# El llenguatge Python

David Masip Rodó

PID_00174126

UOC
Universitat Oberta de Catalunya

# Índex

# Introducció

En aquest mòdul aprendrem les bases del llenguatge de programació Python. Aquest mòdul no pretén ser una guia exhaustiva de totes les característiques que ens ofereix aquest llenguatge de programació, sinó que està enfocat a introduir l'estudiant en l'entorn Python. L'objectiu del text serà ajudar a comprendre els exemples que s'exposen en els mòduls de teoria, i proporcionar una base que permeti a l'estudiant elaborar les diferents activitats que requereix el curs.

Pel que fa a prerequisits, no assumirem cap coneixement previ de Python. No obstant això, és molt important tenir coneixements d'algun llenguatge de programació previ (C, C++, Java, ...), ja que els conceptes elementals (variable, bucle, funció, ...) no es tractaran en profunditat en aquest mòdul.

El llenguatge Python va ser dissenyat a la fi de la dècada dels vuitanta per Guido van Rossum. Es tracta d'un llenguatge de programació de molt alt nivell, amb una sintaxi molt clara i una aposta ferma per la llegibilitat del codi. Sens dubte és un llenguatge de programació molt versàtil, fortament tipat, imperatiu i orientat a objectes, encara que conté també característiques que el converteixen en un llenguatge de paradigma funcional.

El Python es pot considerar un llenguatge semiinterpretat. A diferència de C, el codi Python no s'executa directament en la màquina de destinació, sinó que és executat per un SW intermedi (o intèrpret). No obstant això, igual que Java, el Python compila el codi escrit en llenguatge d'alt nivell per a obtenir un pseudocodi màquina (*bytecode*), que és el que executa pròpiament l'intèrpret. Hi ha versions de l'intèrpret de Python per a la major part de les plataformes.

En aquest mòdul explicarem els passos necessaris per a instal·lar el Python en una determinada plataforma. A continuació veurem les principals característiques del llenguatge des del punt de vista sintàctic.

# Objectius

Els objectius que l'alumne ha d'haver aconseguit una vegada estudiats els continguts d'aquest mòdul són els següents:

1. Refrescar els conceptes elementals de programació apresos en cursos anteriors.
2. Aprendre les bases del llenguatge Python.
3. Ser capaç d'entendre el codi que es mostra en els exemples de teoria i poder modificar-lo per a fer altres funcionalitats.
4. Usar característiques avançades de Python que permetin utilitzar-lo com a llenguatge ràpid per al disseny i codificació de prototips d'intel·ligència artificial.

# 1. Instal·lació de Python

Actualment hi ha múltiples implementacions del llenguatge Python, en funció del llenguatge base en el qual s'ha construït l'intèrpret. La més coneguda és CPython, o simplement Python, que ha estat implementada en llenguatge C. Altres opcions disponibles són: IronPython (codificada en C#) o JPython (codificada en JAVA).

Per a instal·lar una determinada versió de Python, aconsellem recórrer al seu lloc web oficial*. En el moment d'escriure aquests manuals, aquesta URL conté tota la informació necessària per a instal·lar l'entorn. En la secció de descàrregues** es poden obtenir els últims paquets d'instal·lació per a Windows, Linux i Mac OS X en les seves diferents versions.

* http://www.python.org/

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

Una vegada obtingut el paquet per al sistema operatiu i versió adequats, executarem l'instal·lador, i en una línia d'ordres executarem l'intèrpret de Python, i provarem que tot funciona correctament mitjançant el clàssic "Hello World".

```
macbook:~ david$ python
Python 2.6.6 (r266:84292, Oct 16 2010, 21:41:03)
[GCC 4.0.1 (Apple Inc. build 5490)] on darwin
Type ''help'', ''copyright'', ''credits'' or ''license'' for
more information.
>>> print ''Hello World''
Hello World
>>> quit()
macbook:~ david$
```

En aquest exemple, hem cridat l'intèrpret de Python des del símbol de sistema. En iniciar-se, després d'imprimir la informació de versions, el Python ens mostra el símbol "`>>>`" per a indicar-nos que l'intèrpret està esperant ordres. En aquest petit exemple li demanem que ens imprimeixi la cadena de caràcters "Hello World", i després de veure'n el resultat, sortim del Python mitjançant la instrucció `quit()`, i tornem al símbol de sistema. Arribats a aquest punt, podem considerar que tenim la instal·lació bàsica de Python a punt per a utilitzar-la.

En aquest mòdul treballarem fonamentalment amb un editor de textos, i executarem els *scripts*, que programarem mitjançant l'intèrpret. En l'exemple anterior, podríem haver desat la línia `print "Hello World"` en un arxiu anomenat *Hola.py*, i executar-lo mitjançant l'intèrpret amb la instrucció:

```
macbook:~ david$ python Hola.py
Hello World
```

### Entorns de desenvolupament IDE

Hi ha la possibilitat d'instal·lar entorns de desenvolupament IDE específics per a Python. No és l'objectiu d'aquest mòdul analitzar els IDE disponibles al mercat actualment, encara que citarem els tres entorns lliures més coneguts. D'una banda PyDEV* és un conegut connector de Python per a l'entorn de desenvolupament Eclipse, bastant utilitzat en el sistema universitari. També hi ha l'opció d'usar un entorn propi, com SPE (Stani's Python Editor)** , o Wing***. En aquest mòdul treballarem directament amb l'intèrpret d'ordres, i qualsevol editor de textos simple serà suficient per a les funcionalitats que desenvoluparem.

* http://pydev.org/

** http://sourceforge.net/projects/spe/

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

# 2. Variables

Els tipus bàsics del llenguatge Python són essencialment els ja coneguts en qualsevol llenguatge de programació: els valors numèrics, les cadenes de text, i els valors booleans. En el codi 2.1 se'n mostra un exemple de cada tipus.

Codi 2.1: Exemples d'ús de variables

```
1  # Exemples de variables
2
3  a =42;   #valor enter
4  along = 42L; #valor enter long (32-64 bits en funció de la plataforma)
5  ahex = 0x2a;  #a en notació hexadecimal
6  aoctal = 052;  #a en notació octal
7
8  b = 3.1416;  #valor en coma flotant
9  bnotacion = 3.14e0;
10 c = 3 + 7j;  #Python suporta nombres complexos
11 d = "Exemple de cadena de caracters" #Una cadena de caracters
12
13 #Imprimir les variables per pantalla
14 print a,b,c,d;
15
16 #tipus de les variables
17 type(a);
18 type(b);
19 type(c);
20 type(d);
```

Els nombres enters es poden representar en notació decimal, octal (anteposant un "0" al valor) o hexadecimal (anteposant "0x" al valor). Igual que en llenguatge C (en el qual està escrit el Python), els nombres es poden representar mitjançant enters (per defecte) o *long* (enter llarg), que permet un rang més gran de valors possibles. Aquest rang, de nou com en C, dependrà de la plataforma subjacent, i pot ser de 32 o 64 bits.

Els valors flotants són implementats a baix nivell directament amb el tipus *double* de C (registre de 64 bits). Per la seva banda, els nombres complexos estan suportats de base i en la pràctica s'implementen mitjançant dos flotants, un per a la part real i un altre per a la part imaginària.

## 2.1. Operadors

Els operadors que suporta el llenguatge Python són els clàssics de qualsevol llenguatge de programació. Les taules 1 i 2 mostren un resum amb exemples d'aquestes operacions. A més, el Python proveeix extensions per a operacions més complexes en el mòdul *math*.

Taula 1. Operadors amb els tipus bàsics

| Nombre | Suma | Resta o negació | Multiplicació | Divisió | Mòdul | Divisió entera | Exponenciació |
|---|---|---|---|---|---|---|---|
| Operador | + | - | * | / | % | // | ** |
| Exemple | $a = 1 + 2$<br>$\#a = 3$ | $a = 2 - 5$ o $a = -3$<br>$\#a = -3$ | $a = 2 * 3$<br>$\#a = 6$ | $a = 5.4/2$<br>$\#a = 2.7$ | $a = 5\%2$<br>$\#a = 1$ | $a = 5.8//2$<br>$\#a = 2.0$ | $a = 3 * *4$<br>$\#a = 81$ |

Taula 2. Operadors a escala de bit

| Nombre | And | Or | Xor | Not | Desplaçament a la dreta | Desplaçament a l'esquerra |
|---|---|---|---|---|---|---|
| Operador | & | \| | $\wedge$ | $\sim$ | $\gg$ | $\ll$ |
| Exemple | $a = 2\&1$<br>$\#a = 0(10\&01)$ | $a = 2\|1$<br>$\#a = 3(10\|01)$ | $a = 3 \wedge 1$<br>$\#a = 1$ | $a = \sim 1$<br>$\#a = -2$ | $a = 4 \gg 2$<br>$\#a = 1$ | $a = 1 \ll 2$<br>$\#a = 4$ |

El valor de retorn d'un operador en Python està determinat pel tipus de les variables que intervenen en l'operació. Així, per exemple, si sumem dos nombres enters, ens retornarà un altre valor enter. Si un dels dos operands és un valor en punt flotant, el resultat de l'operació serà també un valor en punt flotant.

El tipus bàsic booleà pot rebre dos valors (*true, false*), i s'utilitzen fonamentalment per a expressar el resultat de condicions, especialment útils en els bucles i control de flux condicional. En la taula 3 es resumeixen els diferents operadors que treballen amb valors booleans, comuns a la major part de llenguatges de programació.

Taula 3. Operadors amb booleans

| Operador | And | Or | Not | == | ! = | < | > |
|---|---|---|---|---|---|---|---|
| Exemple | *b = False and True* | *b = False or True* | *b = Not False* | $8 == 9$ | $8! = 9$ | $8 < 9$ | $8 > 9$ |
| Resultat | b serà False | b serà True | a=n serà True | False | True | True | False |

## 2.2. Cadenes de caràcters

Les cadenes de caràcters són fragments de text delimitats en Python per cometes simples ('Exemple de cadena') o dobles ("Exemple de cadena"). Si volem introduir salts de línia, disposem del caràcter d'escapament '\n'. També podem usar altres caràcters d'escapament tradicionals com '\t' (tabulació) o '\b' (esborra el caràcter). Per a aconseguir que el text s'imprimeixi tal com apareix en el codi font sense haver de recórrer als codis d'escapament, el podem delimitar amb cometes triples (en podeu veure un exemple en el codi 2.2).

Codi 2.2: Exemples d'ús de cadenes de caràcters

```
1  # Exemples de cadenes
2
3  a = "hola\n"
4  b = "\t Aquest és un exempp\ble de cadenes de caràcters\n"
5  c = """És possible escriure salts de línia
6  sense necessitat de codis d'escapament."""
7
8  print a,b,c
9
10 >>>hola
```

```
11          Aquest és un exemple de cadenes de caràcters
12 És possible escriure salts de línia
13 sense necessitat de codis d'escapament.
14
15 d␣=␣a␣+␣b;␣␣#concatenació
16 e␣=␣"repite"
17 f␣=␣3*e;␣␣␣␣#repetició
18 g␣=␣e*3;␣␣␣␣#equivalent␣al␣cas␣anterior
19
20 print␣d,f,g
```

Alguns operadors usats amb valors numèrics es troben sobrecarregats en el cas de les cadenes de caràcters. En són exemples la igualtat (que assigna una cadena a una variable) i l'operador suma "+". La suma de dues cadenes és el resultat de concatenar la segona darrere de la primera. De la mateixa manera, el producte d'una cadena per un escalar dóna per resultat la mateixa cadena repetida tantes vegades com indiqui l'operador numèric.

Finalment, una funció molt útil de Python és **str**, que permet fer la conversió de valors numèrics a cadena de caràcters.

**Exemple**

Si teclegem *str(8.987)* en la línia d'ordres, obtindrem la sortida *'8.987'*.

# 3. Control de flux

Tots els llenguatges de programació posen a la nostra disposició instruccions de control de flux. Aquestes instruccions permeten al programador alterar l'ordre seqüencial del codi amb la finalitat de permetre executar diferents ordres en funció d'una sèrie de condicions sobre l'estat. El Python ofereix els dos tipus bàsics de sentències de control de flux: les **sentències condicionals** i els **bucles** (o repeticions).

## 3.1. Sentències condicionals: la instrucció *if*

La instrucció condicional **if** (en anglès, *si*) rep com a entrada una expressió booleana, i serveix per a executar una porció de codi en funció de si es compleix aquesta condició (el resultat de l'avaluació és *true*).

La sintaxi de la instrucció consisteix en la paraula clau **if**, a continuació l'expressió booleana de la condició, i un signe :, que indica el final de la condició, i finalment, el codi per executar en cas que la condició s'avaluï a *true*.

En el codi 3.1 teniu un exemple d'instrucció **if**.

Codi 3.1: Exemple de sintaxi d'un bloc *if*

```
1 # Exemples d'ús de if
2
3 c = 37;
4 if c > 0:
5         print "El␣nombre␣és␣positiu\n"
6         print "Que␣tingui␣un␣bon␣dia\n"
7 # continuació del programa
```

Un detall molt important que cal observar en aquest exemple de codi és la **sagnia**. En Python els blocs de codi es delimiten mitjançant una sagnia correcta. A diferència d'altres llenguatges, en què disposem de paraules clau específiques (*begin*, *end*,...) o claus ( {,..., }) reservades per a definir blocs de codi, en Python només podem usar sagnies.

La sagnia és un tret molt característic del codi Python, i permet, entre altres coses, una lectura molt més agradable dels programes, i una identificació fàcil de les diferents parts.

### 3.1.1. Sentències *if-elif-else*

Quan l'objectiu de la sentència condicional és dividir l'execució del codi en funció de si es compleix la condició o no, i volem que es faci alguna acció de manera explícita quan la condició no es compleix, usem la paraula clau **else** (sempre seguida de **:**).

Hi ha una tercera paraula clau en les instruccions de control de flux condicionals, **elif**, que s'utilitza per a afegir més condicions a la sentència *if*. Això ens pot ser molt útil si tenim molts casos per diferenciar i els volem tractar tots. Es poden afegir tants blocs *elif* com vulguem. En el codi 3.2 podem veure un exemple d'ús de l'entorn *if-elif-else*.

Codi 3.2: Exemple de sintaxi d'un bloc *else*

```
1  # Exemples d'ús de if
2
3  c = -10;
4  if c > 0:
5          print "El nombre és positiu\n"
6          print "Que tingui un bon dia\n"
7  elif c == 0:
8          print "El nombre és exactament 0"
9  else:
10         print "El nombre és negatiu"
11         print "Que tingui sort"
12 # continuació del programa
```

Si el valor de *c* fos 0, s'executaria la línia `print "El nombre és exactament 0"`. Quan no es compleix la condició de *if*, ni de cap dels *elif* que hi pot haver, s'acaba executant el bloc d'instruccions corresponents a **else**.

Finalment, hi ha una manera compacta de representar sentències condicionals (de manera similar a com vam usar *Cond ? acció si A és cert: acció si Cond és fals* en C) en Python. Es tracta dels blocs **Accio1 if Cond else Accio2**. En aquest cas, es comprova la condició *Cond* i s'executa *Accio1* si és *true* o *Accio2* si és *false*.

## 3.2. Seqüències iteratives: bucles

Els bucles són estructures de control de flux que permeten repetir un bloc de codi un nombre de vegades determinat o indeterminat. Essencialment hi ha dos tipus de bucles: **for** i **while**.

### 3.2.1. Bucles *for ... in*

El bucle **for...in** s'utilitza per a executar una seqüència de passos un nombre de vegades determinat (ja conegut prèviament).

La sintaxi que s'utilitza és: **for** element **in** seqüència: i a continuació el bloc d'accions amb la sagnia correcta.

En l'exemple del codi 3.3 podem veure un ús típic del bucle *for* per a recórrer un vector de dades.

Codi 3.3: Exemple de sintaxi d'un bucle *for...in*

```
1 # Exemples d'ús del bucle for
2
3 vector = ["hola", "bon", "dia"];
4 for paraula in vector:
5         print paraula
6 # imprimiria totes les paraules del vector
```

És important destacar que el bucle *for* no es comporta exactament com estem acostumats en la major part de llenguatges de programació. En Python, a cada pas del bucle s'instancia l'element iterador amb un valor de la seqüència de manera automàtica.

En l'exemple, la variable *paraula* anirà agafant a cada pas el valor d'un dels elements de la llista *vector* (més endavant veurem més detalls sobre el funcionament de les llistes): en la primera iteració *paraula* prendrà el valor "hola", en la segona valdrà automàticament "bons", i així successivament. No serà necessari treballar amb índexs, ja que el Python farà tot el treball per nosaltres.

### 3.2.2. while

El bucle *while* segueix un plantejament molt més similar als llenguatges de programació tradicionals. Aquest bucle ens permet executar un determinat bloc de codi mentre una determinada condició sigui certa.

La seva sintaxi és: **while** Condició:

En l'exemple del codi 3.4 es mostra un ús típic del bucle *while*.

Codi 3.4: Exemple de sintaxi d'un bucle *while*

```
1 # Exemples d'ús del bucle for
2
3 parells = 2;
4 while parells <= 20:
5         print parells
6         parells = parells + 2
7 # imprimiria tots els nombres parells
```

L'exemple imprimeix per pantalla tots els nombres parells del 2 al 20. A cada pas incrementa la variable de control que ens permetrà sortir del bucle quan la condició no sigui satisfeta.

Noteu que la sentència *while* pot generar bucles infinits. De vegades aquests bucles es generen de manera involuntària, la qual cosa va en detriment de la qualitat dels nostres programes. Altres vegades es generen de manera voluntària, en combinació amb la sentència **break**. Aquesta sentència permet sortir de manera incondicional d'un bucle.

El codi 3.5 és equivalent a l'anterior, encara que molt menys elegant. La sentència **break**, no obstant això, pot ser útil quan el flux d'execució depèn d'alguna entrada de l'usuari no prevista en el moment d'execució.

Codi 3.5: Exemple de sintaxi d'un bucle *while*

```
1 # Exemples d'ús del bucle for
2
3 parells = 2;
4 while parells > 0:
5         print parells
6         parells = parells + 2
7         if parells > 20:
8                 break
9 # imprimiria tots els nombres parells
```

# 4. Funcions

Tots els llenguatges de programació ofereixen mecanismes que permeten encapsular una sèrie d'operacions de manera parametritzable, i retornar uns resultats determinats. Aquests fragments de codi es denominen *funcions*.

En Python les funcions es declaren mitjançant la paraula clau **def**. La seva sintaxi és:

**def** nom_funcio(parametre1, parametre2, ..., parametreN):

A continuació s'escriu un salt de línia i les operacions associades a la funció, com sempre amb una sagnia a la dreta (tabulació), que indica que aquest codi es correspon a la funció.

**Delimitació de codi font**

Cal recordar que el Python no usa claus ni paraules clau per a delimitar codi font.

Una vegada s'ha definit la funció, es pot cridar amb diferents paràmetres dins del codi.

La crida o execució d'una funció segueix la sintaxi: nom_funcio(parametre1, parametre2,..., parametreN).

## 4.1. Paràmetres d'entrada

Les crides a funcions reben com a arguments una llista de paràmetres (implementada en Python com un tuple), que es corresponen amb els paràmetres establerts en el moment de la definició de la funció. En l'exemple del codi 4.1 es pot observar la crida a una funció que imprimeix el resultat d'elevar un nombre base (*parametre1*) a un exponent (*parametre2*).

Codi 4.1: Exemple de sintaxi d'una funció

```
1 # Exemples de definicions i crides de funcions
2 # Funció potència de dos valors (usa com a base el primer paràmetre
3 # i com a exponent el segon paràmetre)
4
5 def exponenciar(parametre1, parametre2):
```

```
6         ""Exemple de funció que eleva un nombre o base a un exponent
7         print parametre1**parametre2
8
9   def exponenciar2(parametre1, parametre2=2):
10        print parametre1**parametre2
11
12  exponenciar(2,3);
13  exponenciar(parametre2=3,parametre1=2);
14  exponenciar(3,2);
15  exponenciar(parametre2=2,parametre1=3);
16
17  exponenciar2(5);
18  exponenciar2(5,3);
19
20  #crides errònies
21  exponenciar(2,3,4);
```

En primer lloc noteu que la sagnia torna a tenir un paper clau. Per a diferenciar el final de la definició de la funció i la resta de codi del programa principal, n'hi ha prou de mirar el codi font per damunt. La sagnia delimita clarament els blocs.

També es pot observar una cadena inicial que comença per " ". Aquestes cadenes s'utilitzen per a documentar les funcions. Els programadors familiaritzats amb el Java poden veure cert paral·lelisme amb el *doc* utilitzat en aquest llenguatge.

Al final de la definició de la funció es troben les crides. És interessant observar les diferents maneres com es poden parametritzar les funcions en Python. L'ús normal sol ser cridar la funció amb els paràmetres en el mateix ordre en què s'ha definit la funció. No obstant això, es pot cridar també la funció amb el nom del paràmetre i el seu valor associat. D'aquesta manera són possibles crides que se salten la regla de l'ordre, com les de l'exemple, en què trobem *exponenciar(parametre2=3, parametre1=2)*, que és equivalent a *exponenciar(2,3)*. És important destacar que el Python obliga a cridar les funcions usant exactament el mateix nombre d'arguments, i en cas contrari es generarà un error d'execució.

Hi ha també la possibilitat d'inicialitzar paràmetres amb valors per defecte en cas que no se'n proporcioni el valor en el moment de la crida. La funció *exponenciar2* mostra un exemple en què en cas de no proporcionar un argument utilitzaria un 2 per defecte (elevaria al quadrat).

El Python permet també funcions amb un nombre variable de paràmetres. En l'exemple del codi 4.2 es pot veure la sintaxi utilitzada en una funció que imprimeix la suma de tots els seus arguments.

Codi 4.2: Exemple de sintaxi d'una funció amb paràmetres variables

```
1   # Exemples de definicions i crides de funcions
2   #funció suma
3
4   def sumarLlista( *parametres):
5       resultat = 0;
```

```
6       for val in parametres:
7           resultat = resultat + val;
8       print resultat;
9
10  sumarLlista(1,2,3,4);
```

La sintaxi d'aquest exemple és una mica confusa, ja que encara no hem tractat els tipus de dades complexes, com les llistes i els tuples. Essencialment es defineix un argument que conté un tuple amb tots els paràmetres possibles per rebre. Aquest tuple es recorre en temps d'execució per a poder utilitzar els paràmetres de manera adequada.

**Vegeu també**

Les llistes i els tuples s'estudien en l'apartat 5 d'aquest mòdul.

Un últim factor que cal tenir en compte en els paràmetres a funcions en qualsevol llenguatge de programació és la modificació en temps d'execució. Tradicionalment els passos per paràmetre funcionen **per valor** o **per referència**. En el primer cas els arguments a una funció no es modifiquen en sortir-ne (en cas que el codi intern els alteri). Tècnicament, es deu al fet que realment no passem a la funció la variable en qüestió, sinó una còpia local a la funció, que és eliminada en acabar l'execució. En el cas del pas per referència es passa a la funció un punter a l'objecte (en el cas de C) o simplement una referència (en llenguatges d'alt nivell) que en permet la indirecció. D'aquesta manera, les modificacions als paràmetres que es fan dins de la funció es veuen reflectides en l'exterior una vegada acaba la funció. En Python els passos per paràmetres són en general per referència. L'excepció la conformen els tipus de dades bàsiques o immutables (enters, flotants, ...), que es passen per valor. El codi 4.3 mostra un exemple d'aquest fet.

Codi 4.3: Exemple de pas per valor i referència en Python

```
1   # Exemples de definicions i crides de funcions
2   #funció per a verificar la modificabilitat dels paràmetres
3
4   def persistenciaParametres(parametre1, parametre2):
5       parametre1 = parametre1 + 5;
6       parametre2[1] = 6;
7       print parametre1;
8       print parametre2;
9
10  nombres = [1,2,3,4];
11  valorImmutable = 2;
12  persistenciaParametres(valorImmutable,nombres);
13  print nombres;
14  print valorImmutable;
```

Com es pot observar a partir de l'execució de l'exemple, el vector de nombres sofreix les modificacions degudes al codi intern de la funció, mentre que el valor enter es modifica dins de la funció però perd aquesta modificació en retornar el control al programa principal.

### 4.2. Valors de retorn

Fins ara s'han descrit les funcions Python com a fragments de codi que efectuen unes accions determinades de manera reutilitzable. Els exemples vistos

són en el fons versions procedimentals de funcions, ja que en cap cas no es retorna un valor de retorn.

La sintaxi per al retorn de valors en Python difereix dels llenguatges de programació habituals, ja que el Python permet retornar més d'un valor de retorn. La paraula clau per utilitzar és **return**, seguida de la llista d'arguments que cal retornar.

Hi ha també la possibilitat d'obviar els valors de retorn i utilitzar el pas de paràmetres per referència, i s'obté un resultat similar. El codi 4.4 mostra tres exemples d'ús de retorn de valors en funcions. Noteu que la funció *sumaPotencia* retorna el resultat d'efectuar les dues operacions en forma de llista. En realitat la funció només retorna un valor (la llista), però l'efecte produït és el mateix que si pogués retornar múltiples valors.

Codi 4.4: Exemple de retorn de valors en Python

```
1  # Exemples de definicions i crides de funciones
2  #funció per a practicar el retorn de valors
3
4  def suma(parametre1, parametre2):
5      return parametre1 + parametre2;
6
7  def potencia(parametre1, parametre2):
8      return parametre1 ** parametre2;
9
10 def sumaPotencia(parametre1, parametre2):
11     return parametre1 + parametre2, parametre1 ** parametre2;
12
13 resultat1 = suma(2,3);
14 print resultat1;
15 resultat2 = potencia(2,3);
16 print resultat2;
17 resultat3 = sumaPotencia(2,3);
18 print resultat3;
```

# 5. Tipus de dades en Python

En l'apartat anterior hem vist els tipus elementals de dades en Python: els enters, els valors flotants, les cadenes de text i els valors booleans. En aquest apartat veurem tipus de dades més complexos que permeten treballar amb agrupacions d'aquests tipus de dades bàsiques. Especialment, veurem els tuples, les llistes, els conjunts, els diccionaris i finalment, els fitxers.

## 5.1. Tuples

Un tuple és una seqüència immutable i ordenada d'elements. Cadascun dels elements que conformen un tuple pot ser de qualsevol tipus (bàsic o no). La sintaxi per a declarar un tuple consisteix a especificar-ne els elements separats per una coma (,).

És molt habitual agrupar tots els elements d'un tuple entre parèntesis, encara que no és imprescindible (només s'exigeix l'ús del parèntesi quan hi pot haver confusió amb altres operadors). Sovint es col·loca una coma al final del tuple per a indicar la posició de l'últim element. En el codi 5.1 es poden veure diferents exemples d'ús de tuples.

Codi 5.1: Exemple de tuples en Python

```
1  # Exemples d'ús de tuples
2
3  tuple_buida = ();
4  tuple1 = 1,2,3,4,6,
5  tuple1b = (1,2,3,4,6);
6  tuple2 = 'hola',2,3
7  tuple3 = tuple2,90
8  tuple4 = tuple('exemple');
9
10 #Resultat de l'execució
11 >> print tuple1
12 (1, 2, 3, 4, 6)
13 >>> print tuple1b
14 (1, 2, 3, 4, 6)
15 >>> print tuple2
16 ('hola', 2, 3)
17 >>> print tuple2
18 ('hola', 2, 3)
19 >>> print tuple3
20 (('hola', 2, 3), 90)
21 >>> print tuple4
22 ('e', 'j', 'e', 'm', 'p', 'l', 'o')
```

## 5.2. Llistes

Una llista és una seqüència mutable i ordenada d'elements. A diferència dels tuples, els elements de les llistes es poden modificar una vegada han rebut un valor assignat. Per a especificar els elements que formen una llista, s'usa una tira d'elements separada de nou per comes (,), i al principi i al final la llista s'envolta entre claudàtors [,].

En el codi 5.2 es poden veure exemples d'usos de llistes.

Codi 5.2: Exemple de llistes en Python

```
# Exemples d'ús de llistes

llista_buida = [];
llista = [1,2,3,4,6]
llista2 = ['hola',2,3];
llista3 = [llista2,90,'good'];
llista4 = list('exemple');
llista5 = [1,2,3,4,5,6,7,8,9];

#Resultat de l'execució

>>> print llista_buida
[]
>>> print llista
[1, 2, 3, 4, 6]
>>> print llista2
['hola', 2, 3]
>>> print llista3
[['hola', 2, 3], 90, 'good']
>>> print llista4
['e', 'j', 'e', 'm', 'p', 'l', 'o']
>>> llista3[2]
'good'
>>> llista3[0]
['hola', 2, 3]

#Exemples d'accés a subllistes
>>> llista5[2:3]
[3]
>>> llista5[3:]
[4, 5, 6, 7, 8, 9]
>>> llista5[:3]
[1, 2, 3]
>>> llista5[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

#Afegir elements a una llista
>>> llista = [1,2,3,4,6]
>>> llista.append([7,8,9])
>>> print llista
[1, 2, 3, 4, 6, [7, 8, 9]]
>>> llista = [1,2,3,4,6]
>>> llista.extend([7,8,9])
>>> print llista
[1, 2, 3, 4, 6, 7, 8, 9]

#Buscar i eliminar elements
>>> llista2.index('hola')
0
>>> print llista2
['hola', 2, 3]
>>> llista2.remove('hola')
>>> print llista2
```

```
54 [2, 3]
55
56 #operadors sobrecarregats de concatenació
57 >>> print llista2+llista2
58 [2, 3, 2, 3]
59 >>> print llista2*4
60 [2, 3, 2, 3, 2, 3, 2, 3]
61 >>>
```

Noteu que els accessos a les llistes (i també als tuples) es fan mitjançant indexació directa. Per a llegir el primer element d'una llista s'usa la sintaxi **nom_llista[0]**. És important destacar que els índexs comencen a comptar des de 0, i arriben fins al nombre d'elements – 1. Així, en l'exemple, *lista3* té 3 elements, el primer dels quals (índex 0) és una altra llista, el segon (índex 1) és un 90 i el tercer (índex 2) és 'good'. Hi ha també la possibilitat de referenciar els elements d'una llista amb índexs negatius. En aquest cas es comença a comptar des del final de la llista, i l'índex –1 és l'últim element de la llista. Per exemple, `llista3[-1]` ens donaria la paraula *good*, `llista3[-2]` un 90, i `llista3[-3]` la llista equivalent a la `llista2`.

A part de poder consultar els elements individuals de les llistes, també es pot accedir a una subllista. Per a això s'usa l'operador :. A l'esquerra de l'operador es col·loca l'índex inicial i a la dreta l'índex final. Si no s'especifica cap índex significa que s'agafaran tots els elements fins a arribar a l'extrem corresponent. En l'exemple, veiem com `llista5[:3]` ens dóna tots els elements fins al segon (inclòs), i `llista5[3:]` comença pel tercer fins al final de la llista. En el cas de `llista5[:]` ens retornarà tota la llista.

Per a afegir elements a una llista, disposem de dos mètodes, **append** i **extend**. *Append* afegeix un element a una llista (independentment del seu tipus), i *extend* rep una llista d'elements, i els concatena a la llista actual. En l'exemple es pot observar la diferència de comportament de tots dos mètodes davant la mateixa entrada.

**Vegeu també**

Més endavant, en l'apartat6, dedicat a l'orientació a objectes, es descriurà amb més detall el concepte de mètode.

Altres mètodes auxiliars útils són la cerca d'elements i l'eliminació d'elements. La cerca se sol fer amb el mètode **index**, que retorna la posició on es troba la primera ocurrència de l'element. L'eliminació es fa amb el mètode **remove**, que elimina la primera ocurrència d'un determinat element en una llista. L'exemple mostra l'ús d'aquests dos mètodes, en què es busca un element determinat.

Una altra funció molt útil en les llistes, sobretot per a usar en bucles *for*, és **range**. Aquesta funció, donat un nombre natural *N*, ens retorna una llista amb tots els elements fins a *N* – 1. Així doncs, si teclegem *range(8)* en la línia d'ordres, obtindrem *[0, 1, 2, 3, 4, 5, 6, 7]*, que podran ser seguits en un bucle *for* per a executar una determinada acció 8 vegades.

Finalment, hi ha sobrecàrregues dels operadors aritmètics bàsics +,*. L'operador *suma* concatena dues llistes, i l'operador *multiplicació* rep una llista i un

valor escalar, i retorna la repetició de la llista (tantes vegades com indica el valor).

### 5.3. Diccionaris

Els diccionaris conformen un dels tipus de dades més útils per als programadors en Python, i representen una interessant novetat respecte als llenguatges de programació imperatius tradicionals. Es tracta d'unes estructures de memòria **associativa**. Els elements en els diccionaris no es consulten mitjançant un índex, sinó que s'hi accedeix directament per contingut. El codi 5.3 mostra un exemple de diccionari i com s'usa.

Codi 5.3: Exemple de tuples en Python

```
1  # Exemples d'ús de diccionaris
2
3  diccionari1 = {'nom': 'Alexandre', 'edat': 34, 'numerosfavorits': (3,7,13)}
4
5  tuple = (('nom', 'Alexandre'), ('edat', 34), ('numerosfavorits', (3,7,13)))
6
7  diccionari2 = dict(tuple);
8  diccionari3 = {'llibres':'el juego de Ender', 'telefon':934572345}
9
10 a = diccionari1['nom']
11 >>> print a
12 Alexandre
13
14 >>> 'nom' in diccionari2
15 True
16
17 diccionari1.update(diccionari3)
18 >>> print diccionari1
19 {'edat': 34, 'nom': 'Alexandre', 'numerosfavorits': (3, 7, 13), 'llibres': 'el juego de Ender',
20  'telefono': 934572345}
21
22 del diccionari1['nom']
23 >>> print diccionari1
24 {'edat': 34, 'numerosfavorits': (3, 7, 13), 'llibres': 'el juego de Ender', 'telefon': 934572345}
25
26 >>> 'nom' in diccionari1
27 False
```

Per a crear el diccionari simplement indicarem entre claus {,} les parelles d'elements que el conformen. En cada parella indicarem primer el valor 'clau' per a accedir a l'element, i després el valor que contindrà (que pot ser de qualsevol dels tipus disponibles en Python). La consulta d'un valor del diccionari es reduirà a preguntar pel valor que té la clau associada. En l'exemple, desem en la variable **a** el contingut de la posició de memòria *nom*. La cadena 'nom' ens serveix per a indexar posicions de memòria o variables. Les estructures de memòria associativa que permeten aquesta funcionalitat són de gran utilitat en el tractament de dades textuals i els sistemes de classificació de llenguatge natural.

Hi ha diversos mètodes auxiliars per a treballar amb diccionaris, entre els quals destaquen el mètode **in**, que permet consultar si una determinada clau està

present en el diccionari, el mètode **update**, que permet combinar dos diccionaris, i el mètode **del**, que elimina una entrada d'un diccionari.

### 5.4. Conjunts

Els conjunts són seqüències ordenades d'elements únics (concepte matemàtic de conjunt). Hi ha dos tipus de conjunts, els **sets** i els **frozensets** (que són mutables i immutables, respectivament). Les operacions típiques sobre conjunts són: conèixer la longitud d'un conjunt (mètode **len**), conèixer si un element es troba en un conjunt (mètode **in**), unió (mètode **union**), la intersecció (mètode **intersection**), afegir un element a un conjunt (mètode **add**) i eliminar un element d'un conjunt (mètode **remove**). El codi 5.4 mostra un exemple d'ús de conjunts en Python.

Codi 5.4: Exemple de conjunts en Python

```
1  # Exemples d'ús de conjunts
2
3  conjunt1 = set([1,2,3,4])
4  conjunt2 = set([3,4])
5  conjunt2.remove(3)
6  conjunt2.add(5)
7  print conjunt2
8
9
10 uni = conjunt1.union(conjunt2)
11 int = conjunt1.intersection(conjunt2)
12 >>> print uni
13 set([1, 2, 3, 4, 5])
14 >>> print int
15 set([4])
```

### 5.5. Fitxers

Els fitxers no són un tipus de dades en si mateixos, sinó un objecte que permet interactuar amb el sistema d'entrada i sortida per a escriure dades en el disc.

**Observació**

Hem preferit explicar la funcionalitat bàsica dels fitxers en aquests materials, ja que poden ser necessaris durant tot el curs per a carregar les dades necessàries per als algorismes de teoria.

En aquest manual, encara no hem tractat el concepte d'orientació a objectes en Python. No obstant això, serà necessari introduir alguns mètodes bàsics per a accedir a l'objecte que fa d'interfície amb el disc. La funció bàsica d'accés a un fitxer es denomina **open**, que rep com a paràmetre un nom de fitxer i un mode (lectura "r", escriptura "w"). L'accés a les dades es pot fer mitjançant els mètodes:

- **read**: llegeix un determinat nombre de bytes del fitxer (argument del mètode).
- **seek**: s'encarrega de posicionar el lector en el byte indicat en l'argument.
- **tell**: retorna la posició que s'està llegint actualment en el fitxer.
- **close**: tanca el fitxer.

Codi 5.5: Exemple d'accés a un fitxer. Executar el codi i verificar el funcionament de les funcions bàsiques d'accés a fitxer

```
# Exemples d'ús de fitxers

f=open("files.py","r")
f.tell()

f.read(30)

f.read(10)
f.tell()

f.seek(2)
f.tell()
f.read(8)
f.close()
```

### 5.5.1. Fitxers de text

Els fitxers de text són un cas particular que Python permet tractar de manera molt més còmoda i integrada en el llenguatge. Hi ha un mètode **readlines()**, que permet llegir el contingut del fitxer per línies, i retorna una matriu en què es pot indexar cada línia. De la mateixa manera, és possible accedir línia per línia al contingut d'un fitxer, i efectuar un tractament individualitzat de cadascuna mitjançant un simple bucle *for*. En l'exemple del codi 5.6 podeu veure el codi que recorre un fitxer i l'imprimeix per pantalla.

Codi 5.6: Exemple d'accés a un fitxer

```
# Exemples d'ús de fitxers

#Llegir una linia
f = open("Textfiles.py", "r")
linies = f.readlines()

#imprimir per pantalla la primera línia
print linies[0]
#imprimir-les totes
print linies[:]

#Tancar el fitxer
f.close();

#Llegir tot el fitxer línia per línia
for line in open("Textfiles.py", "r"):
    print line

#Bucle que recorre les línies d'un fitxer origen i les
#copia en el de destinació
fwrite = open("CopiaTextfiles.py", "w")
for line in open("Textfiles.py", "r"):
    fwrite.write(line);

fwrite.close()
```

Finalment, disposem del mètode **write**, que permet escriure continguts en un fitxer. En l'exemple anterior, recorrem el primer fitxer i línia per línia l'anem escrivint en el fitxer destinació ("CopiaTextfiles").

# 6. Python i l'orientació a objectes

El llenguatge Python permet treballar mitjançant el paradigma de programació imperativa clàssica (com en tots els exemples vistos fins ara), o mitjançant el paradigma de l'orientació a objectes, i fins i tot mitjançant llenguatge funcional. En aquest apartat veurem la sintaxi de l'orientació a objectes (OO) en Python, amb alguns exemples pràctics. Assumirem que l'estudiant coneix les bases de l'OO, i que està familiaritzat amb els conceptes de classe, objecte i mètode.

## 6.1. Els objectes en Python

Un objecte no és més que el resultat d'encapsular una determinada entitat, que està formada per un estat (les dades o **atributs**) i un funcionament (els **mètodes**). Una classe és la plantilla o concepte genèric d'un objecte, que s'usa per a definir-ne les propietats i serveis. Un objecte és, doncs, una instància concreta d'una classe.

Per a definir una classe usem la paraula clau **class**. En el fragment de codi 6.1 mostrem un exemple de definició d'una classe.

Codi 6.1: Exemple de definició d'una classe

```
1  # Definició de la classe Forn
2  class Forn:
3        def __init__(self, pans, pastetes):
4            self.pans = pans
5            self.pastes = pastetes
6            print "A la botiga hay", self.pans, "pans i",pastes, "pastes encara que aquestes no es venen"
7        def vendre(self):
8            if self.pans > 0:
9                  print "Un pa venut!"
10                 self.pans -= 1
11           else:
12                 print "Ho sentim, no queden pans per vendre"
13       def coure(self, peces):
14           self.pans += peces
15           print "Queden", self.pans, "pans"
16
17 forn1 = Forn(3,4)
18 forn2 = Forn(1,2)
19
20 forn2.vendre()
21 forn2.vendre()
22
23 forn2.coure(1)
24 forn2.vendre()
```

El primer que observem, després dels comentaris, és l'ús de la paraula clau **class**, que ens permet definir l'entorn d'una classe. A continuació ve el nom de la classe, *Fleca*, que s'utilitzarà posteriorment per a construir objectes de la classe. Per a definir els mètodes de la classe, fem ús de nou de la paraula clau def. Els mètodes es defineixen seguint una sintaxi bastant semblant a les funcions.

**Vegeu també**

La paraula clau **def** s'estudia en l'apartat 4, dedicat a les funcions.

Un altre punt destacable és el mètode **__init__**. Aquest mètode és especial, sempre rep el mateix nom (independentment de la classe), i serveix per a especificar les accions que cal dur a terme en el moment de la creació i inicialització d'un objecte de la classe en qüestió. En aquest cas, rep tres paràmetres, encara que el primer és, de nou, especial; es tracta de la paraula reservada **self**. **self** s'utilitza per a fer referència a l'objecte mateix. Una vegada creat, **self** ens permetrà diferenciar els noms de les variables membres de l'objecte de la resta de valors. La funció de **self** és semblant a **this** en altres llenguatges de programació (C++, Java,...). Si continuem mirant l'exemple, observem com tenim una variable membre anomenada *pans*, i un argument a la funció **__init__** amb el mateix nom. En l'expressió (**self.pans=pans**) diferenciem tots dos valors mitjançant l'ús de **self**.

### 6.1.1. Creació d'instàncies (objectes) d'una classe

Per a crear un objecte concret d'una classe, n'hi ha prou d'usar el nom que hem donat a la classe seguit dels arguments que necessita la funció d'inicialització (que aquí té un efecte similar als constructors del llenguatge C++). En l'exemple hem creat dos objectes de la classe *Fleca* (*fleca1* i *fleca2)*.

L'execució de mètodes, com en molts llenguatges de programació, segueix la sintaxi **nom_objecte.nom_metode(arguments,...)**.

En l'exemple es crida sovint als mètodes *vendre* i *coure* dels objectes creats. Executeu el fragment de codi i interpreteu el resultat que s'imprimeix per pantalla.

## 6.2. Herència

Quan diem que una classe hereta d'una altra, ens referim al fet que la classe resultant contindrà tots els atributs i mètodes de la classe *pare* o *superclasse*, i es permet d'aquesta manera una especialització progressiva de les classes i més reutilització de codi.

Per a fer que una classe hereti d'una altra en Python, simplement ho hem d'indicar a continuació del nom de la classe, entre parèntesis. En el codi 6.2 es mostra un exemple molt simple d'herència.

Codi 6.2: Exemple d'ús d'herència en les classes Python

```
# Definició de la classe Animal: exemples d'herència
class Animal:
        def __init__(self, age, weight):
                self.age = age
                self.__weight = weight
        def __privateMethod(self):
                print self.weight;
        def getWeight(self):
                return self.__weight;
        def eat(self, kgm):
                self.__weight += kgm;
                print "The animal Weitghs:", self.__weight,"after eating."

class Bird(Animal):
        def fly(self):
                print "I fly as a bird!"

class Mammal(Animal):
        def fly(self):
                print "I can not fly , I am a mammal!"

class Ostrich(Animal,Bird):    #Estruç
        def fly(self):
                print "I can not fly , I am a Bird but ostrichs do not fly!"

class Platypus1(Mammal, Bird):
        pass

class Platypus2(Bird ,Mammal):
        pass


animal1 = Animal(3,0.5)
animal1.eat(0.1)

canary = Bird(1,0.45)
canary.eat(0.02)
canary.fly()

bear = Mammal(10,150)
bear.eat(10)
bear.fly()

ostrich = Ostrich(5,30)
ostrich.fly()

platypus = Platypus1(2,3)
platypus.fly()

platypus = Platypus2(2,3)
platypus.fly()

print bear.getWeight()
bear.privateMethod()
```

Observeu que en aquest exemple treballem amb quatre classes diferents: *Animal*, *Bird*, *Mammal* i *Ostrich*. *Animal* és superclasse de la resta, que hereten d'ella. En el cas d'*Ostrich*, noteu que hereta de dues classes alhora (encara que no sigui estrictament necessari). El Python permet herència múltiple. Executeu el codi, i n'entendreu el funcionament. Observeu que el mètode *fly()* té un comportament diferent en funció del tipus concret de l'objecte.

Molts llenguatges no permeten herència múltiple, atès que això pot originar conflictes quan s'hereten mètodes o atributs amb el mateix nom de dues superclasses.

En cas de conflicte, Python dóna preferència a la classe situada més a l'esquerra en el moment de la definició.

Torneu a executar l'exemple, i observeu el resultat de les crides a *fly* de les dues classes *Platypus1* i *Platypus2*. Aquest exemple sintètic il·lustra l'ordre de preferències en funció de l'ordre de definició de l'herència en cas de conflictes de nom.

## 6.3. Encapsulació

Un dels principals avantatges de la programació orientada a objectes és l'encapsulació. Aquesta propietat permet construir objectes amb mètodes i atributs que no es poden cridar externament. Es tracta de codi intern que el programador ha preparat i que no vol que es vegi alterat. L'objecte ofereix una sèrie de serveis a l'exterior, i oculta part de la seva codificació interna. En Python no tenim paraules clau específiques per a denominar l'encapsulació. Tots els mètodes en Python són públics, excepte aquells que comencen per un doble guió baix (__). En l'exemple anterior, la crida a *bear.privateMethod()* produeix una excepció en temps d'execució.

**Encapsulació en altres llenguatges**

En altres llenguatges sí que s'utilitzen paraules clau per a denominar l'encapsulació. Per exemple, en C++ o Java, que usen *public, private, protected*...

L'encapsulació té molta utilitat si es volen amagar els detalls d'implementació d'una classe determinada. Suposem, per exemple, que internament el pes d'una classe *Animal* es desa en altres unitats (sistema anglès) en lloc de quilograms. L'ús de funcions *setters* i *getters* públiques permetria que la interfície amb l'usuari fos sempre la mateixa, independentment d'aquesta codificació interna, que seria privada. S'ha exemplificat aquest fet en el mètode **getWeight()** de la classe *Mammal*.

**Vegeu també**

El concepte de *getter* i *setter* s'ha tractat amb profunditat en l'assignatura *Programació orientada a objectes*.

Com a curiositat, observeu que la funció **__init__** comença també amb dos guions baixos. Els mètodes Python que comencen i acaben amb guions baixos són especials, i no tenen a veure amb el concepte de mètode privat. Una altra funció especial és el destructor o **__del__**, que es crida quan l'objecte es deixa d'utilitzar per a eliminar-lo.

## 6.4. Polimorfisme

En el cas de Python, com en la majoria de llenguatges de programació, es basa en l'ús d'herència.

La paraula *polimorfisme** en programació denomina la propietat que tenen molts llenguatges d'executar codi diferent en funció de l'objecte que fa la crida.

* Del grec *diverses formes*.

Així, podríem referenciar objectes mitjançant el tipus *superclasse*, però en el moment d'executar-ne els mètodes es cridaran realment els mètodes de la classe derivada. El concepte de polimorfisme es troba molt lligat a l'enllaç dinàmic. El Python per defecte ja usa l'enllaç dinàmic, amb la qual cosa no requereix cap notació especial per a usar polimorfisme.

**Enllaç dinàmic**

L'enllaç dinàmic és la decisió de quin codi s'executa en temps d'execució en lloc de fer-ho en temps de compilació.

# 7. El Python com a llenguatge funcional

La programació funcional és un paradigma de programació basat en el concepte matemàtic de funció, no en un sentit procedimental, com hem vist fins ara, sinó més aviat en l'ús de funcions d'ordre superior. Aquest concepte fa referència a l'ús de les funcions com si fossin valors propis del llenguatge.

> És a dir, en la programació funcional podem desar una funció en una variable, i posteriorment aplicar-la sobre uns arguments, i es permet fins i tot que una funció retorni una altra funció com a sortida.

Les característiques funcionals de Python no seran les més utilitzades en aquest curs, però en aquest apartat farem un breu resum amb algun exemple orientatiu, ja que hi ha alguns iteradors bastant utilitzats que es basen en el paradigma funcional.

El codi 7.1 és un exemple en el qual es mostra l'ús bàsic de la programació funcional, l'accés a les funcions com a variables.

Codi 7.1: Exemple simple d'aplicació de Python funcional

```
1  # Exemple d'ús de llenguatge funcional
2
3  def money(country):
4      def spain():
5          print "Euro"
6      def japan():
7          print "Yen"
8      def eeuu():
9          print "dollar"
10     functor_money ={"es":spain,
11                     "jp":japan,
12                     "us":eeuu}
13     return functor_money[country]
14
15 f = money("us")
16 money("us")()
17 f()
18
19 f = money("jp")
20 f()
```

Introduïu l'exemple i executeu-lo. Com podreu observar, la línia de codi *f = money("us")* no fa cap acció visible. Aquesta línia s'encarrega de crear una nova variable, que serà una funció. Aquesta funció es genera o selecciona en funció de l'entrada de l'usuari, en aquest cas una cadena de caràcters que activa el

selector en el diccionari intern a la funció *money*. El resultat desat en *f* és una funció, i per tant pot ser cridat per a executar-ho. Com podeu observar, les crides *money("us")()* i *f()* són equivalents. *A posteriori* hem canviat el valor on "apunta" la funció *f*, i n'hem modificat en conseqüència la codificació (i per tant, es genera una sortida diferent).

El paradigma funcional té múltiples avantatges i aplicacions. En aquest manual ens centrarem a descriure tres iteradors que s'han usat molt en conjunció amb les llistes: **map**, **filter** i **reduce**. L'estudiant familiaritzat amb els llenguatges LISP o ML veurà un cert paral·lelisme amb aquests iteradors. El codi 7.2 en mostra l'exemple d'ús.

Codi 7.2: Exemple d'aplicació d'iteradors

```
1  # Exemple d'ús d'iteradors
2
3  def double(num):
4      return num*2
5  def even(num):
6      return (num%2) == 0
7  def operation(num1,num2):
8      return num1*num2+1;
9
10 l1 = range(10);
11
12 l2 = map(double,l1);
13 l3 = filter(even,l1);
14 l4 = reduce(operation,l1);
15
16 #Equivalent però usant lambda function
17 l5 = map(lambda num: num*2,l1)
18 l6 = filter(lambda num: num%2==0,l1)
19 l7 = reduce(lambda num1,num2: num1*num2 +1,l1)
20
21 #Comprensió de llistes
22 l8 = [num*2 for num in l1];
23
24 print l1
25 print l2,l5
26 print l3,l6
27 print l4,l7
28
29 print l8
```

Observeu que la sintaxi de **map**, **filter** i **reduce** és similar. Sempre reben un primer argument, que és el nom de la funció que executaran sobre els elements de la llista (passada com a segon argument). En el cas de **map**, l'iterador aplica la funció a cada element de la llista, i retorna una nova llista amb el resultat d'aplicar aquesta funció a cada element. Per la seva banda, **filter**, retorna una llista amb aquells elements de la llista original que passen l'avaluació de la funció. En aquest cas la funció que rep **filter** retorna un valor booleà (*true, false*) que fa les funcions de selecció. Finalment **reduce** aplica recursivament la funció a cada parell d'elements de la llista, fins a deixar un sol resultat.

### 7.1. Funcions lambda

Les funcions lambda són funcions anònimes definides en línia, que no seran referenciades posteriorment. Es construeixen mitjançant l'operador **lambda**, sense usar els parèntesis per a indicar els arguments. Aquests van directament després del nom de la funció, que finalitza la declaració amb dos punts (**:**). Just després, en la mateixa línia s'escriu el codi de la funció. En l'exemple anterior s'ha inclòs la versió de crida a **map**, **filter** i **reduce** usant funcions lambda. Observeu que aquestes funcions estan limitades a una sola expressió.

### 7.2. Comprensió de llistes

Una sintaxi alternativa als iteradors anteriors que s'està imposant en les últimes versions de Python és la comprensió de llistes. En aquest cas, es pretén crear una llista a partir d'una altra llista. La sintaxi és de nou molt senzilla: s'especifica entre claudàtors (o parèntesis) l'expressió per aplicar, seguida de la paraula clau **for**, la variable per iterar, la paraula clau **in** i la llista origen. Com es pot observar en l'exemple anterior, hem reproduït la funció que duplicava cada element de la llista aplicant comprensió, i l'expressió es llegiria: "per cada num de l1 fes num*2".

***Pattern matching***

A diferència d'altres llenguatges purament funcionals, el Python no implementa *pattern matching*, encara que és possible simular-lo. Aquests conceptes queden ja fora de l'abast d'aquest manual.

# 8. Biblioteques: *NumPy*, *PyLab* i *SciPy*

El codi Python es pot agrupar en mòduls i paquets per millorar-ne l'organització i poder reutilitzar i compartir tot el que s'ha programat. Cada fitxer equival a un mòdul. Per a poder utilitzar codi d'aquests fitxers s'utilitza la paraula clau **import**, seguida del nom del mòdul, de la mateixa manera que en C usàvem **include**. Cal tenir en compte que **import** carrega literalment tot el contingut del fitxer, i executa també tota la part executable dins del mòdul. És habitual definir mòduls en què només es defineixin les funcions i classes que es volen publicar per evitar execucions involuntàries. També és possible importar només un objecte concret que ens interessi; això es fa mitjançant la construcció: **from** mòdul **import** nom de l'objecte. Les crides a objectes solen anar precedides amb el nom del mòdul on es troben, per preservar l'espai de noms i mantenir objectes amb el mateix nom en diferents mòduls.

El Python té associada una variable d'entorn (*PYTHONPATH*) en la qual s'indica on es troba (la carpeta concreta) la major part de les biblioteques integrades en el llenguatge. En cas de crear nous mòduls en una altra ubicació diferent del codi actual, es pot afegir aquesta ruta a la variable d'entorn *PYTHONPATH*.

Els paquets no són més que entitats que organitzen els mòduls. Els paquets tenen la seva equivalència amb el sistema de carpetes o directoris, en què podem desar de manera estructurada diferents fitxers (mòduls). La paraula clau **import** s'utilitza també per a importar mòduls dels paquets, i s'usa la mateixa nomenclatura quant a espai de noms. Així doncs, per a importar un mòdul anomenat *modulExemple*, del paquet *PaquetsExemple*, seguiríem la sintaxi: **import** *PaquetsExemple.modulExemple*. Suposant que en aquest mòdul tinguéssim de nou definida la funció *double*, la cridaríem de la manera següent: *PaquetsExemple.modulExemple.double(3)*, acció que ens retornaria un 6. Aquesta notació es pot simplificar notablement amb la paraula clau **as**. Seguint l'exemple, si importéssim el codi com: **import** *PaquetsExemple.modulExemple* **as** *e*, podríem cridar posteriorment la funció *double* com *e.double(3)*.

En aquest curs treballarem amb diversos paquets de Python, però tres seran imprescindibles, ja que que contenen les principals biblioteques de *machine learning* ja implementades en Python. Es tracta de *NumPy*, *PyLab* i *SciPy*.

### 8.1. NumPy

La biblioteca *NumPy* ens permet treballar amb dades científiques, i equipara en certa manera el potencial de Python al d'altres llenguatges com Matlab o Octave. *NumPy* es troba disponible de manera gratuïta a Internet*. S'hi pot baixar i consultar la documentació de les diferents propietats que ens ofereix *NumPy*, que se centren bàsicament en el tractament de matrius i *arrays* de dimensió *N*, i en un conjunt de funcionalitats d'àlgebra lineal i tractament del senyal aplicat a l'anàlisi científica.

* http://numpy.scipy.org/

### 8.2. PyLab

*PyLab* és una biblioteca que intenta aportar funcionalitats extra a *NumPy* integrant gran part de les funcions Matlab que s'han usat històricament en entorns de *machine learning*. A Internet* es pot trobar el paquet a punt per a baixar, amb la documentació corresponent.

* http://www.scipy.org/pylab

### 8.3. SciPy

*SciPy* és una expansió de *NumPy*, que integra nous paquets per al tractament científic de dades. Integra gran quantitat de funcions de processament d'imatges, processament del senyal, estadística, integració numèrica. A Internet* es pot trobar l'última versió de la plataforma i la documentació associada.

* http://www.scipy.org/

## Resum

En aquest mòdul hem vist els elements bàsics de la programació en Python. El mòdul està pensat perquè un estudiant amb coneixements de programació es pugui introduir ràpidament en Python, i pugui fer els seus primers programes en poc temps. L'objectiu primordial del mòdul i de l'assignatura és el seguiment correcte dels conceptes exposats en teoria. No es pretén formar experts programadors en Python, sinó més aviat capacitar l'estudiant per a poder entendre l'abundant codi d'exemple que incorporen els materials.

És aconsellable que feu els exercicis per a acabar d'assentar els coneixements bàsics. No és necessari fer-los tots, sinó simplement intentar fer els que a primera vista semblin més complexos. Juntament amb el material, podreu trobar a l'aula les solucions als exercicis, per a poder fer les consultes i comparacions oportunes.

## Exercicis d'autoavaluació

**1.** Escriviu una funció en Python que, donada una llista de nombres, retorni una altra llista en ordre invers. Per a fer aquest exercici s'haurà d'utilitzar un bucle o estructura repetitiva. No es permet l'ús de funcions membres de la classe *list* (especialment *list.reverse()*).

**2.** Escriviu una funció que, donat un nombre enter *N*, retorni una llista amb tots els nombres primers fins a *N*. Per a solucionar l'exercici heu de crear una funció auxiliar que indiqui si un nombre determinat és primer (i retorni un valor booleà).

**3.** Escriviu una funció que rebi un tuple compost per caràcters, i retorni una llista amb els caràcters en majúscules. Heu de recórrer el tuple caràcter per caràcter per a fer la conversió. Per a convertir un caràcter a majúscula podeu usar el mètode *upper()*. Per exemple *'a'.upper()* ens retorna 'A'.

**4.** Convertiu el text 'exemple' en una llista que contingui els seus 7 caràcters. Després convertiu-lo en un tuple i usant la funció de l'exercici anterior obtingueu una llista amb el text en majúscules.

**5.** Escriviu una funció que, donada una llista de nombres, retorni una llista amb només els elements en posició parell.

**6.** Amplieu la funció anterior perquè, donada una llista i uns índexs, ens retorni la llista resultat d'agafar només els elements indicats pels índexs. Per exemple, si tenim la llista [1,2,3,4,5,6] i els índexs [0,1,3], hauria de retornar la llista [1,2,4].

**7.** Escriviu una funció que ens retorni quantes vegades apareix cadascuna de les paraules d'un text (freqüència d'aparició de les paraules). Per a això podeu usar un diccionari en què la clau sigui cadascuna de les paraules del text i el contingut desi el nombre d'aparicions de la paraula. Per simplificar l'exercici, podeu usar el mètode *split(' ')*, que, donat un separador (l'espai), ens retorna una llista amb totes les paraules d'un text de manera separada. Per exemple: *'hola, això és un exemple'.split(' ')* ens retornaria: *['hola', 'això', 'és', 'un', 'exemple']*

**8.** Escriviu una funció que retorni un conjunt format pels nombres compostos (no primers) més petits que un *N* donat.

**9.** Codifiqueu una funció que escrigui en un fitxer de text els nombres primers que van des de l'1 fins al 999.999.

**10.** Escriviu una funció que llegeixi el contingut d'un fitxer de text i ens doni la freqüència d'aparició de cada paraula. Podeu usar el codi de l'exercici 7, en el qual s'usaven diccionaris per a comptar les aparicions de cada paraula.

**11.** Implementeu un programa que tingui dues classes, *Camió* i *Cotxe*, totes dues subclasses de la superclasse *Vehicle*. Trieu tres atributs comuns a *Cotxe* i *Camió* i dos atributs específics de cada classe. Penseu bé on heu de col·locar cada atribut. Escriviu un mínim de dos mètodes en cada classe i executeu-los en el programa principal.

**12.** Escriviu una versió de l'exercici 2 que utilitzi programació funcional. Podeu usar l'iterador *filter* per a mantenir només aquells valors de la llista que siguin primers.

**13.** Escriviu una funció que depenent d'un selector executi algun dels primers 5 exercicis d'aquest apartat. La funció rebrà un caràcter *('1','2',...,'5')* i haurà de retornar una funció que testegi l'apartat corresponent. Per exemple, si escrivim *f=selector('4')*, *f* haurà de ser una funció que en executar-se finalment ens retorni la paraula *exemple* en majúscules.

# Bibliografia

**González Duque, Raúl**. *Python para todos*. CC. (versió en línia)

**Lutz, Mark** (2011). *Programming Python* (4a. ed.). O'Reilly Media.

**Lutz, Mark** (2007). *Learning Python* (3a. ed.). O'Reilly Media.

**Martelli, Alex** (2006). *Python in a Nutshell* (2a. ed.). O'Reilly Media.

**Pilgrim, Mark** (2004). *Dive Into Python*. APress (versió en línia).