcter és un dígit és un procés molt utilitzat, existeix una seqüència especial equivalent: "\d". Existeixen altres seqüències disponibles que llistem a continuació:

- \d: un dígit. Equival a [0-9].
- \D: qualsevol caràcter que no sigui un dígit. Equival a [^ 0-9].
- \w: qualsevol caràcter alfanumèric. Equival a [a-zA-Z0-9_].
- \W: qualsevol caràcter no alfanumèric. Equival a [^ a-zA-Z0-9_].
- \s: qualsevol caràcter en blanc. Equival a [\t\n\r\f\v].
- \S: qualsevol caràcter que no sigui un espai en blanc. Equival a [^\$t\n\r\f\v$].

Vegem ara com representar repeticions de caràcters, atès que no seria de molta utilitat, per exemple, haver d'escriure una expressió regular amb 30 caràc-

ters \d per a cercar números de 30 dígits. Per a aquesta tasca tenim els caràcters especials +, * i ?, a més de les claus {}.

El caràcter + indica que el que tenim a l'esquerra, sigui un caràcter com 'a', una classe com a '[abc]' o un subpatró com '(abc)', pot trobar-se una o més vegades. Per exemple, l'expressió regular "python+" descriuria les cadenes "python", "pythonn" i "pythonnn", però no "pytho", ja que hi ha d'haver, com a mínim, una 'n'.

El caràcter * és similar a +, però en aquest cas el que se situa a la seva esquerra pot trobar-se zero o més vegades.

El caràcter ? indica opcionalitat, és a dir, el que tenim a l'esquerra pot aparèixer o no (pot aparèixer 0 o 1 vegades).

Finalment, les claus serveixen per a indicar el nombre exacte de vegades que pot aparèixer el caràcter de l'esquerra, o bé un rang de vegades que pot aparèixer. Per exemple {3} indicaria que ha d'aparèixer exactament 3 vegades, {3,8} indicaria que ha d'aparèixer de 3 a 8 vegades, {,8}, de 0 a 8 cops i {3,}, tres vegades o més (les que siguin).

Un altre element interessant en les expressions regulars, per acabar, és l'especificació de les posicions en què s'ha de trobar la cadena. Aquesta és la utilitat de '^' i '$', que indiquen, respectivament, que l'element sobre el qual actuen ha d'anar al principi de la cadena o al final d'aquesta.

La cadena "http://mundogeek.net", per exemple, s'ajustaria a l'expressió regular ^*http*, mentre que la cadena "El protocol http" no ho faria, ja que "http" no es troba al principi de la cadena.

12.2. Utilització del mòdul `re`

Ja hem vist per sobre com s'utilitza la funció `match` del mòdul `re` per a comprovar si una cadena s'ajusta a un determinat patró. El primer paràmetre de la funció és l'expressió regular, el segon, la cadena a comprovar i hi ha un tercer paràmetre opcional que conté diferents indicadors, o *flags*, que es poden utilitzar per a modificar el comportament de les expressions regulars.

Alguns exemples d'indicadors del mòdul `re` són `re.IGNORECASE`, que fa que no es tingui en compte si les lletres són majúscules o minúscules, o també `re.VERBOSE`, que fa que s'ignorin els espais i els comentaris a la cadena que representa l'expressió regular.

El valor de retorn de la funció serà `None`, en cas que la cadena no s'ajusti al patró, o un objecte de tipus `MatchObject` en cas contrari. Aquest objecte compta amb mètodes `start` i `end` que retornen la posició en què comença

i acaba la subcadena reconeguda i mètodes `group` i `groups` que permeten accedir als grups que han propiciat el reconeixement de la cadena.

Quan cridem el mètode `group` sense paràmetres se'ns retorna el grup 0 de la cadena reconeguda. El grup 0 és tota la subcadena reconeguda per l'expressió regular, encara que no existeixin parèntesis que delimitin el grup.

```
>>> mo = re.match("http://.+\net", "http://mundogeek.net")

>>> print mo.group()

http://mundogeek.net
```

Podríem crear grups utilitzant els parèntesis, com vam aprendre al subapartat anterior, i així obtindríem la part de la cadena que ens interessa.

```
>>> mo = re.match("http://(.+)\net", "http://mundogeek.net")

>>> print mo.group(0)

http://mundogeek.net

>>> print mo.group(1)

mundogeek
```

El mètode `groups`, per la seva banda, retorna una llista amb tots els grups, excepte el grup 0, que s'omet.

```
>>> mo = re.match("http://(.+)\(.{3})", "http://mundogeek.net")

>>> print mo.groups()

('mundogeek', 'net')
```

La funció `search` del mòdul `re` funciona de forma similar a `match`. Disposem dels mateixos paràmetres i del mateix valor de retorn; l'única diferència és que quan fem servir `match` la cadena ha d'ajustar-se al patró des del primer caràcter de la cadena, mentre que amb `search` busquem qualsevol part de la cadena que s'ajusti al patró. Per aquesta raó el mètode `start` d'un objecte `MatchObject` obtingut mitjançant la funció `match` sempre retornarà 0, mentre que en el cas de `search` això no té perquè ser així.

Una altra funció de cerca del mòdul `re` és `findall`. Aquest mètode pren els mateixos paràmetres que les dues funcions anteriors, però retorna una llista amb les subcadenes que coincideixen amb el patró.

Una altra possibilitat, si no volem totes les coincidències, és, per exemple, utilitzar `finditer`, que retorna un iterador amb què podem consultar un a un els diferents `MatchObject`.

Les expressions regulars no només permeten realitzar cerques o comprovacions, sinó que, com hem comentat anteriorment, també disposen de funcions per a dividir la cadena o realitzar reemplaçaments.

La funció `split`, sense anar més lluny, pren com a paràmetres un patró, una cadena i un enter opcional que indica el nombre màxim d'elements en què volem dividir la cadena, i utilitza el patró a manera de punts de separació per a la cadena, retornant una llista amb les subcadenes.

La funció `sub` pren com a paràmetres un patró a substituir, una cadena per a fer servir com a reemplaçament cada vegada que trobem el patró, la cadena sobre la qual realitzem les substitucions i un enter opcional que indica el nombre màxim de substitucions que volem realitzar.

Quan cridem aquests mètodes el que passa en realitat és que es crea un nou objecte de tipus `RegexObject`, que representa l'expressió regular, i es criden mètodes d'aquest objecte que tenen els mateixos noms que les funcions del mòdul.

Si volem utilitzar un mateix patró diverses vegades ens pot interessar crear un objecte d'aquest tipus i cridar els seus mètodes nosaltres mateixos; d'aquesta manera evitem que l'intèrpret hagi de crear un nou objecte cada vegada que fem servir el patró i millorarem el rendiment l'aplicació.

Per a crear un objecte `RegexObject` s'utilitza la funció `compile` del mòdul, a la qual es passa com a paràmetre la cadena que representa el patró que volem utilitzar en la nostra expressió regular i, opcionalment, una sèrie d'indicadors d'entre els que hem comentat anteriorment.

12.3. Algunes coses útils a recordar

Quan treballem amb expressions regulars, convé recordar el següent:

- ^ : indica el principi de la cadena.
- $: indica el final de la cadena.
- $\backslash b$: indica els límits d'una paraula.
- $\backslash d$: indica qualsevol caràcter numèric.
- $\backslash D$: indica qualsevol caràcter no numèric.

- *x*? : indica un caràcter *x* opcional (en altres paraules, indica zero o una vegada)
- *x*∗ : indica un caràcter *x* zero o més vegades.
- *x*+ : indica un caràcter *x* una o més vegades.
- *x*{*n*,*m*} : indica un caràcter *x* com a mínim *n* vegades, però no més d'*m* vegades.
- (*a*|*b*|*c*) : indica *a* o *b* o *c*.
- (*x*) : en general els parèntesis recorden un grup. Es pot obtenir el valor del grup mitjançant el mètode `groups()` de l'objecte retornat per `re.search`.

12.4. Exercicis d'autoavaluació

1. Escriviu un programa que obri un fitxer de text i retorni totes les línies del fitxer que continguin números.

2. Modifiqueu el programa anterior perquè, d'una llista de totes les paraules que comencen per vocal, indiqui quantes vegades apareix cada una d'aquestes paraules.

3. Escriviu un programa que demani dates en format "12/05/2003" i retorni la data en format "12 de maig de 2003". Els mesos poden donar-se tant com a 05 o com a 5. No cal verificar si les dades introduïdes són correctes o no.

12.5. Solució als exercicis d'autoavaluació

1. Si per exemple el fitxer s'anomena text.txt el programa podria ser el següent:

```
import re

file = open("text.txt","r")
text = file.readlines()
file.close()

keyword = re.compile(r"[0-9]+")

for line in text:
    if keyword.search (line):
       print line
```

2. Una possible solució és la següent:

```
import re

file = open("text.txt","r")
text = file.readlines()
file.close()
```

```
keyword = re.compile(r"^(a|e|i|o|u)",re.I)

paraulavocal={}

for line in text:
    paraules=line.split()
    for paraula in paraules:
        result = keyword.search (paraula)
        if result:
            if (paraulavocal.has_key(paraula)):
                paraulavocal[paraula]+=1
            else:
                paraulavocal[paraula]=1

lista=paraulavocal.keys()
lista.sort(lambda x, y: paraulavocal[y]-paraulavocal[x])
for palabra in lista:
    print paraulavocal[palabra], palabra
```

3. Una possible solució és la següent:

```
import re

mesos={"1":"gener","2":"febrer","3":"marc","4":"abril","5":"maig","6":"juny",
"7":"juliol","8":"agost","9":"setembre","10":"octubre","11":"novembre","12":"desembre"}

data=raw_input()

keyword = re.compile(r"((\d{1,2})/(\d{1,2})/(\d{4}))")
result = keyword.search(data)
if result:
    print "DATA ",data
    resultats=result.groups()
    dia=resultats[1]
    mes=resultats[2]
    any=resultats[3]
    #eliminem el 0 del mes si la primera xifra es zero
    if (mes[0]=="0"):
       mes=mes[1]
    print dia,"de",mesos[mes],"de",any
```

13. Conclusions

En aquest mòdul hem introduït una sèrie d'aspectes fonamentals del llenguatge de programació Python. Com tot llenguatge de programació, Python és molt ampli i és impossible recollir tota la informació en aquestes pàgines. Per això recomanem consultar una sèrie de fonts disponibles a Internet:

- La documentació oficial de Python, disponible a http://python.org/doc/ (en anglès). Alguns d'aquests documents també estan disponibles en castellà: http://pyspanishdoc.sourceforge.net/.

- El llibre *Dive into Python*: http://diveintopython.org/toc/index.html.

- El llibre complet en què es basa aquest mòdul: *Python para todos*, disponible a http://mundogeek.net/tutorial-python.

Existeixen moltes altres opcions que podreu trobar amb el vostre cercador d'Internet preferit.