Introducció a la programació

Versió preliminar

Jeffrey Elkner Allen B. Downey Chris Meyers
Antoni Soto-Riera Marc Vigo-Anglada Sebastià Vila-Marta
Jordi Vives

1 de desembre de 2011

Aquest llibre és una obra derivada de l'original *"How to Think Like a Computer Scientist. Learning with Python 2nd Edition"* de Jeffrey Elkner, Allen B. Downey i Chris Meyers. L'original pot ser consultat seguint l'enllaç `http://openbookproject.net/thinkcspy`.

La composició d'aquest llibre s'ha realitzat amb LaTeX i els fonts corresponents es poden trobar seguint aquest enllaç `http://devel.cpl.upc.edu/infodocs`

Copyright del text original © Jeffrey Elkner, Allen B. Downey and Chris Meyers.
Copyright de la traducció i l'obra derivada © Antoni Soto-Riera, Marc Vigo-Anglada, Sebastià Vila-Marta i Jordi Vives.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with Invariant Sections being Foreward, Preface, and Contributor List, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Sumari

Pròleg vii

Prefaci ix
1 Com i per què vaig canviar a `Python` ix
2 Com vaig trobar un llibre de referència x
3 `Python` com a introducció a la programació x
4 Com s'ha construït la comunitat xii

Llista de contribuïdors xiii
5 Segona Edició xiii
6 Primera Edició xiv

1 El mètode de programar 1
1.1 El llenguatge de programació `Python` 1
1.2 Què és un programa? 3
1.3 Què és depurar? 4
1.4 Errors sintàctics 4
1.5 Errors en temps d'execució 4
1.6 Errors semàntics 4
1.7 Depuració experimental 5
1.8 Llenguatges formals i naturals 5
1.9 El primer programa 6
Exercicis 7

2 Variables, expressions i sentències 9
2.1 Valors i tipus 9
2.2 Variables 10
2.3 Noms de variable i paraules clau 11
2.4 Sentències 12
2.5 Avaluació d'expressions 12
2.6 Operadors i operands 13
2.7 Ordre de les operacions 14
2.8 Operacions amb cadenes 14
2.9 Entrada 15
2.10 Composició 15
2.11 Comentaris 16
Exercicis 16

3 Funcions 19
3.1 Definicions i ús 19
3.2 Flux d'execució 21
3.3 Paràmetres, arguments i la sentència import 21
3.4 Composició 23
3.5 Les variables i paràmetres són locals 23
3.6 Diagrames de pila 24

Exercicis . . . 25

4 Condicionals **29**
4.1 L'operador mòdul . . . 29
4.2 Valors booleans i expressions . . . 29
4.3 Operadors lògics o booleans . . . 30
4.4 Execució condicional . . . 30
4.5 Execució alternativa . . . 31
4.6 Condicionals encadenats . . . 32
4.7 Condicionals niuats . . . 32
4.8 La sentència "return" . . . 33
4.9 Entrades de teclat . . . 33
4.10 Conversió de tipus . . . 34
Exercicis . . . 36

5 Funcions fructíferes **39**
5.1 Valors de retorn . . . 39
5.2 Desenvolupament de programes . . . 40
5.3 Composició . . . 42
5.4 Funcions booleanes . . . 43
5.5 El tipus "function" . . . 44
5.6 Programar amb estil . . . 44
5.7 Triples cometes . . . 45
5.8 Prova de components amb "doctest" . . . 45
Exercicis . . . 47

6 Iteració **53**
6.1 Assignacions repetides . . . 53
6.2 Actualització de variables . . . 53
6.3 La sentència "while" . . . 54
6.4 La traça d'un programa . . . 55
6.5 Comptatge de dígits . . . 56
6.6 Assignació abreviada . . . 57
6.7 Taules . . . 57
6.8 Taules de dues dimensions . . . 59
6.9 Encapsulació i generalització . . . 59
6.10 Més encapsulació . . . 60
6.11 Variables locals . . . 60
6.12 Més generalització . . . 61
6.13 Funcions . . . 63
6.14 El mètode de Newton . . . 63
6.15 Algorismes . . . 63
Exercicis . . . 64

7 Cadenes de caràcters **71**
7.1 Un tipus de dades compost . . . 71
7.2 Longitud . . . 71
7.3 Recorreguts i el bucle "for" . . . 72
7.4 Segments de cadenes . . . 73
7.5 Comparació de cadenes . . . 74
7.6 Les cadenes són immutables . . . 74
7.7 Els operadors "in" i "not in" . . . 75
7.8 Una funció "find" . . . 75

7.9 Bucles i comptatge 76
7.10 Paràmetres opcionals 76
7.11 Operacions sobre cadenes 77
7.12 Classificació de caràcters 78
7.13 El mètode per formatar cadenes 79
Exercicis 81

8 Llistes 87
8.1 Creació de llistes 87
8.2 Accés als elements 88
8.3 Longitud d'una llista 89
8.4 Pertinença a una llista 89
8.5 Operacions amb llistes 89
8.6 Segments de llista 90
8.7 La funció "range" 90
8.8 Les llistes són mutables 91
8.9 Esborrat en llistes 92
8.10 Objectes i valors 92
8.11 Àlies 93
8.12 Clonat de llistes 94
8.13 Llistes i l'iterador "for" 94
8.14 Les llistes com a paràmetres 96
8.15 Funcions pures i accions 96
8.16 Llistes niuades 97
8.17 Matrius 98
8.18 Mètodes del tipus llista 98
8.19 Desenvolupament guiat per tests 99
8.20 Cadenes i llistes 102
Exercicis 103

9 Mòduls i fitxers 107
9.1 Mòduls 107
9.2 Alguns mòduls estàndard 107
9.2.1 El mòdul "random" 107
9.2.2 El mòdul "time" 108
9.2.3 El mòdul "math" 109
9.3 Creació de mòduls 109
9.4 Àmbits 110
9.5 Atributs i l'operador punt 111
9.6 Llegir i escriure fitxers de text 111
9.7 Fitxers de text 113
9.8 Directoris 114
9.9 Comptatge de lletres 115
9.10 El mòdul "sys" i l'atribut "argv" 116
Exercicis 117

10 Tuples i mutabilitat 119
10.1 Tuples i mutabilitat 119
10.2 Assignació de tuples 120
10.3 Tuples com a valors de retorn 120
10.4 De nou les accions i funcions pures 121
10.5 Excepcions 122
10.6 Llistes definides per intensió 124

Exercicis 125

11 Diccionaris **127**
11.1 Operacions dels diccionaris 128
11.2 Mètodes dels diccionaris 128
11.3 Àlies i còpies 129
11.4 Matrius disperses 129
11.5 Enters grans 130
11.6 Comptatge de lletres 131
Exercicis 131

GNU Free Documentation License **137**
0 PREAMBLE 137
1 APPLICABILITY AND DEFINITIONS 137
2 VERBATIM COPYING 138
3 COPYING IN QUANTITY 139
4 MODIFICATIONS 139
5 COMBINING DOCUMENTS 141
6 COLLECTIONS OF DOCUMENTS 141
7 AGGREGATION WITH INDEPENDENT WORKS 141
8 TRANSLATION 141
9 TERMINATION 142
10 FUTURE REVISIONS OF THIS LICENSE 142
11 RELICENSING 142

Pròleg

Per David Beazley

Com a educador, investigador i autor, estic complagut de poder veure la consecució d'aquest llibre. `Python` és un llenguatge de programació divertit i extremadament fàcil d'usar que ha guanyat en popularitat a un ritme constant els últims anys. Desenvolupat ara fa uns deu anys per Guido van Rossum, la sintaxi simple de `Python` així com la filosofia general provenen majoritàriament d'ABC, un llenguatge orientat a l'ensenyament desenvolupat als anys 80. No obstant això, `Python` també va ser creat per solucionar problemes reals i pren una amplia varietat de característiques de llenguatges de programació tals com C++, Java, Modula-3 i Schema. Degut a això, una de les característiques més remarcables de `Python` és la seva gran capacitat d'atreure desenvolupadors de programari professional, científics, investigadors, artistes i educadors.

Malgrat aquesta atracció per `Python` per part de moltes comunitats, pot ser que encara et preguntis: Per què `Python`? O, per què ensenyar programació amb `Python`? Contestar aquestes preguntes no és una tasca senzilla (especialment quan l'opinió popular pertany al bàndol de les alternatives més masoquistes com C++ i Java). No obstant, crec que la resposta més directa és que programar en `Python` és simplement molt més divertit i productiu.

Quan imparteixo cursos d'enginyeria informàtica, m'interessa tractar una sèrie de conceptes importants i també que el material sigui interessant i que enganxi als estudiants. Malauradament, existeix la tendència en els cursos introductoris de programació a centrar l'atenció en l'abstracció matemàtica i a fer que els estudiants esdevinguin frustrats pels molestos problemes relacionats amb detalls de baix nivell sobre sintaxis, compilació i el fet de complir una sèrie de normes aparentment rebuscades. Tot i que aquesta abstracció i formalisme són importants per als enginyers de software i per als estudiants que tinguin intenció de continuar els seus estudis d'enginyeria informàtica, fer aquest tipus d'aproximació en un curs introductori, en gran part, només aconsegueix convertir en avorrida l'enginyeria informàtica. Quan imparteixo un curs, no vull tenir una classe amb alumnes insulsos. M'agradaria molt més veure'ls intentar resoldre problemes interessants mitjançant l'exploració de diverses idees, provant aproximacions no convencionals, trencant les normes i aprenent dels errors. Volent fer això, no m'interessa malbaratar la meitat del semestre intentant resoldre obscurs problemes sintàctics, errors de compilació inintel·ligibles o els centenars de formes en que un programa pot generar una fallida de protecció general.

Una de les raons per les que a mi m'agrada `Python` és que proporciona un molt bon equilibri entre el que és pràctic i el que és conceptual. Degut a què `Python` és interpretat, els principiants poden agafar el llenguatge i començar a desenvolupar coses senzilles immediatament sense haver de perdre temps en compilacions i enllaçaments. A més, `Python` ve amb una àmplia varietat de llibreries i mòduls que es poden usar per fer tot tips de tasques que van des de la programació web fins als gràfics. Tenir aquest tipus d'orientació pràctica és una bona manera d'atraure els estudiants i els hi permet completar projectes significatius. No obstant, `Python` també pot servir com a base per introduir conceptes importants d'enginyeria informàtica. Donat que `Python` suporta completament subprogrames i classes, els estudiants poden ser introduïts gradualment a temes com ara l'abstracció de procediments, les estructures de dades i la programació orientada a objectes (totes les quals són aplicables a cursos posteriors de Java o C++). `Python` fins i tot pren una sèrie de característiques dels llenguatges de programació funcionals i pot ser usat per introduir conceptes que podrien ser tractats amb més detall a cursos de Scheme o Lisp.

Llegint el prefaci d'en Jeffrey, em criden l'atenció els seus comentaris en què diu que `Python` li va permetre veure un nivell més alt d'èxit i un nivell més baix de frustració, així com li va permetre avançar més ràpid amb uns millors resultats. Tot i que aquests comentaris fan referència al seu curs introductori, a vegades faig servir `Python` exactament per les mateixes raons als cursos de nivell de graduat avançat als cursos d'enginyeria informàtica de la Universitat de Chicago. En aquests cursos, em trobo constantment enfrontat a la tasca desalentadora d'haver de tractar un munt de material difícil del curs en un devastador trimestre de nou setmanes. Tot i que certament em seria possible infligir una gran quantitat de dolor i sofriment usant un llenguatge com C++, sovint he trobat aquesta aproximació contraproduent (especialment quan el curs tracta d'un tema no relacionat només amb la programació). He descobert que usar `Python` em permet centrar-me millor en el propi tema que pertoqui mentre els alumnes són capaços de completar projectes de classe importants.

Tot i que `Python` encara és un llenguatge jove i en evolució, crec que té un futur brillant a l'educació. Aquest llibre és un pas important en aquesta direcció.

David Beazley
Universitat de Chicago
Autor de "the Python Essential Reference"

Prefaci

Per Jeffrey Elkner

Aquest llibre existeix gràcies a la possibilitat de col·laboració que proporcionen Internet i la comunitat de programari lliure. Els tres autors del llibre —un professor d'escola, un professor d'institut i un programador professional— hem pogut col·laborar estretament tot i no haver-nos trobat mai cara a cara per treballar-hi, ajudats per molt altra gent que han dedicat temps i energia a enviar-nos els seus comentaris.

Pensem que aquest llibre és una referència dels beneficis i de les possibilitats futures d'aquest tipus de col·laboracions, el marc de les quals ha estat fixat per en Richard Stallman i la Free Software Foundation.

1 Com i per què vaig canviar a `Python`

A l'any 1999, per primera vegada l'examen d'informàtica del programa preuniversitari Advanced Placement (AP) del College Board es va fer en C++. Com en molts altres instituts del país, la decisió de canviar de llenguatge va tenir un impacte directe en el pla d'estudis d'informàtica del Yorktown High School d'Arlington, a Virgínia, que és l'institut en el qual jo ensenyo. Fins a aquesta data, el llenguatge de referència havia estat Pascal, tant en els nostres primers cursos com en els del programa AP. Conservant la pràctica habitual d'oferir als estudiants dos anys d'estudi amb el mateix llenguatge, vam decidir de canviar a C++ al primer any del curs 1997-98 per tal que el pròxim curs anéssim al mateix ritme que els cursos del programa AP del College Board.

Al cap de dos anys, ja estava convençut que C++ era una mala opció per introduir la informàtica als estudiants. Tot i que segurament C++ és un llenguatge de programació molt potent, és també un llenguatge extremadament difícil d'aprendre i ensenyar. Constantment havia de lluitar contra la complicació de la seva sintaxi i les múltiples maneres diferents de fer les coses que té i, com a conseqüència, perdia molts estudiants de manera innecessària. Convençut que es podia triar un llenguatge millor pels estudiants de primer curs, vaig començar a cercar una alternativa a C++.

Necessitava un llenguatge que funcionés tant en els ordinadors amb sistema GNU/Linux del nostre laboratori com en els sistemes Windows i Macintosh que la majoria d'estudiants tenen a casa. Jo volia que fos programari lliure per tal que els estudiants el poguessin utilitzar a casa independentment dels seus ingressos. Volia un llenguatge que fos utilitzat per programadors professionals i que a més tingués una comunitat de desenvolupadors activa. El llenguatge havia d'admetre tant el paradigma de programació imperatiu com el d'orientació a objectes. Però, sobretot, havia de ser fàcil d'aprendre i d'ensenyar. Quan vaig investigar les alternatives amb aquests objectius en ment, `Python` es va erigir com el millor candidat per la feina.

Vaig demanar a un estudiant privilegiat de Yorktown, el Matt Ahrens, que donés una oportunitat a `Python`. Al cap de dos mesos no només havia après el llenguatge, sinó que a més havia escrit una aplicació anomenada pyTicket amb la que el personal del centre podia avisar de problemes tecnològics a través de la web. Jo sabia que el Matt no era capaç d'escriure en C++ una aplicació d'aquesta dimensió en un període de temps tan curt, així doncs donada aquesta fita, combinada amb la valoració positiva de `Python` feta pel Matt, em va semblar que `Python` era la solució que estava buscant.

2 Com vaig trobar un llibre de referència

Un cop decidit a utilitzar `Python` a partir del pròxim any en els meus cursos d'introducció a la informàtica, el problema més urgent era la mancança d'un llibre de referència disponible.

En aquest punt, els documents lliures em van ajudar. A començaments d'any, en Richard Stallman m'havia presentat l'Allen Downey. Els dos havíem escrit al Richard tot mostrant-li interès a desenvolupar material educatiu lliure. L'Allen ja havia escrit un llibre d'informàtica pels alumnes de primer curs, el "*How to Think Like a Computer Scientist*". Només acabar de llegir aquest llibre ja vaig saber que el voldria utilitzar en els meus cursos. Era el llibre d'informàtica més clar i de més ajuda amb el que mai m'havia topat ja que remarcava els processos de pensament involucrats en la programació en comptes de les propietats d'un llenguatge en particular. De fet, després de la lectura del llibre vaig créixer com a professor.

El "*How to Think Like a Computer Scientist*" no només era un llibre excel·lent sinó que a més havia estat publicat sota la llicència GNU Public License, el qual significava que es podia utilitzar lliurament i modificar fins a acomplir les necessitats de l'usuari. Un cop feta la decisió d'utilitzar `Python` se'm va ocórrer que podia traduir a aquest nou llenguatge la versió original del llibre de l'Allen, el qual estava escrit per Java. Tot i que jo no hauria estat capaç d'escriure un llibre de referència, sí que m'era possible escriure a partir del llibre de l'Allen. Al mateix temps, això demostrava que el desenvolupament cooperatiu utilitzat amb tants bons resultats al programari també podia funcionar pels materials educatius.

El desenvolupament del llibre durant els darrers dos anys ha estat profitós tant pels meus estudiants com per mi mateix i, a més, els meus estudiants han pres part en el procés. Com que jo podia canviar instantàniament qualsevol error d'escriptura o els trossos difícils que trobessin, els vaig animar a trobar errors al llibre tot donant-los punts extra cada cop que es produís un canvi en el text gràcies als seus suggeriments. Això era doblement beneficiós ja que per una banda els encoratjava a llegir amb atenció el llibre i per altra banda s'obtenia una revisió profunda del llibre amb els ulls crítics més importants: els estudiants que utilitzen el llibre per aprendre informàtica.

Per fer la segona meitat del llibre, el qual tracta la programació orientada a objectes, jo ja sabia que faria falta algú més experimentat amb la programació real. Durant la major part de l'any, el llibre es va quedar a mitges fins que la comunitat de programari lliure, un cop més, va aportar els recursos per acabar-lo.

Un dia vaig rebre un correu electrònic d'un tal Chris Meyers a on m'expressava que tenia interès en el llibre. El Chris és un programador professional que el darrer any havia començat a ensenyar en un curs de programació de `Python` al Lane Community College d'Eugene, a Oregon. Durant la planificació del curs el Chris havia topat amb el llibre i de seguida va voler donar un cop de mà. Al final del curs acadèmic ja havia creat un projecte de col·laboració anomenat *`Python` for Fun* en el nostre lloc web `http://openbookproject.net` i treballava com a professor avançat d'alguns dels meus estudiants privilegiats, tot guiant-los més enllà dels meus límits.

3 `Python` com a introducció a la programació

El procés de traducció i la utilització del *How to Think Like a Computer Scientist* durant els dos darrers anys, ha confirmat la idoneïtat de `Python` per a ensenyar els estudiants de primer curs. `Python` simplifica enormement els exemples de programació i facilita l'ensenyament dels conceptes bàsics de programació.

El primer exemple del llibre demostra aquesta idea. Tradicionalment, és el programa que mostra "*Hola Mon!*", el qual en la versió Java del llibre s'expressa com:

```
class Hola {
  public static void main (String[] args) {
      System.out.println ("Hola Mon!.");
  }
}
```

però en la versió de Python esdevé:

```
print "Hola Mon!!"
```

Encara que sigui un exemple senzill, els avantatges de Python es fan evidents. La informàtica que ensenyo a Yorktown no té prerequisits i per tant molts dels estudiants que veuen l'exemple es troben al davant el seu primer programa. Indubtablement, alguns estan una mica nerviosos ja que han sentit que la programació informàtica té un aprenentatge difícil. Amb la versió Java sempre m'he trobat amb el dilema d'escollir entre dues opcions insatisfactòries: o bé explico les sentències "class Hola", "public static void main", "String[] args", "{" i "}" tot arriscant-me a confondre o intimidar els alumnes només començar, o bé els dic: "No us preocupeu d'aquestes coses, en parlarem més endavant" també arriscant-me al mateix. Els objectius educatius en aquest nivell de curs són introduir la idea de sentència als estudiants i permetre que escriguin el seu primer programa, al mateix temps que s'introdueixen a l'entorn de programació. El programa amb Python té exactament el necessari per complir amb aquests objectius, i no afegeix res més.

Si es compara el text d'explicació que acompanya el programa en cada versió del llibre, es demostra què significa per als estudiants que comencen. En la versió de Java hi ha set paràgrafs per explicar el "Hola Mon!"; en canvi a la versió de Python n'hi ha prou amb unes quantes frases. Però el més important és que els sis paràgrafs de més no desenvolupen grans conceptes de la programació informàtica sinó que es dediquen a les minúcies de la sintaxi de Java. De fet, tot el llibre pateix el mateix problema. A la versió de Python, paràgrafs complets desapareixen simplement gràcies a que la sintaxi més planera de Python els fa innecessaris.

Quan s'usa un llenguatge d'alt nivell com és Python, el professor pot endarrerir l'entrada en detall del baix nivell de l'ordinador fins que els estudiants tenen un coneixement suficient per captar-ne l'essència, de manera que es poden assentar les coses pedagògicament. Un exemple que ho demostra és la manera amb que Python manipula les variables. A Java, una variable és un nom d'un lloc a on s'emmagatzema un valor si és un tipus bàsic i en cas contrari és una referència a un objecte. Per explicar aquesta diferència cal detallar com s'emmagatzemen les dades a l'ordinador, així doncs, el concepte de variable està lligada amb el maquinari de l'ordinador. La potència i els conceptes fonamentals de les variables ja són prou difícils pels estudiants de primer (tant en informàtica com en àlgebra) i els bytes i les adreces de memòria no hi ajuden. A Python una variable és un nom que referencia una cosa. Això és de lluny molt més intuïtiu per als estudiants de primer i és molt més pròxim al significat de variable que aprenen a matemàtiques. Aquest any que he ensenyat les variables d'aquesta manera, he tingut menys dificultat a explicar-les i he dedicat menys temps a resoldre dubtes sobre el seu ús.

La sintaxi de les funcions és un altre exemple de com Python millora l'ensenyament i l'aprenentatge de la programació. Els meus estudiants sempre havien presentat dificultat en la comprensió de les funcions. El problema principal rau en la diferència entre la definició de la funció i la crida a la funció, a més del problema relacionat de distingir entre un paràmetre i un argument. En Python se soluciona el problema ni més ni menys que amb la sintaxi. Les definicions de les funcions comencen amb la paraula clau **def**, així que als estudiants simplement els dic "Quan definiu una funció comenceu amb **def** i a continuació poseu-hi el nom de la funció que esteu definint"; per a cridar-la n'hi ha prou amb anomenar-la, és a dir escriure el seu nom. Els paràmetres s'usen a les definicions, els arguments s'usen a les crides. No hi ha declaració dels tipus retornats, declaració dels tipus dels paràmetres ni paràmetres apuntadors i de referència al valor que destorbin. Així que ara sóc capaç d'ensenyar funcions en menys de la meitat del temps que trigava abans, i a més millorant la comprensió.

L'ús de `Python` va millorar l'efectivitat del nostre curs d'informàtica per a tots els estudiants. Vaig observar un millor nivell general d'èxit i menor frustració que el que havia experimentat quan ensenyava C++ o Java. De seguida vaig obtenir millors resultats. La majoria d'estudiants sortien del curs amb la capacitat de crear programes amb sentit i amb l'actitud positiva cap a la programació que ella mateixa encomana.

4 Com s'ha construït la comunitat

He rebut correus de gent d'arreu del món que utilitza aquest llibre per aprendre o per ensenyar a programar. S'ha començat a crear una comunitat d'usuaris i molta gent ja ha contribuït al projecte enviant materials per compartir al lloc web `http://openbookproject.net/pybiblio`.

Ara que cada cop el `Python` creix més, es pot pronosticar que el creixement de la comunitat d'usuaris continuarà i s'accelerarà. Una de les parts que m'ha engrescat més d'aquest projecte ha estat precisament la creació d'aquesta comunitat d'usuaris, juntament amb el fet que això fa pensar que es poden aplicar esquemes similars de col·laboració entre educadors. Si treballem junts podem augmentar la qualitat dels materials disponibles que utilitzem i d'aquesta manera estalviar temps. Us convido a unir-vos a la nostra comunitat i espero rebre notícies vostres ben aviat. Podeu contactar amb mi a l'adreça de correu `jeff@elkner.net`.

Jeffrey Elkner
Governor's Career and Technical Academy in Arlington
Arlington, Virginia

Llista de contribuïdors

Parafrasejant la filosofia de la *Free Software Foundation*, aquest llibre és lliure, però no necessàriament gratuït[1]. Aquest llibre és fruit d'un procés de col·laboració que no hauria estat possible sense la llicència *GNU Free Documentation License*. Per això, volem agrair a la *Free Software Foundation* haver desenvolupat aquesta llicència i, és clar, haver-la posat a la nostra disposició.

També volem fer arribar el nostre agraïment a més d'un centenar de lectors atents i perspicaços que ens han enviat suggeriments i correccions durant els darrers anys. Seguint l'esperit del programari lliure, volem expressar-los el nostre agraïment mitjançant una llista de contribuïdors. Malauradament, aquesta llista és incompleta malgrat els nostres esforços per mantenir-la al dia. Per altra banda, està creixent massa per incloure tothom qui ha enviat una o dues correccions. Aquestes persones tenen el nostre agraïment i la satisfacció personal d'haver contribuït a millorar un llibre que els ha resultat d'utilitat. Afegirem a la llista de contribuïdors a la segona edició les persones que estan contribuint a hores d'ara.

Si resseguiu la llista, comprovareu que els suggeriments i les correccions aportades per cada un dels contribuïdors us han estalviat, i estalviaran a futurs lectors, confusions provocades o bé per errors tècnics, o bé per explicacions poc clares.

Malgrat que sembli impossible després de tantes correccions, encara podrien haver-hi errors en aquest llibre. Si en trobeu algun, esperem que trobeu el moment per contactar amb nosaltres. L'adreça de correu electrònic és `jeff@elkner.net`. Si els vostres suggeriments provoquen canvis substancials en el llibre, us afegirem a la propera versió de la llista de contribuïdors (llevat que demaneu no figurar-hi, és clar). Gràcies.

5 Segona Edició

1. Un correu electrònic de Mike MacHenry m'il·lustrà sobre recursivitat final. I, a més de descobrir un error en la presentació, suggerí com esmenar-lo.

2. No fou fins que l'estudiant de cinquè grau Owen Davies va assistir a una classe d'aprofundiment de `Python` un dissabte al matí i em va dir que volia escriure el joc de cartes *Gin Rummy* en `Python` quan vaig descobrir que volia usar-lo com a cas d'estudi per als capítols sobre programació orientada a objectes.

3. Vull dedicar un agraïment especial als estudiants pioners de la classe de programació en `Python` d'en Jeff al GCTAA del curs escolar 2009-2010: Safath Ahmed, Howard Batiste, Louis Elkner-Alfaro, i Rachel Hancock. Els vostres continus i assenyats comentaris van motivar canvis en la major part dels capítols del llibre. Vosaltres vàreu establir el perfil d'estudiant actiu i compromès que ajudarà la nova Governor's Academy a esdevenir allò que hagi d'esdevenir. Gràcies a vosaltres puc afirmar que aquest text ha estat provat per estudiants.

4. De la mateixa manera, vull dedicar un agraïment als estudiants de la classe d'Informàtica d'en Jeff al programa HB-Woodlawn del curs escolar 2007-2008: James Crowley, Joshua Eddy, Eric Larson, Brian McGrail, i Iliana Vazuka.

[1] En anglès *free* es pot referir tant a lliure com a gratuït. En aquest cas, el sentit és lliure, com en *free speech* i no gratis com en *free pizza*.

5. Ammar Nabulsi envià un gran nombre de correccions dels capítols 1 i 2.
6. Aldric Giacomoni descobrí un error en la definició de la seqüència de Fibonacci al Capítol 5.
7. Roger Sperberg envià diverses correccions ortogràfiques i descobrí un fragment inconsistent al Capítol 3.
8. Adele Goldberg segué amb en Jeff a la PyCon 2007 i li va proporcionar tot un seguit de suggeriments i correccions d'arreu del llibre.
9. Ben Bruno envià correccions dels capítols 4, 5, 6 i 7.
10. Carl LaCombe descobrí que usàvem incorrectament el terme commutatiu al Capítol 6 quan el terme adient era simètric.
11. Alessandro Montanile envià correccions d'errors tant en el codi dels exemples com en el text dels capítols 3, 12, 15, 17, 18, 19, i 20.
12. Emanuele Rusconi trobà errors als capítols 4, 8, i 15.
13. Michael Vogt descrigué un error en el sagnat d'un exemple del Capítol 6, i envià un suggeriment per tal de clarificar la secció del Capítol 1 sobre les diferències entre l'intèrpret interactiu i els *scripts*.

6 Primera Edició

1. Lloyd Hugh Allen envià una correcció a la Secció 8.4.
2. Yvon Boulianne envià una correcció d'un error semàntic al Capítol 5.
3. Fred Bremmer proposà una correcció a la Secció 2.1.
4. Jonah Cohen escrigué els *scripts* en Perl per tal de convertir la font original en LaTeX d'aquest llibre a HTML.
5. Michael Conlon envià una correcció gramatical al Capítol 2 i una millora d'estil al Capítol 1, i va iniciar un debat sobre aspectes tècnics dels intèrprets.
6. Benoit Girard envià una correcció d'una errada divertida a la Secció 5.6.
7. Courtney Gleason i Katherine Smith escrigueren horsebet.py, que es va usar com a cas d'estudi en una versió preliminar del llibre. El seu programa ara es pot trobar al lloc web.
8. Lee Harr proposà més correccions de les que podem encabir aquí, i, per descomptat, hauria de ser considerat un dels editors principals del text.
9. James Kaylin és un estudiant que usa el text. Ha proposat un bon nombre de correccions.
10. David Kershaw corregí una versió incorrecta de la funció catTwice de la Secció 3.10.
11. Eddie Lam ha enviat un bon nombre de correccions als capítols 1, 2, i 3. Ha corregit el Makefile per tal que generi un índex el primer cop que s'executi i ens va ajudar a establir un esquema de versions.
12. Man-Yong Lee envià una correcció del codi de l'exemple de la Secció 2.4.
13. David Mayo va descobrir que calia canviar la paraula *unconsciously* per *subconsciously* al Capítol 1.
14. Chris McAloon envià diverses correccions a les seccions 3.9 i 3.10.

15. Matthew J. Moelter ha estat contribuint al llibre durant molt de temps amb un bon nombre de correccions i suggeriments.

16. Simon Dicon Montford detectà que mancava la definició d'una funció i diversos errors tipogràfics al Capítol 3. També trobà errors a la funció increment del Capítol 13.

17. John Ouzts corregí la definició de valor de retorn al Capítol 3.

18. Kevin Parks envià valuosos comentaris i suggeriments per tal de millorar la distribució del llibre.

19. David Pool detectà un error tipogràfic en el glossari del Capítol 1 i, a més, ens encoratjà amablement.

20. Michael Schmitt envià una correcció al capítol sobre fitxers i excepcions.

21. Robin Shaw descobrí un error a la Secció 13.1, on la funció printTime s'usava sense haver estat definida.

22. Paul Sleigh trobà un error al Capítol 7 i un altre a l'*script* en Perl de Jonah Cohen que genera HTML a partir de LaTeX.

23. Craig T. Snydal està usant el text en un curs a la Drew University. Ha contribuït amb valuosos suggeriments i correccions.

24. Ian Thomas i els seus estudiants estant usant el text en un curs de programació. Ells han estat els primers a comprovar els capítols de la darrera meitat del llibre i hi han fet nombroses correccions i suggeriments.

25. Keith Verheyden envià una correcció al Capítol 3.

26. Peter Winstanley ens va fer adonar d'un error molt antic en el nostre llatí en el Capítol 3.

27. Chris Wrobel corregí el codi del capítol sobre fitxers i excepcions.

28. Moshe Zadka ha fet contribucions molt valuoses a aquest projecte. A més d'escriure el primer esborrany del capítol sobre diccionaris, ha proporcionat un guiatge continuat durant els primers estadis del llibre.

29. Christoph Zwerschke envià diverses correccions i suggeriments pedagògics, i ens explicà la diferència entre *gleich* i *selbe*.

30. James Mayer descobrí un gran nombre d'errors ortogràfics i tipogràfics, incloent-ne dos en la llista de contribuïdors.

31. Hayden McAfee ens advertí d'una inconsistència entre dos exemples que podia induir a confusió.

32. Angel Arnal forma part d'un equip internacional de traductors que treballen en la versió del text en espanyol. També ha descobert diversos errors en la versió anglesa.

33. Tauhidul Hoque i Lex Berezhny creà les il·lustracions del Capítol 1 i en millorà moltes de les altres.

34. Dr. Michele Alzetta detectà un error al Capítol 8 i feu interessants comentaris i suggeriments pedagògics sobre Fibonacci i Old Maid.

35. Andy Mitchell descobrí un error tipogràfic en el Capítol 1 i un exemple incorrecte en el Capítol 2.

36. Kalin Harvey suggerí un aclariment en el Capítol 7 i detectà alguns errors tipogràfics.

37. Christopher P. Smith detectà diversos errors tipogràfics i ens està ajudant a preparar l'actualització del llibre per a `Python` 2.2.

38. David Hutchins detectà un error tipogràfic en el Pròleg.

39. Gregor Lingl ensenya `Python` en un institut de Vienna, Austria. Està treballant en la traducció a l'alemany del llibre i ha descobert un parell d'errors greus en el Capítol 5.

40. Julie Peters detectà un error tipogràfic en el Prefaci.

1 El mètode de programar

L'objectiu d'aquest curs és ensenyar-te a pensar com un informàtic. Aquesta manera de pensar combina algunes de les millors característiques de les matemàtiques, l'enginyeria i les ciències naturals. Com els matemàtics, els enginyers informàtics usen llenguatges formals per denotar idees, particularment càlculs. Com els enginyers, dissenyen coses, adapten components a sistemes i avaluen les avantatges entre les diferents alternatives. Com els científics, observen el comportament de sistemes complexos, formulen hipòtesis i comproven les prediccions.

L'habilitat més important per a un informàtic és *la capacitat de resoldre problemes*. Resoldre problemes requereix saber formular els problemes, pensar de manera creativa en les seves solucions i conèixer com expressar les solucions de manera clara i acurada. Com es fa evident, el procés d'aprendre a programar és una oportunitat excel·lent de practicar l'habilitat de resoldre problemes. És per aquest motiu que aquest capítol s'anomena *El mètode de programar*.

Per una banda, aprendràs a programar, una habilitat útil per si mateixa. D'altra banda, usaràs la programació com a un medi per aconseguir un fi. A mida que avancem, aquest fi quedarà més clar.

1.1 El llenguatge de programació `Python`

El llenguatge de programació que aprendràs serà `Python`. `Python` és un exemple de *llenguatge d'alt nivell*; altres llenguatges d'alt nivell dels quals potser hauràs sentit a parlar són `C++`, `PHP` i `Java`.

llenguatge d'alt nivell (high level language)

Com potser ja has deduït del nom "llenguatge d'alt nivell", també existeixen els *llenguatges de baix nivell*, a vegades anomenats *llenguatge màquina* o *llenguatge assemblador*. Parlant de manera planera, els ordinadors només poden executar programes escrits en llenguatges de baix nivell. Així, els programes escrits en llenguatges d'alt nivell han de ser processats abans de poder funcionar. Aquest processat pron una mica de temps, la qual cosa és una de les petites desavantatges dels llenguatges d'alt nivell.

llenguatges de baix nivell (low level languages)

llenguatge màquina (machine language)

llenguatge assemblador (assembler)

Però els avantatges són enormes. Primer, és molt més senzill escriure un programa en un llenguatge d'alt nivell. Els programes escrits en llenguatges d'alt nivell també prenen menys temps en ser escrits, són més curts i fàcils de llegir i molt més senzills de corregir. Segon, els programes escrits en llenguatges d'alt nivell són *portables*, la qual cosa vol dir que poden funcionar en diversos tipus de màquina amb poques o cap modificació. Els llenguatges de baix nivell només poden funcionar en un tipus de màquina i han de ser reescrits per tal de funcionar en una altra.

portables (portable)

Degut a aquests avantatges, gairebé tots els programes estan escrits en llenguatges d'alt nivell. Els llenguatges de baix nivell només s'usen per alguns programes molt especialitzats.

Existeixen dos tipus de programa que processen els programes escrits en llenguatges d'alt nivell i els transformen en programes escrits en llenguatges de baix nivell: els *intèrprets* i els *compiladors*. Un intèrpret llegeix un programa escrit en un llenguatge d'alt nivell i l'executa. En l'àmbit de la programació, *executar* significa dur a terme els càlculs que s'especifiquen al programa. L'intèrpret executa el programa de mica en mica, alternant la lectura de línies i duent a terme càlculs.

intèrprets (interpreter)

compiladors (compiler)

executar (empty)

Un compilador llegeix el programa i el tradueix completament abans de què aquest comenci a executar-se. En aquest cas, el programa en alt nivell s'anomena *codi font* i el programa traduït s'anomena *codi objecte* o *executable*. Una vegada un programa s'ha compilat, es pot executar repetidament sense cap

codi font (source code)

codi objecte (object code)

executable

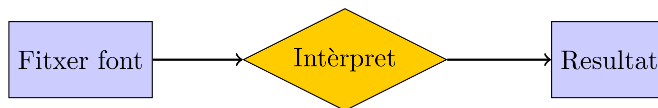


Figura 1.1: Procés d'interpretació

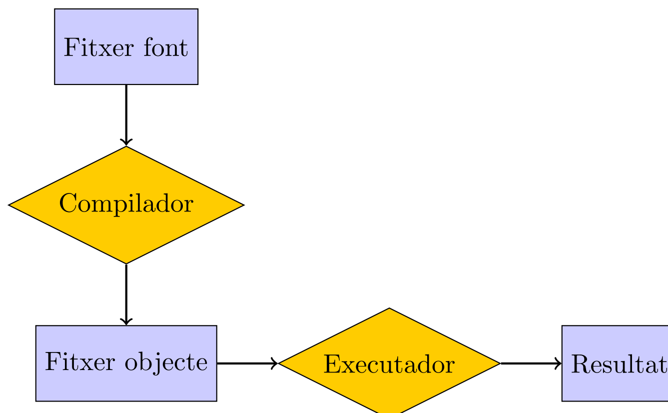


Figura 1.2: Procés de compilació

més traducció.

Molts llenguatges moderns usen ambdós processos. Primer són compilats a un llenguatge de més baix nivell, anomenat *codi intermedi*, i després són interpretats per un programa anomenat *màquina virtual*. `Python` usa ambdós processos, però degut a com interaccionen els programadors amb ell, sovint es considera com un llenguatge interpretat.

codi intermedi (byte code)

màquina virtual (virtual machine)

Existeixen dues maneres d'usar l'intèrpret de `Python`: el *mode interactiu* i el *mode comanda*. En el mode interactiu, s'escriuen sentències `Python` al terminal de `Python` i l'intèrpret imprimeix immediatament el resultat:

mode interactiu (shell mode)

mode comanda (script mode)

```
$python
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.
>>> print 1 + 1
2
>>> print "Abac"
Abac
```

La primera línia d'aquest exemple és la comanda que inicia l'intèrpret de `Python` en mode interactiu en un *terminal* de l'entorn Unix[1]. Les següents tres línies són missatges de l'intèrpret de `Python`. La quarta línia comença amb >>>, que és el *prompt* de l'intèrpret de `Python`. L'intèrpret usa el prompt per indicar-nos que està llest per rebre instruccions. Hem escrit **print** 1 + 1 i l'intèrpret ha respost 2.

terminal (command terminal)

prompt (prompt)

Alternativament, hauríem pogut escriure un programa en un fitxer i usar l'intèrpret de `Python` en mode comanda per executar el contingut d'aquest fitxer. Aquesta mena de fitxers s'anomenen *scripts*. Per exemple, hem usat un editor de texts per crear un fitxer anomenat `firstprogram.py` amb el següent contingut:

scripts (script)

```
print 1+1
```

[1] Unix és una família de sistemes operatius. GNU/Linux és un sistema de la família Unix.

Per conveni, els fitxers que contenen programes `Python` tenen noms acabats amb l'extensió `.py`.

Per executar el programa, hem d'escriure el nom del fitxer que conté el programa `Python` des del terminal de l'entorn Unix:

```
$ python firstprogram.py
2
```

L'intèrpret de `Python` llegeix el fitxer i l'executa, escrivint el resultat per pantalla.

Aquests exemples mostren `Python` funcionant des d'una terminal de comandes Unix. En altres entorns de desenvolupament, els detalls d'executar programes poden variar. A més, la majoria de programes són més interessants que el que acabem de veure.

Els exemples d'aquest curs usen ambdós modes de `Python`: el mode interactiu i el mode comanda. Seràs capaç de diferenciar-los perquè els exemples en mode interactiu sempre començaran amb el prompt de `Python`.

Treballar amb el mode interactiu és convenient per comprovar fragments curts de codi, ja que es rep resposta immediatament. Imagina't que és com un tros de paper que uses per ajudar-te a solucionar problemes. Qualsevol càlcul més llarg d'unes poques línies s'hauria d'escriure en un script.

1.2 Què és un programa?

programa (program)

Un *programa* és *una seqüència de sentències que especifiquen com dur a terme un càlcul.* El càlcul pot ser de tipus matemàtic (com resoldre un sistema d'equacions o trobar les arrels d'un polinomi), però també pot ser un càlcul simbòlic (com cercar i reemplaçar un text en un document o compilar un programa).

sentències bàsiques (empty)

Els detalls varien en els diversos llenguatges, però existeixen una sèrie de *sentències bàsiques* que apareixen gairebé a tots els llenguatges. Aquestes sentències serveixen per:

Entrada Obtenir dades del canal d'entrada (del teclat, d'un fitxer o de qualsevol altre dispositiu).

Sortida Mostrar dades pel canal de sortida (a la pantalla, enviant-les a un fitxer o un altre dispositiu).

Càlcul Dur a terme operacions matemàtiques bàsiques (com les sumes o les multiplicacions).

Condicional Comprovar certes condicions i executar la seqüència de sentències adient segons el resultat de la comprovació.

Iteració Dur a terme determinades accions de manera repetitiva, normalment amb alguna variació.

Ho creguis o no, això és pràcticament tot. Qualsevol programa que hagis pogut usar, no importa com de complicat fos, està construït amb sentències (a vegades també s'anomenen instruccions) com les que acabes de veure. Així, podem descriure el fet de programar com el procés de trencar una tasca llarga i complexa en subtasques cada cop més i més petites, fins que les subtasques són prou senzilles com per ser dutes a terme amb una de les sentències bàsiques.

Això pot sonar una mica vague, però ja tornarem a insistir-hi més endavant quan parlem d'algorismes.

1.3 Què és depurar?

errors (bug)

Programar és un procés complex i, atès que el fan éssers humans, sovint conté *errors*. El procés de trobar-los i corregir-los s'anomena *depuració*.

errors semàntics (semantic error)

errors sintàctics (syntax error)

En un programa podem trobar tres tipus d'errors: *errors sintàctics*, *errors en temps d'execució* i *errors semàntics*. És força útil saber distingir els tres tipus d'error per tal de poder-los localitzar i corregir de la manera més ràpida. Els següents apartats concreten aquests tipus d'errors.

1.4 Errors sintàctics

errors en temps d'execució (runtime error)

`Python` només pot executar un programa si la seva sintaxi és correcta; d'altra manera, el procés falla i retorna un missatge d'error. La sintaxi es refereix a l'estructura del programa i a les normes sobre aquesta estructura. Per exemple, en Català, una frase ha de començar amb una lletra en majúscula i acabar amb un punt. aquesta frase conté un error sintàctic. Aquesta també

Per a molts lectors, uns quants errors sintàctics no són un problema significatiu, però `Python` no és tant permissiu. Si hi ha un sol error sintàctic a qualsevol punt del teu programa, `Python` et mostrarà un missatge d'error i acabarà, i no seràs capaç de fer funcionar el teu programa. Durant les teves primeres setmanes com a programador, passaràs força temps localitzant i corregint errors sintàctics. A mida que vagis adquirint experiència però, cada vegada faràs menys errors i seràs capaç de trobar-los més de pressa.

1.5 Errors en temps d'execució

excepcions (exception)

El segon tipus d'error són els errors en temps d'execució, anomenats així perquè no apareixen fins que no executes el teu programa. Aquests errors també s'anomenen *excepcions* ja que sovint indiquen que alguna cosa excepcional —i dolenta— ha succeït.

Els errors en temps d'execució són estranys als programes senzills que es veuen en els primers capítols, així que encara passarà un temps fins que te'n trobis algun.

1.6 Errors semàntics

El tercer tipus d'error són els errors semàntics. Si el teu programa té un error semàntic, el programa funcionarà correctament. L'ordinador no mostrarà cap missatge d'error, però no calcularà el resultat correcte. Obtindrà un resultat diferent al que tu esperaves.

El problema rau en què el programa que has escrit no és el programa que volies escriure. El significat del programa —la seva semàntica— està malament. Identificar els errors semàntics pot ser complicat ja que, a partir del resultat incorrecte que s'ha obtingut, cal que dedueixis què pot haver passat.

1.7 Depuració experimental

Una de les habilitats més importants que adquiriràs serà la de depurar. Tot i que pot ser frustrant, depurar és una de les activitats més intel·lectualment enriquidores, desafiant i interessants de programar.

D'alguna manera, depurar és com fer de detectiu. T'enfrontes a pistes, i tu has d'inferir els processos i els esdeveniments que han dut als resultats que veus.

Depurar també és com una ciència experimental. Un cop tens una idea de què està anant malament, modifiques el programa i tornes a provar-ho. Si la teva hipòtesi era correcta, aleshores pots predir el resultat de la modificació i ja estàs un pas més a prop d'un programa que funciona. Si la teva hipòtesi era incorrecta, has d'intentar-ho amb una altra. Com deia Sherlock Holmes...

> *una vegada eliminat l'impossible, el que queda, per improbable que sigui, és la veritat*
>
> (A. Conan Doyle, *The Sign of Four*).

Per algunes persones, programar i depurar són el mateix. És a dir, programar és com dur a terme un procés de depuració gradual d'un programa, fins que aquest dugui a terme el que tu vols que faci. La idea consisteix en que hauries de començar amb un programa que faci "alguna cosa" i anar fent petites modificacions, depurant-les a mida que vas avançant, de manera que sempre tinguis un programa que funciona.

Per exemple, `Linux` és un nucli de sistema operatiu que conté milions de línies de codi, però aquest va començar com un simple programa que Linus Torvalds usava per explorar un xip *Intel 80386*. D'acord amb Larry Greenfield, un dels primers projectes de Linus va ser un programa que imprimia de forma alterada `AAAA` i `BBBB`. Aquest més tard va evolucionar fins a ser `Linux`[2].

En lliçons posteriors farem més suggeriments sobre pràctiques de programació i depuració.

1.8 Llenguatges formals i naturals

llenguatges naturals (natural language)

Els *llenguatges naturals* són els llenguatges que usa la gent per parlar, com el Català, el Castellà i l'Anglès. No van ser dissenyats per la gent (tot i que les persones intenten establir-hi alguna mena d'ordre); van evolucionar naturalment.

llenguatges formals (formal language)

Els *llenguatges formals* són llenguatges dissenyats per la gent per a aplicacions específiques. Per exemple, la notació que els matemàtics usen és un llenguatge formal particularment bo en denotar relacions entre símbols i nombres. Els químics usen un llenguatge formal per representar l'estructura de les mol·lècules. I el més important: *Els llenguatges de programació són llenguatges formals que han estat dissenyats per expressar càlculs.*

regles sintàctiques (syntactic rules)

Els llenguatges formals tendeixen a tenir regles estrictes sobre sintaxi. Per exemple, $3 + 3 = 6$ és una expressió matemàtica correcta pel que fa a la sintaxi, però $3 = +6\$$ no ho és. H_2O és una fórmula química de sintaxi correcta, però H_2Zz no ho és.

Les *regles sintàctiques* venen donades per dos conceptes, els *símbols* i les *estructures*. Els símbols són els elements bàsics del llenguatge, com les paraules, nombres i elements químics. Un dels problemes amb $3 = +6\$$ és que el \$ no és un símbol legal a les matemàtiques —si més no, que nosaltres sapiguem— De manera similar, H_2Zz no és legal perquè no hi ha cap element químic amb el símbol Zz.

El segon tipus de regla sintàctica es refereix a l'estructura d'una sentència; és a dir, a la manera en què els símbols es col·loquen. La sentència $3 = +6\$$ és incorrecta pel que fa a la sintaxi atès que no es

[2] *The Linux User's Guide* Beta Version 1)

pot posar un símbol suma immediatament després d'un símbol igual. De manera similar, les fórmules moleculars han de tenir els subíndexs just després del nom d'un element, no abans.

Quan llegeixes una frase en Català o una sentència d'un llenguatge formal, t'has d'imaginar quina és l'estructura de la frase —tot i que en un llenguatge natural això es fa de manera inconscient. Aquest procés s'anomena *anàlisi sintàctica*.

anàlisi sintàctica (parsing)

Per exemple, quan sents la frase, *La gota que fa vessar el got*, entens que *la gota* és el subjecte i *fa* és el verb. Una vegada has analitzat una frase, pots extreure'n què significa, o la semàntica de la frase. Assumint que saps què és una sabata i què significa caure, podràs entendre la implicació general de la frase.

Tot i que els llenguatges formals i els llenguatges naturals tenen moltes característiques en comú (símbols, estructures, sintaxi i semàntica), hi ha força diferències:

ambigüitat El llenguatges naturals estan plens d'ambigüitat, amb la qual la gent s'hi enfronta donant pistes contextuals i altres informacions. D'altra banda, els llenguatges formals estan dissenyats per ser gairebé o completament no ambigus, la qual cosa significa que una sentència té un sol significat, independentment del context.

redundància Per tal de lluitar amb l'ambigüitat i evitar malentesos, els llenguatges naturals usen molt la redundància. Com a resultat d'això, sovint són verbosos. En canvi, els llenguatges formals són menys redundants i més concisos.

literalitat Els llenguatges naturals estan plens d'idiotismes i metàfores. Si algú diu, «la gota que fa vessar el got», probablement no hi hagi cap got ni cap gota fent-lo vessar. Contràriament, els llenguatges formals volen dir exactament el que diuen.

La gent que creix parlant un llenguatge natural —tothom— a vegades ho passen malament per ajustar-se a un llenguatge formal. D'alguna manera, les diferències entre un llenguatge formal i un natural són com les diferències entre la poesia i la prosa:

Poesia Les paraules s'usen pel seu so així com pel seu significat, i el poema sencer crea un efecte o una resposta emocional. L'ambigüitat no només és comuna sinó sovint del·liberada.

Prosa El significat literal de les paraules és més important i l'estructura aporta significat. La prosa és menys sensible a l'anàlisi que la poesia però sovint també és ambigua.

Programes El significat d'un programa és no ambigu i literal i pot ser entès enterament mitjançant l'anàlisi dels símbols i les estructures.

Aquí van unes quantes recomanacions per llegir programes —i altres llenguatges formals—. Primer, recorda que els llenguatges formals són molt més densos que els llenguatges naturals, així que es triga més en llegir-los. A més, l'estructura és molt important, així doncs normalment no és molt bona idea llegir-los de dalt a baix i d'esquerra a dreta. En comptes d'això, aprèn a analitzar-lo al teu cap, identificant els símbols i interpretant les estructures. Finalment, els detalls importen. Les petites coses, com els errors ortogràfics o la mala puntuació, dels quals pots gairebé oblidar-te'n en els llenguatges naturals, poden significar una gran diferència en els llenguatges formals.

1.9 El primer programa

Tradicionalment, el primer programa escrit en un nou llenguatge s'anomena «Hola, Món!» perquè tot el que fa és mostrar per pantalla les paraules `Hola, Mon!` En `Python`, té aquest aspecte:

```
print "Hola Mon!"
```

Aquest és un exemple de sentència d'escriptura que, tot i dir-se **print**, no imprimeix res en un paper. Solament escriu un valor a la terminal. En aquest cas, el resultat són les paraules

```
Hola, Mon!
```

Les cometes al programa marquen l'inici i la fi del valor; no surten al resultat.

Algunes persones jutgen la qualitat d'un llenguatge de programació basant-se en la senzillesa del programa "Hola, Món!". Si ens creiem aquest criteri, conclourem que costa fer-ho millor que `Python`!

Exercicis

Exercici 1.1 Escriviu una frase en Català amb una semàntica comprensible però una sintaxi incorrecta. Escriviu-ne una altra en la que la sintaxi sigui correcte però contingui errors semàntics.

Exercici 1.2 En aquest capítol es parla de tres conceptes diferents: fitxer, script i programa, que són fàcils de confondre. Defineix amb precisió què significa cada concepte i quina relació hi ha entre ells.

Exercici 1.3 L'intèrpret de `Python` pot usar-se de dues formes: en mode interactiu i en mode comanda (del sistema operatiu). Fes una llista de les avantatges i els inconvenients de treballar en cadascun dels modes.

2 Variables, expressions i sentències

2.1 Valors i tipus

Els valor són un dels elements fonamentals (com els caràcters i els números), que un programa manipula. Els valors que hem vist fins ara són dos: el resultat de sumar 1 + 1 i `"Hola, Mon!"`.

tipus (type)

Aquests valors són de *tipus* diferents: 2 és un enter i `"Hola Mon!"` és una *cadena*, anomenada així perquè conté una cadena de lletres. Tu —i l'intèrpret de `Python`— pots identificar una cadena perquè es troba delimitada per cometes.

cadena (string)

La sentència d'escriptura també funciona pels enters, com demostra aquest exemple amb l'intèrpret de `Python` en mode interactiu:

```
>>> print 4
4
```

Si no estàs segur del tipus d'un valor, l'intèrpret de `Python` te'l pot escriure:

```
>>> type("Hello, World!")
<type 'str'>
>>> type(17)
<type 'int'>
```

No sorprèn que les cadenes (*string* en anglès) pertanyin al tipus `str` i els enters (*integers* en anglès) pertanyin al tipus `int`. Els nombres amb decimals (*float* en anglès) pertanyen a un tipus anomenat `float` i es representen fent servir la notació anglesa que, per indicar on comença la part decimal fa servir el caràcter `'.'`, en comptes de la `','`. En el cas de `Python`, el `'.'` és obligatori:

```
>>> type(3.2)
<type 'float'>
```

Què passa amb els valors com ara `"17"` i `"3.2"`? Tenen aspecte de números, però es troben delimitats per cometes com les cadenes.

```
>>> type("17")
<type 'str'>
>>> type("3.2")
<type 'str'>
```

Són cadenes. Les cadenes a `Python` poden estar delimitades per cometes simples (`'`) o cometes dobles (`"`).

```
>>> type('This is a string.')
<type 'str'>
>>> type("And so is this.")
<type 'str'>
```

Les cometes dobles poden, a la vegada, contenir cometes simples dins seu, com per exemple a `"Pensar-s'ho"`, i les cometes simples poden contenir cometes dobles dins seu, com per exemple a `'Els cavallers que diuen "Ni"!'`.

Quan es volen escriure enters llargs, podem estar temptats d'usar els punts per separar grups de tres dígits, com a 1.000.000. Això és incorrecte ja que, com hem dit abans, el punt serveix per representar valors de tipus real.

2.2 Variables

variable (variable)

Una de les característiques més poderoses d'un llenguatge de programació és la capacitat de manipular variables. Una *variable* és un nom que fa referència a un valor.

assignació (assignment)

La sentència d'*assignació* crea noves variables i les hi dóna valor.

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

Aquest exemple fa tres assignacions. La primera assigna la cadena `"What's up, DOC?"` a una nova variable anomenada `message`. La segona emmagatzema l'enter 17 a `n`, i la tercera desa el número real 3.14159 a `pi`.

L'operador d'assignació, =, no s'ha de confondre amb la igualtat. La sentència d'assignació sempre té la mateixa forma: un nom de variable seguit de l'operador d'assignació i, finalment, un valor. Incomplir això és causa d'errors com el que obtindríeu en cas d'escriure:

```
>>> 17 = n
```

diagrama d'estat (state diagram)

Una manera habitual de representar les variables sobre el paper és escriure el seu nom amb una fletxa apuntant al valor de la variable. Aquest tipus d'esquema s'anomena *diagrama d'estat* perquè mostra en quin estat es troba cada variable. La figura 2.1 mostra el diagrama d'estat de les variables que hem definit a l'exemple anterior.

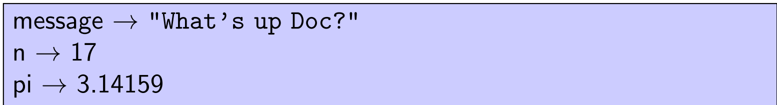


Figura 2.1: Exemple de diagrama d'estat

La sentència **print** també funciona per a les variables.

```
>>> print message
Whats up, Doc?
>>> print n
17
>>> print pi
3.14159
```

En cada cas, el que s'escriu és el valor de la variable. El tipus d'una variable també es pot consultar. Podem preguntar a l'intèrpret de `Python` a quin tipus pertanyen les següents variables:

```
>>> type(message)
<type 'str'>
>>> type(n)
<type 'int'>
>>> type(pi)
<type 'float'>
```

Per ser més precisos, les variables no tenen tipus. Quan preguntem sobre el tipus d'una variable ens estem referint realment al tipus del valor que conté la variable.

2.3 Noms de variable i paraules clau

Els programadors normalment usen noms de variables que siguin representatius; així documenten per a què s'està usant la variable.

Els noms de variable poden ser arbitràriament llargs. Poden contenir lletres i dígits, però han de començar obligatòriament amb una lletra. Tot i que està permès usar lletres majúscules, no ho farem mai. Si decidiu fer-ho, recordeu que podeu tenir problemes. Bruce i bruce són dues variables diferents.

La ratlla baixa (_) pot aparèixer al nom d'una variable. Sovint s'usa als noms de variable compostos per diverses paraules per substituir l'espai, com per exemple a la variable de nom nom_de_variable.

Si doneu un nom il·legal a una variable, obtindreu un error sintàctic.

```
>>> 76trombones = "big parade"
SyntaxError: invalid syntax
>>> more$ = 1000000
SyntaxError: invalid syntax
>>> class = "Computer Science 101"
SyntaxError: invalid syntax
```

76trombones és un nom de variable il·legal perquè no comença per una lletra. more$ és il·legal perquè conté un caràcter no permès, el signe del dòlar. Però, per què és erroni el nom **class**?

paraules clau (keyword)

Resulta que **class** és una de les *paraules clau* de Python. Les paraules clau les defineixen les normes i estructura del llenguatge, i no poden ser usades com a nom de variable.

Python té trenta-una paraules clau:

and	as	assert	break
class	continue	def	del
elif	else	except	exec
finally	for	from	global
if	import	in	is
lambda	not	or	pass
print	raise	return	try
while	with	yield	

T'interessarà tenir aquesta llista a mà. Si l'intèrpret es queixa d'un dels teus noms de variable i no saps perquè, mira't aquesta llista.

2.4 Sentències

sentència (sentence)

Una *sentència* és una instrucció que l'intèrpret de `Python` pot executar. Fins al moment hem vist dos tipus de sentències: **print** i l'assignació.

Quan escrivim una sentència a l'intèrpret de `Python` en el seu mode interactiu, aquest l'executa i escriu el resultat, si n'hi ha. El resultat d'una sentència **print** és un valor. La sentència d'assignació, en canvi, no escriu cap resultat.

Un programa conté una seqüència de sentències. Si hi ha més d'una sentència, els resultats es van escrivint d'un en un a mida que les sentències es van executant.

Per exemple, l'script:

```
print 1
x = 2
print x
```

Escriu la següent sortida:

```
1
2
```

De nou, tal i com ja hem dit, la sentència d'assignació no escriu cap sortida.

2.5 Avaluació d'expressions

expressió (expression)

Una *expressió* és una combinació de valors, variables i operadors. Si escrius una expressió, l'intèrpret de `Python` l'avalua i escriu el resultat:

```
>>> 1 + 1
2
```

L'avaluació d'una expressió genera un valor. Per aquest motiu, una expressió pot aparèixer a la banda dreta de les sentències d'assignació. Un valor per ell mateix també és una expressió i, de la mateixa forma, també ho és una variable.

```
>>> 17
17
>>> x
2
```

Cal adonar-se'n de que avaluar una expressió no és el mateix que escriure un valor.

```
>>> message = "What's up, Doc?"
>>> message
"What's up, Doc?"
>>> print message
What's up, Doc?
```

Quan l'interpret interactiu de `Python` mostra el valor d'una expressió, utilitza el mateix format que tu usaries per introduir un valor. En el cas de les cadenes, això vol dir que inclou les cometes. La sentència **print**, però, escriu el valor de l'expressió, que en aquest cas és el contingut de la cadena. La diferència entre una i altra acció és subtil però significativa.

En un script, una expressió per si mateixa és una sentència correcta, però no fa res de res. L'script:

```
17
3.2
"Hello, World!"
1 + 1
```

no escriu res si el fem executar per l'intèrpret de `Python`. Com canviaries aquest script per que escrivís els valors d'aquestes quatre expressions?

2.6 Operadors i operands

operadors (operators)

operands (operands)

Els *operadors* són uns símbols específics que representen càlculs, com ara la suma i la multiplicació. Els valors que utilitza un operador s'anomenen *operands*.

Els següents exemples són expressions correctes en `Python`, el seu significat el trobareu intuïtiu:

```
20+32
hour-1
hour*60+minute
minute/60
5**2
(5+9)*(15-7)
```

Els símbols +, − i / i l'ús dels parèntesis per agrupar, signifiquen en `Python` el mateix que signifiquen a les matemàtiques. L'asterisc (*) és el símbol de la multiplicació, i el doble asterisc (**) és el símbol de la potenciació.

Quan el nom d'una variable apareix al lloc d'un operand, se substitueix pel seu valor abans de què es calculi l'operació.

La suma, la resta, la multiplicació i la potenciació fan tot el que tu t'esperes que facin, però potser et sorprendràs amb la divisió. La següent operació obté un resultat no esperat:

```
>>> minute = 59
>>> minute/60
0
```

El valor de `minute` és 59, i 59 dividit entre 60 és 0.98333 no pas 0. La raó d'aquesta discrepància és que `Python` està efectuant una divisió entera.

Quan ambdós operands són enters, el resultat també ha de ser un enter i, per convenció, la divisió entera sempre arrodoneix a la baixa, fins i tot en casos com aquest en els quals el següent enter està molt a prop.

Una possible solució a aquest problema és calcular un percentatge en comptes d'una fracció:

```
>>> minute*100/60
98
```

De nou, el resultat s'arrodoneix a la baixa, però com a mínim ara la resposta és aproximadament correcta. Una altra alternativa és usar la divisió de coma flotant —o divisió real—. Això ho veurem al capítol 4: com convertir valors i variables de tipus enter en valors de coma flotant.

2.7 Ordre de les operacions

regles de precedència (priority rules)

Quan en una expressió apareix més d'un operador, l'ordre d'avaluació depèn de les *regles de precedència*. `Python` segueix les mateixes regles de precedència de les matemàtiques pels seus operadors matemàtics.

Els parèntesis tenen la precedència més alta i poden ser usats per forçar una expressió a ser avaluada en l'ordre que tu vols. Donat que les expressions entre parèntesis s'avaluen primer, 2*(3−1) és 4, i (1+1)**(5−2) és 8. També es poden usar els parèntesis per tal què una expressió sigui més senzilla de llegir, com en el cas (minute*100)/60, tot i que l'ús dels parèntesis no canvia el resultat. La potenciació té la següent precedència més alta, així 2**1+1 és 3 i no 4, i 3*1**3 és 3 i no 27. La multiplicació i la divisió tenen la mateixa precedència, que és més gran que la de la suma i la resta, que també tenen la mateixa precedència. Així 2*3−1 retorna 5 en comptes de 4, i 2/3−1 és −1 i no 1 (recorda que en la divisió entera, $\frac{2}{3} = 0$). Els operadors amb la mateixa precedència s'avaluen d'esquerra a dreta. Així a l'expressió minute*100/60, la multiplicació té lloc primer, retornant $\frac{5900}{60}$, que al seu temps retorna 98. Si els operadors s'haguessin avaluat de dreta a esquerra, el resultat hauria sigut $59 * 1$, que és 59 (que és incorrecte).

2.8 Operacions amb cadenes

En general no es poden efectuar operacions matemàtiques amb les cadenes, encara que aquestes puguin tenir aspecte de números. Els següents exemples són incorrectes (assumint que message és una variable de tipus cadena):

```
message-1
"Hello"/123
message*"Hello"
"15"+2
```

concatenació (concatenation)

Curiosament l'operador + funciona per les cadenes, tot i que no fa el que segurament t'esperes. Per a les cadenes, l'operador + representa la *concatenació*, la qual cosa vol dir unir els dos operands, enganxant-los un darrera l'altre. Per exemple:

```
fruit = "banana"
baked_good = " nut bread"
print fruit + baked_good
```

La sortida d'aquest programa és banana nut bread. L'espai abans de la paraula nut és part de la cadena, i es necessari per tal de generar l'espai entre les cadenes concatenades.

L'operador * també funciona per les cadenes. És l'operador de repetició. Per exemple, 'Fum'*3 és 'FumFumFum'. Un dels operands ha de ser una cadena i l'altre ha de ser un enter.

D'una banda, aquesta intrepretació de + i * té sentit fent l'analogia amb la suma i la multiplicació. Com que 4*3 és equivalent a 4+4+4, esperem que 'Fum'*3 sigui el mateix que 'Fum'+'Fum'+'Fum', i així és. D'altra banda, hi ha aspectes en què la concatenació i la repetició són diferents de la suma i la multiplicació d'enters. Quina propietat tenen la multiplicació i la suma que no tenen la concatenació i la repetició?

2.9 Entrada

predefinides (predefined)

Existeixen dues funcions *predefinides* en Python per llegir del teclat:

```
n = raw_input("Please enter your name: ")
print n
n = input("Enter a numerical expression: ")
print n
```

Una execució d'aquest script tindria aquest aspecte:

```
$ python tryinput.py
Please enter your name: Arthur, King of the Britons
Arthur, King of the Britons
Enter a numerical expression: 7 * 3
21
```

Cadascuna d'aquestes dues funcions permet escriure un missatge just abans de fer la lectura. Aquest missatge es passa a les funcions entre parèntesis.

2.10 Composició

Fins ara, hem fet una ullada als elements d'un programa (variables, expressions i sentències) de manera aïllada, sense parlar de com combinar-los.

Una de les funcionalitats més útils d'un llenguatge de programació és la capacitat de combinar petits fragments de codi. Per exemple, sabem com sumar números i sabem com escriure'ls; resulta doncs, que podem fer les dues coses a la vegada, com al següent exemple:

```
>>> print 17 + 3
20
```

En realitat, la suma ha de succeïr abans que l'escriptura, així que les accions no es fan realment al mateix temps. El fet important és que qualsevol expressió que involucri números, cadenes i variables pot ser usada dins d'una sentència **print**. Ja has vist un exemple d'això:

```
print "Number of minutes since midnight: ", hour*60+minut
```

També pots escriure expressions arbitràries a la banda dreta d'una sentència d'assignació:

```
percentage = (minute * 100) / 60
```

Aquesta capacitat pot no semblar molt impressionant ara mateix, però veuràs més exemples on la composició fa possible expressar càlculs complicats de manera clara i concisa.

Hi ha limitacions sobre a on pots usar expressions. Per exemple, a la banda esquerra d'una sentència d'assignació hi ha d'haver el nom d'una variable, no una expressió. Així, el següent exemple és incorrecte: `minute + 1 = hour`.

2.11 Comentaris

A mida que els programes es van fent més grans i complicats, també es fan més difícils de llegir. Els llenguatges formals són densos i sovint és difícil mirar un fragment de codi i entendre què està fent o perquè.

Per aquest motiu, és una bona idea afegir notes als teus programes per explicar en llenguatge natural què està fent el programa. Aquestes notes s'anomenen *comentaris*, i es marquen amb el símbol #:

comentaris (comment)

```
# calcula el percentatge de l'hora que ha transcorregut
percentage = (minute * 100) / 60
```

En aquest cas el comentari ocupa tota una línia. També es poden posar els comentaris al final d'una línia:

```
percentage = (minute * 100) / 60 # precaucio: divisio entera
```

Tot el que va del símbol # fins al final de la línia s'ignora; no té cap efecte en el programa. El missatge està destinat al programador o futurs programadors que vulguin usar el nostre codi. En aquest cas, recorda al lector del comportament sempre sorprenent de la divisió entera.

Exercicis

EXERCICI 2.1 Anoteu què pasa quan executem la següent sentència **print** en el mode interactiu de l'intèrpret:

```
>>> print n = 7
```

I què passa amb la següent?

```
>>> print 7 + 5
```

I per aquesta altra?

```
>>> print 5.2, "this", 4 - 2, "that", 5/2.0
```

EXERCICI 2.2 Determineu la norma general que explica el que es pot escriure darrera de la sentència **print**. Què retorna la sentència **print**?

EXERCICI 2.3 Considereu la següent frase:

> Molt treballar i poc jugar fan de Jack un avorrit.

Emmagatzemeu cada paraula en una variable diferent i, després, escriviu per pantalla la frase en una sola línia usant **print**.

EXERCICI 2.4 Afegiu parèntesis a l'expressió `6 * 1 - 2` de manera que el seu valor canvii de 4 a −6.

EXERCICI 2.5 Essent `a = "Hola"`, `b = "Mon"`, `c = 87` i `d = 2.33145`, escriviu expressions que s'avaluin a les següents cadenes:

a) `"-Hola-Mon-"` (usant `a` i `b`).

b) `"El resultat es: 87"` (usant `c`).

c) `"El resultat es: 87 minuts (5220 segons)"` (usant `c` les dues vegades).

d) `"La temperatura es: 2.3"` (usant `d`).

e) `"Hola Mon"` (usant `a` i `b`).

EXERCICI 2.6 Considereu els següents enunciats. Per a cadascun, usant l'editor de textos, escriviu un script `Python` que faci el càlcul que s'indica:

a) Escriu la suma de `8 + 7`.

b) Llegeix un nombre enter i escriu el resultat de dividir-lo per dos. Observeu què passa si el nombre és senar.

c) Llegeix dos nombres enters i escriu la suma i la mitjana.

d) Llegeix el nom i l'edat d'una persona i escriu un missatge amistós dirigit a aquesta persona.

EXERCICI 2.7 Usant l'intèrpret de `Python` en mode interactiu com a calculadora, calculeu el següent:

a) Calculeu la superfície d'un cercle de radi $r = 4.32$.

b) Calculeu, pel mètode del picapedrer, la superfície d'un cercle de radi $r = 12.3$. El mètode del picapedrer per calcular la superfície aproximada d'un cercle consisteix a calcular la superfície del quadrat que el circumscriu i després restar-ne un 20%.

c) Calculeu $\sum_{i=0}^{5} \frac{1}{2^i}$.

d) Calculeu el valor del polinomi $p(x) = 3x^3 - 2x^2 + 0.56x - 3$ quan $x = 0.437$.

EXERCICI 2.8 Quina de les següents expressions s'avalua de forma diferent :

a) `8 * 2 / 2 ** 3 * 4`

b) `2 * 8 / 2 ** 4 * 3`

c) `2 * 8 / 4 ** 2 * 3`

d) `8 * 2 / 4 ** 2 * 3`

EXERCICI 2.9 Escriviu un script per cadascun dels següents enunciats que calculi el que s'indica.

a) Llegeix una alçada h mesurada en metres i escriu el temps, expressat en segons, que trigaria un objecte en condicions ideals a caure des de l'alçada h.

b) Llegeix la velocitat v i l'angle θ respecte el terra amb que es dispara un projectil i, suposant condicions ideals de tir parabòlic sobre un terra pla, determina a quina distància anirà a parar el projectil i quant temps trigarà en caure.

c) Llegeix un temps t mesurat en segons i escriu la longitud que ha de tenir un pèndol ideal per tal que el seu període d'oscil·lació sigui t.

d) Llegeix les condicions d'un compte d'estalvi: el capital expressat en euros, una taxa d'interès expressat en tant per cent i un nombre d'anys i, escriu el redit de compte expressat en euros.

3 Funcions

3.1 Definicions i ús

En el context de la programació, una funció és una seqüència de sentències que fan una tasca concreta i que identifiquem amb un nom. A `Python`, la sintaxi per definir una funció és la següent:

```
def NAME( PARAMETRES ):
    SENTENCIA 1
    SENTENCIA 2
    ...
```

Pots usar qualsevol nom per a les funcions que defineixis sempre que no usis una paraula clau de `Python`. Els paràmetres especifiquen quina informació, si és que cal, se li ha de proporcionar a la funció per tal d'obtenir el resultat esperat.

Dins la funció hi pot haver qualsevol nombre de sentències, però han d'estar indentades a partir de la línia **def**. Les definicions de funcions són la primera de diverses sentències que estudiarem que comparteixen el mateix patró:

1. Una capçalera, que comença amb una paraula clau i acaba amb dos punts.
2. Un cos consistent en una o més sentències `Python`, totes indentades al mateix nivell des de la capçalera.

En una definició de funció, la paraula clau de la capçalera és **def**, que es seguida pel nom de la funció i per una llista de paràmetres tancada entre parèntesis. La llista de paràmetres pot ser buida o pot contenir qualsevol nombre de paràmetres separats per comes. En qualsevol cas, sempre es requereixen els parèntesis.

El primer parell de funcions que escriurem no tenen paràmetres, així que el seu aspecte sintàctic és el següent:

```
def new_line():
    print # una sentencia print sense arguments imprimeix una linia en blanc
```

Aquesta funció s'anomena new_line. El parèntesis buit indica que no té paràmetres i el seu cos conté una sola sentència, que escriu un salt de línia (això és el fa **print** quan l'usem sense arguments).

Definir una funció no fa que aquesta s'executi. Per executar-la necessitem una crida a funció. La crida a una funció conté el nom de la funció a executar seguit d'una llista de valors, anomenats arguments, que s'assignen als paràmetres de la definició de funció. El nostre primer exemple té una llista de paràmetres buida, així que la crida a la funció no necessita arguments. Noteu però, que també són necessaris els parèntesis en cridar la funció:

```
print "Primera Linia."
new_line()
print "Segona Linia."
```

La sortida d'aquest programa és:

```
Primera Linia.

Segona Linia.
```

L'espai extra entre les dues línies és el resultat de la crida a la funció new_line(). Què hauríem de fer si volguéssim més espais entre les dues línies? Una possible solució seria cridar a la mateixa funció repetidament:

```
print "Primera Linia."
new_line()
new_line()
new_line()
print "Segona Linia."
```

O podríem escriure una nova funció anomenada three_lines que escrigui tres noves línies en blanc:

```
def three_lines():
    new_line()
    new_line()
    new_line()

print "Primera Linia."
three_lines()
print "Segona Linia."
```

Aquesta funció conté tres sentències, indentades usant quatre espais. Com que la següent sentència ja no es troba indentada, Python sap que no forma part del cos de la funció.

Aquesta funció ha de servir per adonar-te'n d'algunes coses:

- Es pot cridar la funció repetidament. De fet, és molt comú.
- Es pot cridar a una funció des del cos d'una altre funció; en aquest cas la funció three_lines() crida a new_line().

Hi ha moltes raons per les quals resulta útil usar funcions. Per exemple:

1. Crear una nova funció et permet donar un nom a un grup de sentències. Les funcions poden simplificar un programa amagant càlculs complicats darrera una senzilla crida i usant paraules en Anglès (o Català o Castellà) en comptes d'un codi arcà.
2. Crear una nova funció pot fer un programa més curt a base d'eliminar codi repetitiu. Per exemple, una manera més curta d'imprimir nou línies en blanc és cridar la funció three_lines() tres vegades.

Posant junts tots els fragments anteriors de codi de la secció anterior en un script anomenat tryme1.py, ens permet obtenir un programa amb el següent aspecte:

```
def new_line():
    print

def three_lines():
    new_line()
    new_line()
    new_line()

print "Primera Linia."
three_lines()
print "Segona Linia."
```

Aquest programa conté dues definicions de funció: new_line() i three_lines(). Les definicions de funció s'executen exactament igual que la resta de sentències, però el seu efecte és crear la nova funció. Les sentències que formen part del cos d'una funció no s'executen fins que no es crida la funció, i la definició de la funció no produeix cap sortida.

Com potser ja suposes, has de crear una funció abans de poder-la utilitzar. En altres paraules, s'ha d'executar la definició de la funció abans de la primera crida a aquesta.

3.2 Flux d'execució

flux d'execució (execution flow)

Per tal d'assegurar que una funció s'ha definit abans del seu primer ús, has de conèixer l'ordre en què les sentències s'executen. Aquest ordre s'anomena el *flux d'execució*.

L'execució sempre comença per la primera sentència del programa. Les sentències s'executen d'una en una, des del principi fins el final.

Les definicions de funcions no alteren el flux d'execució del programa. Recordeu que les sentències a dins del cos d'una funció no s'executen fins que es crida la funció.

Les crides a una funció actuen com un desviament en el flux d'execució. Quan es troba una crida, en comptes d'anar a la següent sentència, el flux salta cap a la primera línia del cos de la funció cridada, executa totes les sentències d'allí, i finalment retorna per continuar des del punt on havia desviat.

Dit així sembla senzill, però cal recordar que una funció en pot cridar una altra. Quan es trobi a mitja funció, el programa pot haver d'executar les sentències d'una altra funció. Però quan es trobi executant la nova funció, el programa pot haver d'executar encara una altra funció!

Afortunadament, Python és molt bo seguint la pista d'on es troba en tot moment, així que cada vegada que una funció es completa, el programa sap retornar al punt des del que havia saltat. Quan s'arriba al final del programa, s'ha acabat.

Què en podem treure de tot això que hem explicat? Quan hàgim de llegir un programa, no ho farem de dalt a baix. En comptes d'això, el llegirem seguint el flux d'execució. Els programes els escrivim com una sèrie de línies en un ordre determinat però el flux d'execució generalment no coincideix amb aquest ordre. Aprendre a copsar quin és el flux d'execució d'un programa que estem llegint és una habilitat fonamental.

3.3 Paràmetres, arguments i la sentència import

La majoria de funcions requereixen arguments, valors que controlen com fa la seva feina la funció. Per exemple, si vols trobar el valor absolut d'un nombre, has d'indicar d'alguna forma quin és el nombre. Python té una funció predefinida per calcular el valor absolut:

```
>>> abs(5)
5
>>> abs(−5)
5
```

En aquest exemple, els arguments per a la funció abs són 5 i -5.

Algunes funcions requereixen més d'un argument. Per exemple, la funció predefinida pow pren dos arguments, la base i l'exponent. A dins de la funció, els valor passats com a arguments s'assignen a una espècie de variables anomenades paràmetres.

```
>>> pow(2, 3)
8
>>> pow(7, 4)
2401
```

Una altra funció predefinida que pren més d'un argument és max:

```
>>> max(7, 11)
11
>>> max(4, 1, 17, 2, 12)
17
>>> max(3 * 11, 5**3, 512 - 9, 1024**0)
503
```

Es pot passar qualsevol nombre d'arguments a max, sempre separats per comes, i la funció retornarà el valor màxim d'entre els passats. Els arguments poden ser simples valors o expressions. A l'últim exemple es retorna 503, donat que és més gran que 33, 125 i 1.

A continuació un exemple de funció creada per l'usuari que té un paràmetre:

```
def print_twice(param):
    print param, param
```

Aquesta funció té un sol argument i l'assigna al paràmetre anomenat param. El valor del paràmetre (arribats a aquest punt no tenim ni idea de quin serà) s'imprimeix dues vegades, seguit d'un salt de línia. El nom param s'ha escollit per tal de reforçar la idea de què es tracta d'un paràmetre, però en general, és preferible usar noms pels paràmetres que reflecteixin la seva utilitat a la funció.

L'intèrpret de Python en mode interactiu ens proporciona una manera convenient de verificar les nostres funcions. Podem usar la sentència **import** per portar les funcions que hàgim definit en un script a la sessió de l'intèrpret. Per tal de veure com funciona això, assumirem que la funció print_twice està definida en un fitxer anomenat chap03.py. Ara podem provar-la interactivament important-la a la nostra sessió de l'intèrpret de Python en mode interactiu:

```
>>> from chap03 import *
>>> print_twice('Spam')
Spam Spam
>>> print_twice(5)
5 5
>>> print_twice(3.14159)
3.14159 3.14159
```

En una crida a funció, el valor de l'argument s'assigna al corresponent paràmetre de la definició de la funció. En efecte, quan es crida a print_twice('Spam') és com si tinguéssim param = 'Spam', quan es crida print_twice(5) tindríem param = 5 i quan es crida a print_twice(3.14159) el valor del paràmetre esdevindrà param = 3.14159.

Qualsevol argument que pugui ser escrit es pot passar a la funció print_twice. A la primera crida a funció l'argument és una cadena de caràcters, a la segona és un enter i a la tercera és un real.

Com en el cas de les funcions predefinides, també podem usar una expressió com a argument per a la funció print_twice:

```
>>> print_twice('Spam' * 4)
SpamSpamSpamSpam SpamSpamSpamSpam
```

'Spam'* 4 s'avalua primer a 'SpamSpamSpamSpam', i aleshores es passa com a argument a print_twice.

3.4 Composició

Com passa a les funcions matemàtiques, les funcions de `Python` es poden composar. Això vol dir usar el resultat de sortida d'una funció com a entrada per a una altra.

```
>>> print_twice(abs(−7))
7 7
>>> print_twice(max(3, 1, abs(−11), 7))
11 11
```

En el primer exemple, abs(−7) s'avalua a 7, que esdevé l'argument de print_twice. En el segon exemple tenim dos nivells de composició ja que abs(−11) s'avalua a 11, després max(3, 1, 11, 7) s'avalua a 11 i finalment s'escriu el resultat de print_twice(11).

També podem usar una variable com a argument:

```
>>> sval = 'Eric, the half a bee.'
>>> print_twice(sval)
Eric, the half a bee. Eric, the half a bee.
```

És important notar que el nom de la variable que hem passat com a argument (sval) no té res a veure amb el nom del paràmetre (param). De nou, és com si tinguéssim param = sval quan es crida a print_twice(sval). No és important com s'anomenava l'argument en la funció que fa la crida: dins de print_twice el seu nom és param.

3.5 Les variables i paràmetres són locals

locals (local)

Quan crees una variable dins d'una funció, aquesta només existeix en el cos de la funció, i no la pots fer servir fora d'aquesta. D'aquestes variables en direm *locals*. Per exemple:

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Aquesta funció necessita dos arguments, concatena els valors, i aleshores escriu el resultat dues vegades. Podem cridar aquesta funció amb dos cadenes:

```
>>> chant1 = "Pie Jesu domine, "
>>> chant2 = "Dona eis requiem."
>>> cat_twice(chant1, chant2)
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.
```

Quan s'acaba cat_twice, la variable cat es destrueix. Si provem d'escriure-la, obtenim un error:

```
>>> print cat
NameError: name 'cat' is not defined
```

Els paràmetres també són locals. Per exemple, a fora de la funció print_twice, no existeix res anomenat param. Si proves d'usar-lo, `Python` es queixarà.

3.6 Diagrames de pila

Per tal de seguir la pista de quines variables es poden usar a cada lloc, sovint és útil dibuixar un diagrama de pila. Com els diagrames d'estat, els diagrames de pila mostren el valor de cada variable, però també mostren la funció a la qual pertany cada variable.

Cada funció es representa per un marc. Un marc és una caixa amb el nom de la funció al costat i els paràmetres i variables de la funció a dintre.

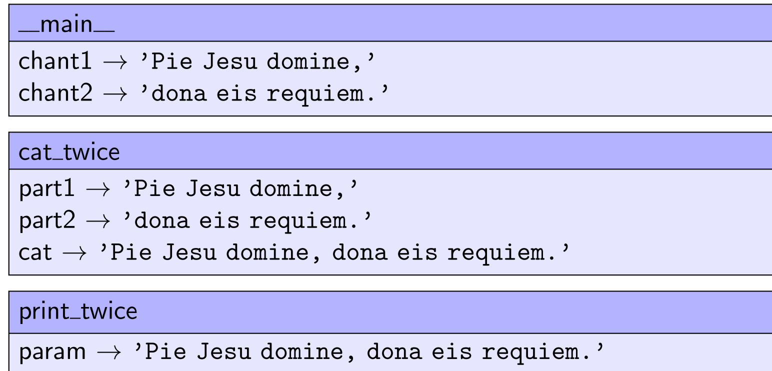


Figura 3.1: Diagrama de pila

L'ordre de la pila mostra el flux d'execució. print_twice l'ha cridat cat_twice, i cat_twice l'ha cridat __main__, que és un nom especial per a la funció principal. Quan crees una variable a fora de totes les funcions, aquesta pertany a la funció __main__.

Cada paràmetre es refereix al mateix valor que el seu corresponent argument. Així, part1 té el mateix valor que chant1, part2 té el mateix valor que chant2, i param té el mateix valor que cat.

Si s'esdevé algun error durant una crida a funció, Python imprimeix el nom de la funció i el nom de la funció que la ha cridat i, el nom de la funció que ha cridat aquesta i, així anar fent, fins arribar a la principal.

Per veure com funciona això, creem un script Python anomenat tryme2.py, que té aquest aspecte:

```
def print_twice(param):
    print param, param
    print cat

def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)

chant1 = "Pie Jesu domine, "
chant2 = "Dona eis requim."
cat_twice(chant1, chant2)
```

Hem afegit la sentència print_cat a dins de la funció print_twice, però cat no es troba definida aquí. L'execució d'aquest script produirà un missatge d'error com aquest:

```
Traceback (innermost last):
  File "tryme2.py", line 11, in <module>
    cat_twice(chant1, chant2)
  File "tryme2.py", line 7, in cat_twice
```

```
    print_twice(cat)
  File "tryme2.py", line 3, in print_twice
    print cat
NameError: global name 'cat' is not defined
```

Aquesta llista de funcions s'anomena *traceback* (determinació d'origen). Et dóna informació d'en quin fitxer del programa s'ha produit l'error, en quina línia i quines funcions s'estaven executant en el moment de l'error. També et mostra la línia de codi que ha causat l'error.

traça de la pila (stack trace)

Cal adonar-se de la similitud entre el traceback i el diagrama de pila. No és una coincidència. De fet, un altre nom habitual per al traceback és *traça de la pila*.

Exercicis

EXERCICI 3.1 Afegiu el cos a la següent funció per tal que escrigui n vegades la frase continguda en el paràmetre s.

```
    def digues_n_vegades(s, n):
        # ompliu el cos aqui
```

Deseu la funció en un fitxer que es digui `test_importar.py`. Ara, en una terminal inicieu l'intèrpret de `Python` en mode interactiu i comproveu que el comportament és el següent:

```
>>> from test_importar import *
>>> digues_n_vegades('Spam', 7)
SpamSpamSpamSpamSpamSpamSpam
```

EXERCICI 3.2 `Python` disposa d'una funció predefinida que ens permet saber el nombre de caràcters d'una cadena. Es tracta de la funció len(). L'expressió len('programar'), per exemple, s'avalua a 9.

Definiu la funció justificat_a_la_dreta que tingui com a paràmetre una cadena s i escrigui per pantalla el valor de la cadena s alineat per la dreta a la columna 70. Això pot fer-ho a base d'escriure prèviament els espais blancs necessaris.

Observeu com hauria de funcionar:

```
>>> justificat_a_la_dreta('hola')
                                                  hola
>>> justificat_a_la_dreta('emacs')
                                                 emacs
```

EXERCICI 3.3 Escriviu en un fitxer de nom `neteja.py` el següent script:

```
def nova_linia():
    print

def tres_linies():
    nova_linia()
    nova_linia()
    nova_linia()
```

A continuació afegiu al mateix fitxer la definició de la funció nou_linies(). La funció nou_linies() ha d'escriure nou línies en blanc i ho ha de fer cridant la funció tres_linies(). Per últim afegiu una nova funció que es digui netejar_terminal() que escrigui 25 línies en blanc. Definiu-la a base de cridar les vegades que sigui necessari les funcions anteriors. Finalment, al final del fitxer, feu un programa que netegi la terminal, escrigui la paraula "Hola", nou línies en blanc i la paraula "Adeu" usant les funcions definides prèviament.

Exercici 3.4 Quin error hi ha en el següent codi:

```
def test():
    assert = 'Hola'
    print assert + '2'
```

Exercici 3.5 En un fitxer de nom `celsius.py` definiu una funció que converteixi una temperatura expressada en graus Celsius a la mateixa temperatura expressada en graus Fahrenheit. Recordeu que la relació entre una i altra escala és $F = \frac{9}{5}C + 32$. Comproveu el seu funcionament usant el intèrpret de `Python` en mode interactiu.

Exercici 3.6 Quin dels següents identificadors de variables és incorrecte : spam, spam431, spam_43, 43spam, spAm ?

Exercici 3.7 És correcte el següent codi? Per què?

```
def hola():
    # Comentari hola

def adeu():
    # Comendari adeu
    print 'adeu'
```

Exercici 3.8 Una funció f pot tenir una variable local i un paràmetre amb el mateix nom? Per què?

Exercici 3.9 Considereu el següent script de `Python`:

```
def resultat(a):
    print "El resultat es: ", a

def multn(n):
    resultat(n*1)
    resultat(n*2)
    resultat(n*3)

multn(4)
multn("A")
```

Numereu les línies de l'script anterior considerant que la primera correspon a l'u. A continuació simuleu l'execució de l'script i aneu anotant la seqüència de números corresponents a les línies que es van executant. El resultat que obteniu descriu el flux d'execució del programa.

Exercici 3.10 Sense usar l'intèrpret de `Python` raoneu què escriu aquest programa:

```
def z2(x):
    z = x * x
    print z

def z3(x):
    z = 2 * x
    z2(z)
    z2(max(30,z))

z = 10
z3(z)
```

EXERCICI 3.11 Suposeu que teniu una funció `f()` que té un paràmetre `p`. Suposeu també que l'heu cridat una vegada i la funció ha fet la tasca que li corresponia. Podeu garantir que si la crideu de nou amb un argument diferent de l'anterior en cap cas es produirà un error en temps d'execució?

Podríeu posar un exemple que avali la vostra resposta?

EXERCICI 3.12 Dissenyeu una funció amb un paràmetre enter n que escrigui per pantalla `"Si"` si `n` és parell i `"No"` en cas que `n` sigui senar.

```
def parell(n):
    ...
```

4 Condicionals

4.1 L'operador mòdul

L'operador mòdul s'aplica als enters o les expressions enteres i calcula el residu resultant de dividir el primer operand entre el segon. A Python, l'operador de mòdul és el signe "tant per cent" (%). La sintaxi és la mateixa que per la resta d'operadors:

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

Així el resultat de dividir 7 entre 3 és 2 i ens en sobra 1.

Per a la nostra sorpresa, l'operador mòdul resulta ser força útil. Per exemple, et pot permetre comprovar si un nombre és divisible per un altre. Si x % y és zero, aleshores x és divisible entre y.

També et pot servir per extreure el dígit o dígits de més a la dreta d'un número. Per exemple, l'expressió x % 10 retorna el dígit de més a la dreta d'x (en base 10). De manera similar, x % 100 retorna els dos últims dígits.

4.2 Valors booleans i expressions

El tipus de Python que serveix per emmagatzemar els valors cert, True, i fals, False, s'anomena booleà (bool). S'anomena així en reconeixement al matemàtic anglès, George Boole. Boole va definir i estudiar l'àlgebra booleana, que és la base per a tota l'aritmètica dels computadors actuals.

Només hi ha dos valors booleans possibles: True i False. Les majúscules són importants!. Fixeu-vos en el següent exemple:

```
>>> type(True)
<type 'bool'>
>>> type(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

Una expressió booleana és una expressió que s'avalua a un valor booleà. L'operador d'igualtat, ==, compara dos valors i retorna un valor booleà:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

A la primera sentència, els dos operands són iguals, així doncs l'expresió s'avalua a True; a la segona sentència, 5 és diferents de 6, així doncs l'expressió s'avalua a False.

L'operador == és un dels operadors de comparació. Els altres són:

x != y	x és diferent d'y
x > y	x es major que y
x < y	x es menor que y
x >= y	x es major o igual que y
x <= y	x es menor o igual que y

Tot i que aquestes operacions segurament et sonen, els símbols que usa `Python` són diferents als símbols matemàtics. Un error molt comú és usar el signe igual (=) en comptes del doble igual (==). Recorda que = és l'operador d'assignació i == és un operador de comparació. Cal recordar també que no existeix cap operador semblant a =< o =>.

4.3 Operadors lògics o booleans

Hi ha tres operadors lògics: **and**, **or** i **not** que corresponen a les formes catalanes *i*, *o* i *no*. La semàntica —és a dir el significat— d'aquests operadors és similar al seu significat en català. Per exemple, l'expressió x > 0 **and** x < 10 només és certa si x és més gran que 0 i més petit que 10.

D'altra banda, l'expressió n % 2 == 0 **or** n % 3 == 0 només és certa si alguna de les condicions és certa, això és, si el n és divisible entre 2 o entre 3.

Finalment, l'operador **not** nega una expressió booleana, així l'expressió **not**(x > y) val cert si (x > y) val fals, és a dir, si x és menor o igual que y.

operadors booleans (empty)

Dels operadors lògics també en diem *operadors booleans* i sovint es descriuen usant *taules de veritat*, que tabulen el resultat d'operar el valor a amb el b per a totes les combinacions possibles d'a i b. En el cas dels tres operadors booleans que hem vist, les taules corresponents serien aquestes:

taules de veritat (empty)

a	b	**not** b	a **or** b	a **and** b
True	True	False	True	True
True	False	True	True	False
False	True	—	True	False
False	False	—	False	False

Concorden amb el que us dictava la vostra intuïció?

4.4 Execució condicional

Per tal de poder escriure programes útils, cal poder comprovar condicions i canviar el comportament del programa d'acord amb el resultat d'aquestes comprovacions. Les sentències condicionals ofereixen aquesta possibilitat. La forma més senzilla és la sentència **if** és la següent:

```
if x > 0:
    print "x es positiu"
```

L'expressió booleana que segueix la paraula clau **if** s'anomena condició. Si aquesta és certa, aleshores s'executen les sentències indentades. Si no és certa, aleshores no passa res.

La sintaxi d'una sentència **if** té el següent aspecte:

```
if EXPRESSIO BOOLEANA:
    SENTENCIES
```

Com en la definició de funcions del capítol anterior i altres sentències compostes, la sentència **if** consisteix en una capçalera i un cos. La capçalera comença per la paraula clau **if** seguida d'una expressió booleana i acabada per dos punts (:).

bloc (block)

cos (body)

Les sentències indentades que segueixen formen un *bloc*. La primera sentència no indentada marca el final del bloc. El bloc de sentències que trobem dins d'una sentència composta com **if** s'anomena el *cos* de la sentència.

En el cas de la sentència **if**, si l'expressió booleana s'avalua a cert cadascuna de les sentències de dins del cos s'executa seqüèncialment seguint l'ordre en que s'han escrit. D'altra banda, si l'expressió booleana s'avalua a fals, no s'executa cap sentència del cos. No hi ha cap límit en el nombre de sentències que poden aparèixer al cos d'una sentència **if**, però com a mínim n'hi ha d'haver una. En ocasions, és útil tenir un cos sense sentències, per exemple per que vols escriure el cos en un altre moment. En aquest cas, es pot usar la sentència **pass**, que no fa absolutament res, per omplir el cos:

```
if True: # Aixo sempre es cert
    pass # per tant aixo sempre s'executa, pero no    # fa res
```

4.5 Execució alternativa

Una segona forma de la sentència **if** és l'execució alternativa, en la que hi ha dues possibilitats i la condició determina quina d'elles s'executa. La sintaxi té el següent aspecte:

```
if x % 2 == 0:
    print x, "es parell"
else:
    print x, "es senar"
```

Si el residu de la divisió d'x entre 2 és 0, aleshores sabem que x és parell, i conseqüentment el programa mostra un missatge apropiat a aquest efecte. Si la condició és falsa, s'executa el segon bloc de sentències. Donat que la condició ha de ser certa o falsa, s'executarà exactament una de les alternatives. Les alternatives s'anomenen branques, perquè són branques en el flux d'execució.

Com a parèntesi, només dir que si necessites calcular la paritat (si un nombre és parell o senar) d'un nombre molt sovint, pots embolicar aquest codi en una funció:

```
def print_parity(x):
    if x % 2 == 0:
        print x, "es parell"
    else:
        print x, "es senar"
```

Per qualsevol valor d'x, print_parity escriu el missatge apropiat. Quan crides aquesta funció, pots proporcionar qualsevol expressió entera com a argument.

```
>>> print_parity(17)
17 es senar.
>>> y = 41
>>> print_parity(y+1)
42 es parell.
```

4.6 Condicionals encadenats

A vegades existeixen més de dues possibilitats i necessitem més de dues branques. Per expressar càlculs com aquests podem usar condicionals encadenats:

```
if x < y:
    print x, "es mes petit que", y
elif x > y:
    print x, "es mes gran que", y
else:
    print x, "i", y, "son iguals"
```

elif és una abreviació d'**else if**. De nou, només s'executarà exactament una branca. No hi ha límit al nombre de sentències **elif** però només és permet una única (i opcional) sentència **else**, i ha de ser la última branca de la sentència:

```
if choice == 'a':
    function_a()
elif choice == 'b':
    function_b()
elif choice == 'c':
    function_c()
else:
    print "Seleccio incorrecta."
```

Cada condició es comprova seguint l'ordre en que s'han escrit. Si la primera és falsa, es comprova la següent, i així successivament. Si una d'elles és certa, s'executa la branca corresponent, i aleshores s'acaba la sentència. Encara que més d'una condició sigui certa, només s'executa la primera branca certa que es troba..

4.7 Condicionals niuats

Un condicional pot estar niuat dins d'un altre. Podríem haver escrit l'exemple de la tricotomía de la següent manera:

```
if x == y:
    print x, "i", y, "son iguals"
else:
    if x < y:
        print x, "es mes petit que", y
    else:
        print x, "es mes gran que", y
```

El condicional extern conté dues branques. La primera branca conté una senzilla sentència d'escriptura. La segona sentència conté una altra sentència **if**, que també té dues branques. Aquestes branques són, ambdues, sentències d'escriptura, tot i que haurien pogut ser també sentències condicionals.

Tot i que la indentació de les sentències ens mostra clarament l'estructura, els condicionals niuats resulten difícils de llegir ràpidament. En general, és bona idea evitar-los sempre que sigui possible.

Els operadors lògics sovint ens proporcionen una manera de simplificar els condicionals niuats. Per exemple, el següent codi es pot reescriure usant un únic condicional:

```
if 0 < x:
    if x < 10:
        print "x es un unic digit positiu."
```

La sentència **print** només s'executa si es compleixen ambdues condicions simultàniament, així doncs podem usar l'operador **and** i reescriure-ho com:

```
if 0 < x and x < 10:
    print "x es un unic digit positiu."
```

Aquesta mena d'expressions booleanes són tant comunes que `Python` proporciona una sintaxi específica, molt semblant a la notació que s'usa a les matemàtiques en aquests casos:

```
if 0 < x < 10:
    print "x es un unic digit positiu."
```

Aquesta condició és semànticament idèntica a l'expressió booleana que usava l'**and** i al condicional niuat anterior.

4.8 La sentència "return"

La sentència **return** permet acabar l'execució d'una funció abans d'arribar al seu final. Una raó per usar-la pot ser forçar l'acabament en detectar una condició d'error:

```
def print_square_root(x):
    if x <= 0:
        print "Nomes nombres positius, si us plau."
        return
    result = x**0.5
    print "L'arrel quadrada de", x, "es", result
```

La funció `print_square_root` té un paràmetre anomenat `x`. La primera cosa a fer és comprovar si `x` és menor o igual que 0, en qual cas es mostra el missatge d'error i s'usa la sentència **return** per acabar la funció. En executar una sentència **return**, el flux d'execució retorna immediatament a qui ha efectuat la crida, i la resta de sentències de la funció no s'executen.

4.9 Entrades de teclat

A l'apartat 2.9, Entrada, vàrem parlar de les funcions predefinides de `Python` que permetien la lectura de dades des del teclat: `raw_input` i `input`. Anem a parlar-ne de nou però amb més profunditat.

Quan es crida a qualsevol d'aquestes dues funcions, el programa s'atura i queda a l'espera de què l'usuari teclegi alguna cosa. Quan l'usuari prem la tecla Return o Intro, el programa es recupera i `raw_input` retorna allò que l'usuari hagi escrit com a una cadena de caracters:

```
>>> my_input = raw_input()
A que esperes?
>>> print my_input
A que esperes?
```

prompt (prompt)

Abans de cridar a raw_input, és una bona idea mostrar un missatge explicant a l'usuari què ha d'escriure. Aquest missatge s'anomena *prompt*. Podem proporcionar el prompt com a argument per a la funció raw_input:

```
>>> name = raw_input("Quin... es el teu nom? ")
Quin... es el teu nom? Artus, Rei dels Bretons!
>>> print name
Artus, Rei dels Bretons!
```

Cal adonar-se que el prompt és una cadena, així que s'ha d'escriure entre cometes.

Si esperem que la resposta sigui un enter podem usar la funció input, que avalua la resposta com si fos una expressió de Python:

```
prompt = "Quina... es la velocitat de vol d'una oreneta?\n"
speed = input(prompt)
```

Si l'usuari tecleja una cadena de dígits, aquesta es convertirà en un enter i s'assignarà a la variable speed. Desafortunadament, si l'usuari tecleja caràcters que no conformen una expressió vàlida de Python, el programa deixarà de funcionar:

```
>>> speed = input(prompt)
Quina... es la velocitat de vol d'una oreneta?
A quina us referiu, a l'oreneta Africana o l'Europea?
...
SyntaxError: invalid syntax
```

A l'exemple anterior, si l'usuari hagués fet servir les cometes per convertir la resposta en una expressió de Python vàlida, no hauria provocat cap error:

```
>>> speed = input(prompt)
Quina... es la velocitat de vol d'una oreneta?
"A quina us referiu, a l'oreneta Africana o l'Europea?"
>>> speed
'A quina us referiu, a l'oreneta Africana o l'Europea?'
>>>
```

Per tal d'evitar aquest tipus d'errors, és bona idea usar raw_input per obtenir una cadena i després usar les comandes de conversió per convertir-la en un valor del tipus que ens interessi.

4.10 Conversió de tipus

Cada tipus de Python té associada una funció predefinida que permet convertir valors d'altres tipus en valors d'aquest. La funció int(), per exemple, pren qualsevol valor com a paràmetre i el prova de convertir en un enter. En cas que no sigui possible es queixa:

```
>>> int("32")
32
>>> int("Hola")
ValueError: invalid literal for int() with base 10: 'Hola'
```

int() també pot convertir valors reals en enters, però cal recordar que truncarà la part decimal:

```
>>> int(−2.3)
−2
>>> int(3.99999)
3
>>> int("42")
42
>>> int(1.0)
1
```

La funció float() converteix a valors reals. És corrent aplicar-la a enters i a cadenes de caràcters:

```
>>> float(32)
32.0
>>> float("3.14159")
3.14159
>>> float(1)
1.0
```

Pot resultar estrany que `Python` faci distinció entre el nombre enter 1 i el nombre real 1.0. Representen el mateix nombre, però pertanyen a tipus diferents. La causa d'aquesta diferenciació s'ha de buscar en el fet que, internament, en el computador enters i reals es representen de formes diferents.

La funció str() converteix qualsevol argument donat al tipus cadena de caràcters:

```
>>> str(32)
'32'
>>> str(3.14149)
'3.14149'
>>> str(True)
'True'
>>> str(true)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'true' is not defined
```

str() funcionarà per qualsevol valor i el convertirà en una cadena. Com ja s'ha comentat anteriorment, True és un valor booleà, mentre que true no ho és.

Pels valors booleans, la conversió és especialment interessant:

```
>>> bool(1)
True
>>> bool(0)
False
>>> bool("Ni!")
True
>>> bool("")
False
>>> bool(3.14159)
True
>>> bool(0.0)
False
```

Python pot convertir valors de qualsevol tipus a booleans. Pels tipus numèrics com els enters i els reals, els valors zero es converteixen a False i els valors que no siguin zero a True. Per les cadenes, la cadena buida es converteix en False i la resta en True.

Exercicis

tautologia (empty)

Exercici 4.1 Una *tautologia* és una expressió booleana tal que, siguin quins siguin els valors de les dades, sempre s'avalua a True. Assumint que les variables a, b i c contenen enters, digueu quines de les expressions següents són tautologies:

a) a > b

b) a > 0 **or** a <= 0

c) a < b **or** a > b

d) True **and** bool(a)

e) c < 3 **or** **not**(c < 3)

f) **not**((2 < a < 5) **and** (a < 0))

Exercici 4.2 Els operands de l'operador mòdul són valors enters. En aquest capítol només s'ha explicat què calcula quan les dades són enters positius. Investiga, amb l'ajuda de l'intèrpret en mode interactiu, com es comporta aquest operador quan un o dos dels seus operands són negatius. Escriu una regla que defineixi exactament el seu significat sigui quin sigui el signe dels seus operands.

Exercici 4.3 Escriviu un petit programa que mostri un menú simple d'una màquina expenedora, demani a l'usuari que introdueixi el número corresponent a l'opció que desitja, i escrigui l'acció corresponent o bé escrigui un missatge d'error indicant que l'opció no és vàlida.

Observeu els següents exemples d'ús:

```
1. Amanida vegetal
2. Pasta amb salsa bolonyesa
3. Plat especial del dia

Introdueix la teva tria: 2
Pasta amb salsa bolonyesa a punt!
```

```
1. Amanida vegetal
2. Pasta amb salsa bolonyesa
3. Plat especial del dia

Introdueix la teva tria: 5
Opcio incorrecta
```

Exercici 4.4 Simplifiqueu els següents condicionals niuats:

a)
```
if x < 0:
    if x != 0:
        print "ok"
```

b)
```
if 0 < a < 10:
    if a > 3:
        print "ok"
```

c)
```
if m > 0 or x < 0:
    if nom == "Marta":
        print "ok"
```

d)
```
if m > 0 or x < 0:
    if d > 12:
        if m > 12:
            print "ok"
```

Exercici 4.5 Escriviu un programa que calculi les arrels d'una eqüació de segon grau i indiqui si tenia dues solucions reals, una solució real doble o cap solució real.

Exercici 4.6 Demostreu que es compleixen les següents igualtats entre expressions booleanes tot assumint que les variables a, b i c sempre contenen valors booleans. Per fer-ho, podeu tabular quan val cada expressió per totes les combinacions possibles de valors de les variables.

a) a **or** b **and** True ≡ a **or** b

b) False **and** b **and** c ≡ False

c) a **and** (b **or** c) ≡ (a **and** b) **or** (a **and** c)

Exercici 4.7 Què fa el següent programa? Primer, determineu-ho sense usar l'intèrpret, després comproveu la vostra solució amb l'ajuda de l'interpret.

```
a=input("Primer nombre: ")
b=input("Segon nombre: ")
c=input("Tercer nombre: ")
if a>=b and a>=c:
    m=a
if b>=a and b>=c:
    m=b
if c>=a and c>=b:
    m=c
print m
```

Exercici 4.8 Quin serà el resultat de convertir els següents valors al tipus booleà usant bool? Primer, determineu-ho sense usar l'intèrpret, després comproveu les solucions amb l'interpret.

a) 12.8

b) "True"

c) "False"

d) ""

e) −23

f) 0

g) "0"

Exercici 4.9 Calculeu el valor de les següents expressions. Primer feu-ho sense l'ajuda del computador i després, usant l'intèrpret en mode interactiu, comproveu que ho heu fet correctament. Si us heu equivocat, aprofiteu per esbrinar què no havíeu entès.

a) int(2.34 + abs(−2)) / 2

b) 3 < int(3.89 + 12.8) **or** True

c) 2 < 4 < 10

d) abs(float(4) − 3.4)

Exercici 4.10 Escriviu un programa que, donats 5 nombres enters, determini quin dels quatre darrers nombres és més proper al primer.

Exemple d'execució:

```
Escriu enter: 2
Escriu enter: 6
Escriu enter: 4
Escriu enter: 1
Escriu enter: 10

El nombre 1 es el mes proper al 2
```

5 Funcions fructíferes

5.1 Valors de retorn

Les funcions predefinides que hem usat fins al moment, com poden ser abs, pow i max, produeixen resultats. La crida a cadascuna d'aquestes funcions genera un valor, que normalment assignem a una variable o que usem com a part d'una expressió. Per exemple:

```
biggest = max(3, 7, 2, 5)
x = abs(3 - 11) + 10
```

Fins ara, però, cap de les funcions que hem dissenyat nosaltres ha retornat un valor.

En aquest capítol, escriurem funcions que retornin valors. Les anomenarem «funcions fructíferes», ja que no tenim un nom millor [1]. El primer exemple és area, una funció que retorna l'àrea d'un cercle donat el radi:

```
def area(radi):
    temp = 3.14159 * radi**2
    return temp
```

Anteriorment hem vist la sentència **return**, però en una funció fructífera la sentència **return** inclou un valor de retorn. Aquesta sentència significa: retorna immediatament d'aquesta funció i utilitza la següent expressió com a valor de retorn. L'expressió proporcionada pot ser arbitràriament complicada, així que podríem haver escrit aquesta funció de manera més concisa fent:

```
def area(radi):
    return 3.14159 * radi**2
```

temporary variables (empty)

Aquesta versió potser sembla més interessant. Això no obstant, a vegades l'ús de les *temporary variables*variables temporals com temp sovint fan la feina de depurar més senzilla.

En algunes circumstàncies és útil tenir més d'una sentència de retorn, una a cada branca d'un condicional per exemple. Ja hem vist en capítols anteriors la funció predefinida abs, però ara definirem la nostra pròpia versió:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

[1] En l'argot informàtic habitual, les "funcions que no retornen valors" s'anomenen *accions* i el concepte *funció* s'aplica sistemàticament a les "funcions que si retornen valors". Això no obstant, en l'àmbit de Python s'acostuma a parlar únicament de funcions. Per aquesta raó hem mantingut la denominació *sui generis* de "funcions fructíferes" com a forma de distingir unes d'altres.

Donat que les sentències **return** es troben en una sentència condicional, només se n'executarà una. Tan aviat com s'executi un **return**, la funció acabarà.

Una altra manera d'escriure la funció anterior és oblidar-se de l'**else** i seguir la condició de l'**if** amb la segona sentència **return**.

```
def absolute_value(x):
    if x < 0:
        return -x
    return x
```

És interessant pensar una mica en aquesta solució fins quedar convençut de que funciona de la mateixa manera que la primera proposta.

codi mort (dead code)

El codi que apareix després d'una sentència **return**, o a qualsevol altre lloc on el flux d'execució no arribarà mai, s'anomena *codi mort*.

En una funció fructífera és bona idea assegurar-se que tots els possibles camins que pugui seguir el programa acaben arribant a una sentència **return**. La següent versió d'absolute_values, per exemple, no compleix aquest criteri:

```
def absolute_value(x):
    if x < 0:
        return -x
    elif x > 0:
        return x
```

Aquesta versió no és correcta. Si s'esdevé que x pren per valor 0, cap condició és certa i la funció acaba sense executar una sentència **return**. En aquest cas, el valor de retorn és un valor especial anomenat None:

```
>>> print absolute_value(0)
None
```

None és l'únic valor d'un tipus anomenat Nonetype:

```
>>> type(None)
```

Qualsevol funció de `Python` que acabi sense executar una sentència **return** retornarà el valor None.

5.2 Desenvolupament de programes

Arribats a aquest punt, hauries de ser capaç de mirar funcions completes i dir què fan. També, si has anat fent els exercicis, hauràs escrit petites funcions. A mida que vagis escrivint funcions més llargues, hauràs començat a trobar-te amb més problemes, especialment amb els errors semàntics i en temps d'execució.

Per tal de fer front als programes cada vegada més complexos, et suggerirem una tècnica de treball anomenada desenvolupament incremental. L'objectiu del desenvolupament incremental és millorar l'eficàcia quan dissenyem programari. La tècnica consisteix en desenvolupar programari a base d'anar alternant l'addició de petits fragments de codi i comprovant que funcionen correctament. Després d'haver anat afegint i comprovant molts d'aquests petits fragments acabarem tenint el programa que ens interessava.

Com a exemple, suposa que vols esbrinar la distància entre dos punts, donats per les coordenades (x_1, y_1) i (x_2, y_2). Pel teorema de Pitàgores, sabem que la distància d és:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

El primer pas és considerar quin aspecte tindria una funció distancia en Python. En altres paraules, quines són les entrades (paràmetres) i quina ha de ser la sortida (valor de retorn)?

En aquest cas, els dos punts són les entrades, que podem representar usant quatre paràmetres. El valor de retorn és la distància, què és un valor de tipus real.

Ja podem escriure un esbós de la funció:

```
def distancia(x1, y1, x2, y2):
    return 0.0
```

Òbviament, aquesta funció no calcula distàncies: sempre retorna zero. Però és sintàcticament correcte, i funcionarà, la qual cosa vol dir que la podem verificar abans de fer-la més complicada.

Per provar aquesta nova funció, la cridem amb uns valors de prova:

```
>>> distancia(1, 2, 4, 6)
0.0
```

Hem escollit aquests valors de tal manera que la distància horitzontal sigui igual a 3 i la distància vertical sigui igual a 4; així, el resultat serà 5 (la hipotenusa del triangle). Quan s'està verificant una funció, és molt útil conèixer quin ha de ser el resultat correcte!

Arribats a aquest punt, hem comprovat que la funció és sintàcticament correcte, i ja podem començar a afegir-hi noves línies de codi. Després de cada canvi incremental, verifiquem la funció de nou. Si s'esdevé un error a qualsevol punt, sabrem on ha de ser: a la darrera línia afegida.

Un primer pas lògic en el càlcul és trobar la diferència entre $(x_2 - x_1)$ i $(y_2 - y_1)$. Emmagatzemarem aquests valor a dues variables temporals anomenades dx i dy i les escriurem.

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print "dx is", dx
    print "dy is", dy
    return 0.0
```

Si la funció està funcionant, la sortida hauria de ser 3 i 4. Si és així, sabem que la funció està rebent els paràmetres i duent a terme el primer càlcul correctament. Si no és així, només hi ha unes quantes línies a comprovar.

Després duem a terme la suma dels quadrats de dx i dy:

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print "dsquared is: ", dsquared
    return 0.0
```

Noteu que hem eliminat la sentència **print** que havíem escrit al pas anterior.[2]

De nou, executaríem el programa en aquest punt i comprovaríem la sortida. Hauria de ser 25. Finalment, usant l'exponent real 0.5 per tal de trobar l'arrel quadrada, calculem i retornem el resultat:

```
def distancia(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = dsquared**0.5
    return result
```

Si funciona correctament, ja hem acabat. Altrament, potser t'interessarà escriure el valor de result abans de la sentència de retorn.

Quan comencis a aprendre aquesta metodologia incremental de treball, hauries d'afegir una o dues línies de codi cada vegada que vas ampliant el teu programa. A mida que vagis adquirint més experiència, et trobaràs a tu mateix escrivint i depurant trossos més grans. De qualsevol manera, el procés de desenvolupament incremental et pot estalviar molt de temps de depuració.

Els aspectes claus d'aquest procés són els següents:

1. Comença amb un programa que funcioni i vés fent petits afegits incrementals. A qualsevol punt, si hi ha un error, sabràs exactament on es troba.
2. Utilitza variables temporals per capturar valors intermedis per tal de què els puguis escriure i verificar.
3. Una vegada el programa funcioni correctament, pot ser que vulguis eliminar alguna variable sobrera o consolidar múltiples sentències en expressions compostes. Fes-ho només si el programa no esdevé més difícil de llegir.

5.3 Composició

Tal i com pot ser que ja sospitis, es pot cridar una funció des de dins d'una altra. Aquesta tècnica s'anomena composició.

Com a exemple, escriurem una funció que pren dos punts, el centre d'un cercle i un punt del perímetre, i calcula l'àrea del cercle.

Assumim que el centre del cercle es transfereix a través dels paràmetres xc i yc, i el punt del perímetre a través de xp i yp. El primer pas consisteix a trobar el radi del cercle, què és la distància entre els dos punts. Afortunadament, acabem d'escriure una funció, distancia, que fa justament això, així que ara tot el que hem de fer és usar-la:

```
radius = distancia(xc, yc, xp, yp)
```

El segon pas consisteix a trobar l'àrea d'un cercle donat aquest radi, i retornar-la. De nou usarem una de les nostres funcions prèvies:

```
result = area(radius)
return result
```

Empaquetant tot això en una funció, obtenim:

[2] Aquest tipus de codi els angloparlants l'anomenen *scaffolding* (bastida), ja que és útil per construir el programa però no forma part del producte final.

```
def area2(xc, yc, xp, yp):
    radius = distancia(xc, yc, xp, yp)
    result = area(radius)
    return result
```

Hem anomenat aquesta funció area2 per tal de distingir-la de la funció area que hem definit anteriorment. Dins d'un mòdul només pot haver-hi una funció amb un nom donat.

Les variables temporals radius i result són útils pel desenvolupament i la depuració, però una vegada el programa ja funciona, podem fer-lo més concís composant les crides a funció:

```
def area2(xc, yc, xp, yp):
    return area(distancia(xc, yc, xp, yp))
```

5.4 Funcions booleanes

Les funcions poden retornar valors booleans, la qual cosa sovint és convenient per amagar comprovacions complicades a dins de funcions. Per exemple:

```
def es_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

El nom d'aquesta funció és es_divisible. És habitual donar a les funcions booleanes que sonin a preguntes de tipus si/no. es_divisible retorna True o False per indicar si x és o no és divisible entre y.

Podem fer la funció més concisa aprofitant-nos del fet que la condició de la sentència **if** és una expressió booleana per si mateixa. Podem retornar-la directament, evitant-nos la sentència **if** sencera:

```
def es_divisible(x, y):
    return x % y == 0
```

Aquesta nova sessió mostra la nova funció en acció:

```
>>> es_divisible(6, 4)
False
>>> es_divisible(6, 3)
True
```

Les funcions booleanes sovint s'usen en sentències condicionals:

```
if es_divisible(x, y):
    print "x es divisible entre y"
else:
    print "x no es divisible entre y"
```

Tot i que pot ser temptador escriure alguna cosa com ara:

```
if es_divisible(x, y) == True:
```

sempre cal evitar-ho. Comparar el resultat d'una funció booleana amb True o False és generalment una mala opció.

5.5 El tipus "function"

Una funció és en realitat un altre tipus de `Python`, que s'afegeix als que ja coneixíem: `int`, `float`, `str`, `bool` i `NoneType`.

```
>>> def func():
... return "function func was called..."
...
>>> type(func)
<type 'function'>
>>>
```

Tal i com passa amb els altres tipus, les funcions es poden passar com a arguments a d'altres funcions. Observeu el següent exemple:

```
def f(n):
    return 3*n - 6

def g(n):
    return 5*n + 2

def h(n):
    return -2*n + 17

def doto(value, func):
    return func(value)

print doto(7, f)
print doto(7, g)
print doto(7, h)
```

`doto` es crida tres vegades. A cada crida:

- l'argument corresponent al paràmetre `value` és 7.
- l'argument corresponent al paràmetre `func` són les funcions `f`, `g` i `h` respectivament.

La sortida d'aquest programa és:

```
15
37
3
```

Aquest exemple és una mica forçat, però més endavant veurem situacions on és molt útil passar una funció com a paràmetre a una altra funció.

5.6 Programar amb estil

La facilitat de lectura és molt important pels programadors ja que els programes s'escriuen una vegada però es llegeixen i modifiquen molt més sovint. Tots els exemples de codi d'aquests capítols són consistents amb el *Python Enhancement Proposal 8 (PEP 8)*, una guia d'estil desenvolupada per la comunitat de `Python`.

Podrem dir més coses sobre l'estil dels nostres programes a mida que aquests es facin més complexos, però hi ha alguns criteris que ara ja seran útils:

- Useu 4 espais per a la indentació.
- Les sentències **import** han d'anar al principi del fitxer.
- Separeu les definicions de funció amb dues línies en blanc.
- Mantingueu les definicions de funció juntes.
- Mantingueu les sentències d'alt nivell, incloses les crides a funció, juntes al final del programa. És el que sovint en diem el "programa principal".

5.7 Triples cometes

A més de les cadenes delimitades per cometes i les dobles cometes que vàrem veure a l'apartat de valors i tipus, Python també té les triples cometes:

```
>>> type("""Aquesta es una cadena amb triples cometes usant 3 dobles cometes.""")
<type 'str'>
>>> type('''Aquesta es una cadena amb triples cometes usant 3 cometes simples.''')
<type 'str'>
>>>
```

Les triples cometes poden contenir dintre seu les cometes simples i les dobles cometes sense que això provoqui confusions:

```
>>> print '''"Oh no", va exclamar, "La bicicleta d'en Ben esta trencada!"'''
"Oh no", va exclamar, "La bicicleta d'en Ben esta trencada!"
>>>
```

Finalment, les triples cometes poden contenir més d'una línia:

```
>>> missatge = """Aquest missatge
... contindra diverses
... linies."""
>>> print missatge
Aquest missatge
contindra diverses
linies.
>>>
```

5.8 Prova de components amb "doctest"

prova de components (unit testing)

Avui en dia, en el desenvolupament de programes, es considera una bona praxis usar un sistema automàtic de verificació.

Les eines de *prova de components* possibiliten verificar de manera automàtica que les diferents peces o components individuals que formen un programa, com ara les funcions fan, fins un cert punt, el que cal. Això permet, per exemple, modificar la implementació d'una funció *a posteriori* i comprovar ràpidament, emprant aquestes eines, que la nova implementació continua fent allò que s'esperava d'ella.

joc de proves (empty)

El funcionament és senzill d'entendre. S'associa a cada component —en el nostre cas a cada funció— un conjunt de proves que anomenem *joc de proves*. Cada prova consisteix en un càlcul on intervé la funció juntament amb el resultat que s'ha d'obtenir. Després, de forma automàtica, podem comprovar que la funció o component satisfà totes les proves del joc de proves.

Python té un mòdul predefinit anomenat doctest que facilita la prova de components. Crear un joc de proves per a una funció és senzill. Només cal escriure entre triples cometes i just després de la capçalera de la funció un conjunt de proves que la funció ha de satisfer. El format d'aquesta cadena recorda a una sessió interactiva de l'intèrpret de Python. Aquesta cadena que conté un doctest es coneix amb el nom de *cadena de documentació* ja que, a banda d'indicar les proves que ha de superar la funció, també ajuda a documentar-la.

cadena de documentació (docstring)

El mòdul doctest automàticament executarà qualsevol sentència que comenci per >>> en aquesta cadena i compararà el resultat amb la següent línia de la cadena.

Per veure com funciona, escriviu el següent programa en el fitxer anomenat myfunctions.py:

```
def is_divisible_by_2_or_5(n):
    """
     >>> is_divisible_by_2_or_5(8)
     True
    """
```

Per passar el doctest i comprovar si el resultat de la prova és positiu o negatiu, des del terminal escriviu la següent comanda:

```
$ python -m doctest myfunctions.py
```

Obtindreu el següent resultat:

```
**********************************************************************
File "myfunctions.py", line 3, in myfunctions.is_divisible_by_2_or_5
Failed example:
    is_divisible_by_2_or_5(8)
Expected:
    True
Got nothing
**********************************************************************
1 items had failures:
   1 of 1 in myfunctions.is_divisible_by_2_or_5
***Test Failed*** 1 failures.
```

Això és un exemple de joc de proves que conté una prova amb un resultat negatiu. El resultat de passar el doctest diu:

> Segons el doctest, si crideu is_divisible_by_2_or_5(8) el resultat hauria de ser True. Això no obstant, s'ha fet la crida i la funció no ha retornat res. Per tant, el resultat d'aquesta prova és negatiu.

Podríem fer que el resultat de la prova fós positiu simplement retornant True:

```
def is_divisible_by_2_or_5(n):
    """
     >>> is_divisible_by_2_or_5(8)
     True
    """
    return True
```

Si ara demanem a l'intèrpret de Python que passi de nou el doctest per veure si les proves són positives ens trobarem que no s'escriu res:

```
  $ python -m doctest myfunctions.py
```

Aquest és el comportament esperat. Si en executar una prova de components no s'escriu res és que tot va com estava previst. Si volem que també en aquest cas s'escriguin missatges indicant com va la prova de components cal usar l'opció `-v` de la següent forma:

```
$ python -m doctest -v myfunctions.py
Trying:
    is_divisible_by_2_or_5(8)
Expecting:
    True
ok
1 items had no tests:
    myfunctions
1 items passed all tests:
   1 tests in myfunctions.is_divisible_by_2_or_5
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Tot que ara el programa passa les proves és clarament erroni ja que la funció retornarà `True` sense tenir en compte el valor dels seus paràmetres. Si afegíssim noves proves el resultat seria provablement erroni. A continuació teniu una versió completa de la funció `is_divisible_by_2_or_5` juntament amb un conjunt de proves més complet:

```python
def is_divisible_by_2_or_5(n):
    """
      >>> is_divisible_by_2_or_5(8)
      True
      >>> is_divisible_by_2_or_5(7)
      False
      >>> is_divisible_by_2_or_5(5)
      True
      >>> is_divisible_by_2_or_5(9)
      False
    """
    return n % 2 == 0 or n % 5 == 0
```

Comproveu que ara la funció passa correctament la prova de components corresponent.

Exercicis

EXERCICI 5.1 Dissenyeu una funció amb un paràmetre enter n que escrigui per pantalla `"Si"` si `n` és parell i `"No"` en cas que `n` sigui senar.

```python
def parell(n):
    ...
```

Ara modifiqueu la funció per tal que torni el resultat com un valor en comptes d'escriure'l.

EXERCICI 5.2 Dissenyeu una funció que calculi el màxim de dos nombres. Acompanyeu-la d'un conjunt de casos de prova que siguin exhaustius. Aquesta funció, pot aplicar-se indistintament a arguments de tipus enter o real?

Exercici 5.3 Quin deu ser el resultat d'executar el següent programa? ATENCIÓ: No el proveu directament en el intèrpret!! Primer raoneu quin és el resultat. Després, si voleu, comproveu-ho amb el intèrpret. En cas que el resultat de l'intèrpret no coincideixi amb el que havíeu pensat, esbrineu en què us havíeu equivocat en el raonament.

```
def f2(x,y):
    t = x + y
    return y ** t

x = 3
t = 4
y = f2(t,x)
print y
```

Exercici 5.4 Què escriu el següent programa?

```
def f(a):
    a = a + 5
    return a

b = 0
f(b)
print b, "," ,
b = f(b)
print b
```

Exercici 5.5 Dissenyeu una funció amb un paràmetre enter `n` que retorni `True` si `n` és parell i `False` en cas que `n` sigui senar. Documenteu la funció mitjançant casos de prova i verifiqueu el seu comportament.

Exercici 5.6 Dissenyeu una funció amb dos paràmetres reals `a` i `b` que retorni `True` si i solament si `a` és més gran que `b` i, al mateix temps, el producte d'`a` per `b` és la unitat. Documenteu la funció mitjançant doctests i assegureu-vos que passa correctament la prova.

Exercici 5.7 Dissenyeu una funció que donada una cadena `s` i una cadena `c` retorni la cadena `csc`. Per exemple, si `c="A"` i `s="BB"`, la funció retorna `"ABBA"`. Com sempre, afegiu els casos de prova necessaris.

Exercici 5.8 Dissenyeu la funció `centra`. Aquesta funció té dos paràmetres: `w` i `s`. El paràmetre `w` és un enter que indica l'amplària d'una pàgina mesurada en caràcters. El paràmetre `s` és una cadena. La funció ha de tornar una cadena de caràcters tal que, quan s'escriu mostra la paraula `s` centrada en una línia de `w` caràcters d'ample. Afegiu els casos de prova necessaris i assegureu-vos que funciona correctament.

Exercici 5.9 La darrera lletra del DNI es pot calcular a través dels seus nombres. Per calcular-la, cal calcular el residu resultant de dividir el valor numèric del DNI per 23. El mòdul és un nombre enter entre 0 i 22. Cada valor del residu té associada una lletra. La lletra que correspon a cada valor es mostra en la següent taula:

Residu	Lletra
0	T
1	R
2	W
3	A
4	G
5	M
6	Y
7	F
8	P
9	D
10	X
11	B
12	N
13	J
14	Z
15	S
16	Q
17	V
18	H
19	L
20	C
21	K
22	E

Es demana que escriguis un script que llegeixi pel teclat un número de DNI i escrigui per pantalla la lletra que li correspon. Estructura el programa en base a funcions i, especialment, crea una funció que calculi i retorni la lletra del DNI. Documenta'l de manera adequada tot incloent els casos de prova que s'escaiguin. Específicament tingues en compte el cas dels nombres negatius.

Exemple d'execució:

```
Introdueix DNI: 99999999
Lletra del NIF: R
```

EXERCICI 5.10 Escriu el següent programa en un fitxer de nom `quadrant.py`. Executa'l i complementa'l amb el joc de proves que s'escaigui.

```
from math import ceil # ceil arrodoneix per exces

def calcula_quadrant(a):
    return int(ceil(a) % 360) / 90

angle = float(raw_input("Escriu un angle en graus: "))
quadrant = calcula_quadrant(angle)
if quadrant == 0:
    print "primer quadrant"
elif quadrant == 1:
    print "segon quadrant"
elif quadrant == 2:
    print "tercer quadrant"
elif quadrant == 3:
    print "quart quadrant"
```

Exercici 5.11 Escriviu un programa que calculi la conversió a bitllets i monedes d'una quantitat exacta d'euros. Considereu que hi ha bitllets de 500, 200, 100, 50, 20, 10 i 5 euros i monedes de 2 i 1 euros. Utilitzeu les eines de documentació i verificació de manera adequada.

Exemple de funcionament:

```
Introdueix euros: 434

2 bitllets de 200 euros
1 bitllet de 20 euros
1 bitllet de 10 euros
2 monedes de 2 euros
```

Exercici 5.12 Escriu un programa tal que donat un nombre enter, escrigui si aquest és el doble d'un nombre senar. Utilitza les eines de documentació i verificació de manera adequada.

Exemple de funcionament:

```
Introdueix nombre: 14
El 14 es el doble del nombre senar 7

Introdueix nombre: 12
El 12 no es el doble de cap nombre senar
```

Exercici 5.13 Dissenyeu una funció que donat un valor enter x i un interval tancat $[a, b]$, retorni:

- El caràcter `'+'` si x és més petit que el límit inferior de l'interval.
- El caràcter `'='` si x es troba dins de l'interval.
- El caràcter `'-'` si x és més gran que el límit superior de l'interval.

Per implementar la funció, primer escriviu la capçalera, després un joc de casos de prova, i finalment el cos de la funció. Comproveu que satisfà els casos de prova.

Exercici 5.14 Estudia el següent codi `Python` i, sense usar l'intèrpret, raona que escriu. Després, comprova que no t'has equivocat fent servir l'intèrpret:

```
def f(a,b):
    return a + b

def g(a,b):
    return a * b

def prova(m,n,r,f):
    x = f(m,n)
    if x == r:
        print "ok"
    else:
        print "fail"

prova(3,2,5,f)
prova(4,6,8,f)
prova(2,2,4,g)
```

Exercici 5.15 Per a cadascun dels següents enunciats, defineix una funció i el corresponent joc de proves. Recorda escriure primer la capçalera de la funció, després el joc de proves i finalment el cos. No t'oblidis de verificar que passa el joc de proves.

a) Una funció tal que donats els coeficients d'una equació de segon grau, retorna `True` ssi té una solució doble.

b) Una funció tal que donada una cadena de caràcters, retorna `True` ssi comença i acaba amb la lletra `"z"`.

c) Una funció tal que donats dos reals m i n que representen la massa en kilograms de dos objectes i un tercer real d que és una distància en metres, retorna la força d'atracció entre ambdós objectes mesurada en Newtons.

Exercici 5.16 En aquest exercici anem a crear una funció "universal" per calcular una fórmula. Prenguem la Llei d'Ohm, per exemple. Aquesta llei ens indica que en un circuit elèctric de corrent contínua es compleix:

$$I = \frac{V}{R}$$

on R és la resistència, I és la intensitat i V la diferència de potencial. Es vol que implementeu una funció `ohm(r,i,v)` tal que, si li passem com arguments una tríada en la que un valor és `None` retorna aquest valor segons la llei d'Ohm. Per exemple, si la cridem amb `ohm(None,0.2,1.4)` hauria de tornar 7.0, ja que $R = \frac{V}{I}$ i, per tant, $R = \frac{1.4}{0.2} = 7.0$. En canvi, si la cridem com `ohm(1200, 0.3, None)` hauria de tornar 360.0, ja que $V = RI$ i per tant $R = 1200 \cdot 0.3 = 360.0$. Finalment, si cap dels arguments és `None`, la funció ha de tornar `True` o `False` segons si els valors donats compleiexen o no la Llei d'Ohm.

Recordeu acompanyar la funció d'un joc de proves exhaustiu i de comprovar que el satisfà.

Exercici 5.17 Seguint la mateixa estratègia de l'exercici 5.16, dissenyeu una funció similar que permeti resoldre tots els casos que es presenten en un tir parabòlic ideal. Aquí les dades són:

Paràmetre	Significat
θ	Angle del tir
v	Mòdul de la velocitat inicial
d_x	Distància horitzontal recorreguda
d_y	Distància vertical recorreguda

6 Iteració

6.1 Assignacions repetides

Com potser ja has descobert, és legal fer més d'una assignació a una mateixa variable. Una nova assignació fa que una variable ja existent es refereixi a un nou valor (i ja no faci referència al valor anterior).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

La sortida d'aquest programa és `5 7`, ja que el primer cop que s'imprimeix `bruce`, el seu valor és 5, i la segona vegada, el seu valor és 7. La coma al final de la primera sentència **print** suprimeix el salt de línia després de la sortida, la qual cosa explica perquè les dues sortides apareixen a la mateixa línia.

Amb l'assignació múltiple és especialment important distingir entre una operació d'assignació i la sentència d'igualtat. Donat que `Python` usa el signe igual (=) per a les assignacions, pot ser temptador interpretar la sentència `a = b` com una sentència d'igualtat. No ho és!

Noteu que la igualtat és simètrica, i l'assignació no ho és. Per exemple, en matemàtiques, si $a = 7$ aleshores $7 = a$. Però a `Python`, la sentència `a = 7` és legal i, en canvi, la sentència `7 = a` no ho és.

És més, a matemàtiques una sentència d'igualtat sempre és certa. Per exemple, si `a = b` ara, aleshores `a` sempre serà igual a `b`. En un programa `Python`, una sentència d'assignació pot fer que el valor de dues variables esdevingui igual. Ara bé, aquesta "coincidència de valors" no ha de mantenir-se així indefinidament. Observeu:

```
a = 5
b = a # a i b son ara iguals
a = 3 # a i b son de nou diferents
```

En l'exemple anterior, la tercera línia canvia el valor d'`a` però no canvia el valor de `b`, per tant, si bé després de la segona línia `a=b`, després de la tercera ja no són iguals.

En alguns llenguatges de programació, per tal de remarcar que l'assignació i la igualtat són operacions diferents, s'utilitza un símbol diferent per a l'assignació. Són símbols habituals $<-$ o $:=$. D'aquesta manera s'afavoreix la distinció entre assignació i igualtat.

6.2 Actualització de variables

Una de les formes més habituals de l'assignació és l'actualització, on el nou valor de la variable depèn del seu valor anterior:

```
x = x + 1
```

Això significa: pren el valor d'x, suma-li una unitat i, posteriorment, actualitza x amb el nou valor obtingut.

Si proves d'actualitzar una variable que no existeix, obtindràs un error, ja que Python avalúa l'expressió de la banda dreta de l'assignació abans d'assignar el valor resultant a la variable de la banda esquerra:

```
>>> x = x + 1
Traceback (most recent call last):
  File "", line 1, in
NameError: name 'x' is not defined
```

Abans de poder actualitzar una variable, cal haver-la inicialitzat, normalment amb una assignació senzilla. Per exemple:

```
>>> x = 0
>>> x = x + 1
>>>
```

Actualitzar una variable sumant-li 1 s'anomena incrementar; restar-li 1 s'anomena decrementar.

6.3 La sentència "while"

Els computadors sovint s'usen per automatitzar tasques repetitives. Repetir tasques que són idèntiques o molt semblats sense equivocar-se és una cosa que els computadors fan molt bé i els éssers humans força malament.

L'execució repetida d'un conjunt de sentències s'anomena *iteració*. Com que la iteració és tan usual, Python proporciona diverses sentències per a expressar-la de manera fàcil. La primera sentència que veurem és la sentència **while**.

A continuació et trobaràs una funció anomenada countdown que mostra l'ús de la sentència **while**:

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print "Blastoff!"
```

Gairebé es pot llegir la sentència **while** com si fos en català. Significa, mentre n sigui més gran que 0, escriu el valor d'n i, a continuació, decrementa el valor de n en 1. Quan arribis a 0, escriu a la terminal la paraula *Blastoff!*.

Més formalment, aquest és el flux d'execució d'una sentència **while**:

1. Avalua la condició, obtenint True o False.
2. Si la condició és False, acaba la sentència **while** i continúa l'execució a partir de la següent sentència.
3. Si la condició és True, executa cadascuna de les sentències del cos i retorna al pas 1.

El cos el formen totes les sentències que trobem després de la capçalera **while** i que tenen el mateix nivell d'indentació.

bucle (loop)

Aquest tipus de flux s'anomena *bucle* per l'estructura circular que comporta que el tercer pas retorni al primer. Cal notar que si la condició és False la primera vegada que s'arriba al bucle, les sentències del cos del bucle no s'executaran mai.

El cos d'un bucle ha de canviar com a mínim el valor d'una o més variables de tal manera que en algun moment la condició esdevingui False i el bucle acabi. Altrament, el bucle es repetirà per sempre, convertint-se en el que es coneix com a *bucle infinit*.

bucle infinit (infinite loop)

En el cas de countdown podem provar que el bucle acaba perquè sabem dues coses: que el valor d'n és finit; i que el valor d'n disminueix a cada iteració. Així, tard o d'hora n ha d'arribar al 0. En altres casos, no és tan fàcil d'assegurar. Dóna un cop d'ull a la següent funció definida per a tots els enters positius n:

```
def sequence(n):
    while n != 1:
        print n,
        if n % 2 == 0: # n is even
            n = n / 2
        else: # n is odd
            n = n * 3 + 1
```

La condició del bucle és n != 1, així que el bucle continuarà fins que n sigui 1, la qual cosa farà la condició falsa.

A cada iteració, el programa escriu el valor d'n i després comprova si n és parell o senar. Si és parell el valor de n es divideix entre 2. Si és senar, el valor se substitueix per n * 3 + 1. Per exemple, si el valor inicial fos 3, la seqüència resultant seria $3, 10, 5, 16, 8, 4, 2, 1$.

Com que n a vegades s'incrementa i a vegades es decrementa, no hi ha cap prova òbvia de que n arribarà a 1 o de que el programa acaba. Per alguns valors concrets d'n, podem provar que acaba. Per exemple, si el valor de partida és una potència de dos, aleshores el valor d'n serà parell a cada iteració fins que esdevingui 1. En un cert moment del càlcul l'exemple anterior cau en una seqüència d'aquest tipus. Exactament això passa quan n esdevé 16. A partir d'aquest punt, ja sabem que acabarà.

Valors concrets a banda, la qüestió realment interessant és demostrar que aquest programa acaba per tots els valors possibles d'n. Fins ara, ningú ha estat capaç de provar-ho o refutar-ho!

6.4 La traça d'un programa

Per tal d'escriure programes efectius és interessant que un programador desenvolupi l'habilitat de *rastrejar* un programa. Rastrejar significa "convertir-se en l'ordinador" i seguir el flux d'execució del programa, anotant l'estat de totes les variables, paràmetres i de qualsevol sortida que el programa pugui generar després d'executar cada instrucció.

rastrejar (trace)

Per tal d'entendre aquest procés, anem a rastrejar la crida a sequence(3) de la secció anterior. En un principi, tenim un paràmetre, n, amb un valor inicial de 3. Donat que 3 és diferent d'1, el cos del bucle **while** s'executa. S'escriu 3 i s'avalua 3 % 2 == 0. Com que s'avalua a False, la branca de l'**else** s'executa i, en conseqüència, ara s'avalua 3 * 3 + 1 i s'assigna el resultat a n.

Per tal de seguir el rastre d'un programa a mà, fes en un tros de paper una taula amb una columna per a cada una de les variables que es crearan. Afegeix-hi una altra columna per a la sortida. En l'exemple que estem rastrejant aquesta taula tindria aquest aspecte:

n	sortida
3	3
10	

traça
(empty)

D'aquesta taula en diem *traça*. Una traça ens mostra com evoluciona l'execució d'un programa per a unes dades concretes.

Donat que 10 != 1 s'avalua a True, el cos del bucle s'executa de nou i s'escriu un 10. 10 % 2 == 0 és True, així que la branca de l'**if** s'executa i n pren per valor 5. En arribar al final la traça és:

n	sortida
3	3
10	10
5	5
16	16
8	8
4	4
2	2
1	

Rastrejar un programa sol ser una feina tediosa i propensa als errors. És per aquest motiu que fem fer aquest procés als ordinadors!. Tot i això és una habilitat essencial que han de tenir els programadors. A partir d'aquest rastre podem aprendre molt sobre com funciona el nostre programa. Podem observar que, tant aviat com n esdevé una potència de 2, per exemple, el programa requerirà $\log_2 n$ iteracions per completar-se. També podem veure que l'1 final no s'escriurà.

estat
(empty)

La traça està formada per una seqüència de files. Cadascuna d'aquestes files es correspon amb un *estat* de l'execució del programa. Vist així, podríem dir que l'execució d'un programa és un procés que comença en l'*estat inicial* i va evolucionant per diversos estats fins arribar a l'*estat final*.

estat inicial
(empty)

estat final
(empty)

6.5 Comptatge de dígits

La següent funció compta el número de dígits d'un enter positiu expressat en format decimal:

```
def num_digits(n):
    count = 0
    while n:
        count = count + 1
        n = n / 10
    return count
```

Una crida a num_digits(710) retornarà 3. Rastreja l'execució d'aquesta crida a funció per tal de convence't.

Aquesta funció demostra un altre patró de programació anomenat comptador. La variable count s'inicialitza a 0 i s'incrementa cada vegada que s'executa el cos del bucle. Quan el bucle acaba, count conté el resultat: el nombre total de vegades que el cos del bucle s'ha executat, la qual cosa és igual al nombre de dígits.

Si només volguéssim comptar el nombre de dígits que són igual a 0 o a 5, caldria afegir una sentència condicional abans d'incrementar el comptador:

```
def num_zero_and_five_digits(n):
    count = 0
    while n:
        digit = n % 10
```

```
        if digit == 0 or digit == 5:
            count = count + 1
        n = n / 10
    return count
```

Rastreja la funció anterior i confirma que num_zero_and_five_digits(1055030250) retorna 7.

6.6 Assignació abreviada

Incrementar una variable és tan habitual que Python proporciona una sintaxi abreviada per fer-ho:

```
>>> count = 0
>>> count += 1
>>> count
1
>>> count += 1
>>> count
2
>>>
```

count += 1 és una abreviació de count = count + 1. El valor de l'increment no ha de ser 1 forçosament:

```
>>> n = 2
>>> n += 5
>>> n
7
>>>
```

També hi ha altres abreviatures com -=, *=, /= i %=, que tenen significats anàlegs al que ja s'ha explicat:

```
>>> n = 2
>>> n *= 5
>>> n
10
>>> n -= 4
>>> n
6
>>> n /= 2
>>> n
3
>>> n %= 2
>>> n
1
```

6.7 Taules

Una de les coses per a les quals són molt útils els bucles és per generar informació tabular. Abans de que les computadores fossin totalment operatives, les persones havien de calcular els logaritmes, el sinus, el cosinus i altres funcions matemàtiques a mà. Per fer-ho més fàcil, els llibres matemàtics contenien

llargues taules que llistaven els valors d'aquestes funcions. Crear les taules era un procés lent i avorrit i, sovint, contenien errors.

Quan van aparèixer els ordinadors a l'escena, una de les reaccions incials va ser: És fantàstic! Podem usar els ordinadors per generar les taules, i així no hi haurà errors. Això va resultar ser majoritàriament veritat però va durar poc temps. Poc temps després, els ordinadors i les calculadores es van tornar tan potents que les taules van acabar essent obsoletes.

Bé, gairebé obsoletes. Per algunes operacions, els computadors usen taules de valors per tal de primer obtenir respostes aproximades i posteriorment realitzar càlculs per tal de millorar l'aproximació. En alguns casos, hi ha hagut errors en aquestes taules, com els famosos errors a la taula que l'Intel Pentium usava per realitzar les divisions de coma flotant.

Tot i que una taula de logaritmes ja no és tan útil com ho era abans, encara serveix com a bon exemple de iteració. El següent programa escriu una taula de dues columnes. La columna de l'esquerra mostra una seqüència de valors i la de la dreta dos elevat a cadascún d'aquests valors:

```
x = 1
while x < 13:
    print x, '\t', 2**x
    x += 1
```

La cadena `'\t'` representa el caràcter tabulador. La barra inversa a `'\t'` indica l'inici d'una seqüència d'escapada. Les seqüències d'escapada s'usen per representar caràcters invisibles com els tabuladors o els salts de línia. La seqüència `'\n'` representa un salt de línia.

Una seqüència d'escapada pot aparèixer a qualsevol punt d'una cadena. En aquest exemple, la seqüència d'escapada del tabulador és l'únic caràcter de la cadena. Com creus que es representaria el caràcter barra inversa en una cadena?

El caràcter tabulador desplaça el cursor a la dreta fins que aquest arriba a una columna de tabulació. Les tabulacions són útils per aliniar els resultats que s'escriuen en columnes. La sortida del programa anterior n'és un exemple:

```
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	1024
11	2048
12	4096
```

A causa del caràcter tabulador que s'escriu entre cada parell de resultats la posició de la segona columna no depèn del nombre de dígits de la primera columna.

6.8 Taules de dues dimensions

Una taula de dues dimensions és aquella en la que es llegeixen els valors a la intersecció d'una fila i una columna. Una taula de multiplicar n'és un bon exemple. Suposem que volem imprimir la taula de multiplicar pels valors de l'1 al 6.

Una bona manera de començar és escriure el bucle que imprimeix els mútliples de 2, tots en una línia:

```
i = 1
while i <= 6:
    print 2 * i, '  ',
    i += 1
print
```

La primera línia inicialitza una variable anomenada i que actua com a comptador o variable de bucle. A mida que el bucle s'executa, el valor d'i s'incrementa de 1 a 6. Quan la i val 7 el bucle acaba. A cada iteració es mostra el valor de 2 * i, seguit per tres espais.

De nou, la coma final de la sentència **print** suprimeix el salt línia. Després de que el bucle acabi, la segona sentència **print** comença una nova línia.

La sortida del programa és:

```
2 4 6 8 10 12
```

Fins aquí tot correcte. El proper pas consisteix a encapsular i generalitzar.

6.9 Encapsulació i generalització

L'encapsulació és el procés de d'embolicar un fragment de codi en una funció. Encapsulant un fragment de codi a els beneficis per als quals són bones les funcions. Fins al moment, ja has vist dos exemples d'encapsulació: print_parity a l'apartat 4.5 i es_divisible a l'apartat 5.4.

Generalitzar significa agafar un fragment de codi específic, com per exemple el que escriu per pantalla els múltiples de 2, i reescriure'l de forma més general, per exemple fent que escrigui els múltiples de qualsevol enter.

La següent funció encapsula el bucle anterior i el generalitza per tal d'escriure el múltiples d'n:

```
def print_multiples(n):
    i = 1
    while i <= 6:
        print n * i, '\t',
        i += 1
    print
```

Per tal d'encapsular, tot el que hem hagut de fer és afegir la primera línia, que declara el nom de la funció i la llista de paràmetres. Per tal de generalitzar, hem hagut de substituir el valor 2 pel paràmetre n.

Si cridem aquesta funció amb 2 com a argument, obtenim la mateixa sortida que a l'apartat anterior. Si ho fem amb 3 com a argument, la sortida és:

```
3 6 9 12 15 18
```

I amb 4, la sortida és:

```
4 8 12 16 20 24
```

A hores d'ara probablement ja sabràs com escriure una taula de multiplicar: cridant print_multiples repetidament amb arguments diferents. De fet, podem usar un altre iteració per fer-ho:

```
i = 1
while i <= 6:
    print_multiples(i)
    i += 1
```

FIxeu-vos en com de similar és aquest bucle al que conté la funció print_multiples. Tot el que hem fet ha sigut reemplaçar la sentència **print** per una crida a funció.

La sortida d'aquest programa és una taula de multiplicar:

```
1   2   3   4   5   6
2   4   6   8  10  12
3   6   9  12  15  18
4   8  12  16  20  24
5  10  15  20  25  30
6  12  18  24  30  36
```

6.10 Més encapsulació

Per tal d'exemplificar de nou l'encapsulació, prenem el codi de la secció anterior i emboliquem-lo en una funció:

```
def print_mult_table():
    i = 1
    while i <= 6:
        print_multiples(i)
        i += 1
```

Aquesta manera de fer és una tècnica de desenvolupament de programari força habitual. Desenvolupem codi escribint línies de codi aïllades de qualsevol funció o, fins i tot, directament sobre l'interpret de `Python`. Quan obtenim un codi que funciona, l'extraiem i l'emboliquem en una funció.

Aquesta tècnica de desenvolupament és particularment útil si no saps com dividir un programa en funcions quan el comences a escriure. Aquesta aproximació et permet fer el disseny a mida que vas escrivint.

6.11 Variables locals

Potser t'estàs preguntant com podem usar la mateixa variable, i, a les dues funcions print_multiple i print_mult_table. No es generen problemes quan una de les funcions canvia el valor de la variable?

La resposta és no, donat que la i a print_multiples i la i a print_mult_table no són la mateixa variable.

Les variables creades a dins d'una definició de funció són locals; no es pot accedir a una variable local des de fora de la funció en que s'han definit. Això vol dir que pots tenir diverses variables amb el mateix nom sempre i quan no es trobin a la mateixa funció.

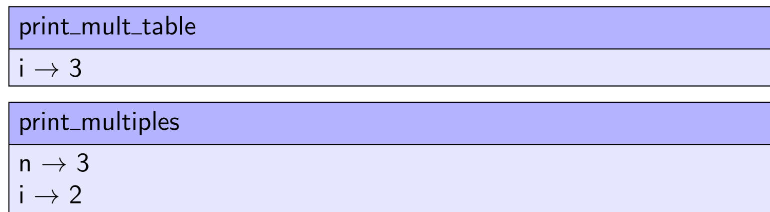


Figura 6.1: Diagrama de pila parcial en un instant de l'escriptura de la taula de multiplicar.

El diagrama de pila d'aquest programa, vegeu la figura 6.1, mostra que les dues variables anomenades i no són la mateixa variable. Poden contenir diferents valors, i canviar-ne un no afecta a l'altre.

El valor d'i a print_mult_table va d'1 a 6. En el diagrama de pila és 3. A la següent iteració serà 4. A cada iteració print_mult_table crida print_multiples amb el valor actual d'i com argument. Aquest valor s'assigna al paràmetre n.

A dins de print_multiples, el valor d'i va de 1 a 6. Al diagrama és 2. Canviar aquesta variable no té cap efecte en el valor d'i a print_mult_table.

És habitual i perfectament legal tenir diverses variables locals amb el mateix nom. Particularment, noms com i i j sovint s'usen com a variables de bucle. Si evites usar-les en una funció només perquè ja les has utilitzat en una altra banda, probablement faràs el programa més difícil de llegir.

6.12 Més generalització

Per exemplificar una vegada més la generalització, imagina que vols un programa que imprimeixi una taula de multiplicar de qualsevol mida, no només de sis per sis. Podries afegir un paràmetre a print_mult_table:

```
def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i)
        i += 1
```

Hem substituit el valor 6 pel paràmetre high. Si cridem a print_mult_table amb l'argument 7, mostrarà:

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18
4	8	12	16	20	24
5	10	15	20	25	30
6	12	18	24	30	36
7	14	21	28	35	42

Això està bé, excepte que probablement voldríem que la taula fos quadrada, és a dir amb el mateix nombre de files i columnes. Per fer això, afegim un altre paràmetre a print_multiples per tal d'especificar quantes columnes ha de tenir la taula.

Només per ser pesats, també anomenem a aquest paràmetre high. Amb això demostrem que funcions diferents poden tenir paràmetres amb el mateix nom, tal i com passava amb les variables locals. Aquí hi ha el programa sencer:

```
def print_multiples(n, high):
    i = 1
    while i <= high:
        print n*i, '\t',
        i += 1
    print

def print_mult_table(high):
    i = 1
    while i <= high:
        print_multiples(i, high)
        i += 1
```

Cal notar que quan hem afegit un nou paràmetre, hem hagut de canviar la primer línia de la funció, que sovint anomenem capçalera, i també hem hagut de canviar la crida a la funció en el cos de print_mult_table.

Tal i com esperàvem, aquest programa genera una taula quadrada de set per set si el cridem com print_mult_table(7):

```
1   2   3   4   5   6   7
2   4   6   8  10  12  14
3   6   9  12  15  18  21
4   8  12  16  20  24  28
5  10  15  20  25  30  35
6  12  18  24  30  36  42
7  14  21  28  35  42  49
```

Quan es generalitza una funció apropiadament, sovint s'obté un programa amb capacitats no esperades. Per exemple, potser has notat que donat que $ab = ba$, totes les entrades a la taula apareixen dues vegades. Podries estalviar-te tinta imprimint només la meitat de la taula. Per fer-ho, només s'ha de canviar una línia de print_mult_table. Cal canviar:

```
print_multiples(i, high)
```

Per:

```
print_multiples(i, i)
```

Si ara cridem la funció obtenim:

```
1
2   4
3   6   9
4   8  12  16
5  10  15  20  25
6  12  18  24  30  36
7  14  21  28  35  42  49
```

6.13 Funcions

Fins ara, hem mencionat en diverses ocasions algunes de les coses per les que les funcions són bones. A hores d'ara, potser t'estàs preguntant quines són aquestes coses. Aquí en van algunes:

1. Donar un nom a una seqüència de sentències fa els programes més fàcils de llegir i depurar.
2. Dividir un programa llarg en funcions et permet separar parts del programa, depurar-les individualment i aleshores composar-les en un tot.
3. Les funcions faciliten l'ús de les iteracions.
4. Les funcions ben dissenyades sovint són útils per diversos programes. Una vegades n'has escrit i depurat una, la pots reutilitzar en altres programes.

6.14 El mètode de Newton

Els bucles sovint s'usen en programes que calculen resultats numèrics començant per un resultat aproximat i millorant-lo iterativament.

Per exemple, una manera de calcular una arrel quadrada és usar el mètode de Newton. Suposem que vols esbrinar l'arrel quadrada de `n`. Si comences per gairebé qualsevol aproximació, pots calcular una aproximació millor amb la següent fórmula:

```
better = (approx + n/approx)/2
```

Aplicant repetidament aquesta fórmula fins que la millor aproximació sigui igual a la immediatament anterior, podem escriure una funció per calcular l'arrel quadrada:

```
def sqrt(n):
    approx = n/2.0
    better = (approx + n/approx)/2.0
    while better != approx:
        approx = better
        better = (approx + n/approx)/2.0
    return approx
```

Proveu de cridar aquesta funció amb l'argument 25 i confirmeu que retorna 5.0.

6.15 Algorismes

El mètode de Newton és un exemple d'algorisme: és un procés mecànic per resoldre una categoria de problemes (en aquest cas, calcular arrels quadrades).

No és fàcil definir un algorisme. Pot ajudar el fet de començar amb alguna cosa que no sigui un algorisme. Quan vas aprendre a multiplicar nombres d'una sola xifra, probablement vas memoritzar la taula de multiplicar. En efecte, vas memoritzar 100 solucions específiques. Aquest tipus de coneixement no és algorísmic.

Però si eres una mica mandrós, probablement vas fer trampa aprenent alguns trucs. Per exemple, per trobar el producte de n per 9, pots escriure $n - 1$ com a primer dígit i $10 - n$ com a segon dígit. Aquest truc és una solució general per multiplicar qualsevol número d'una sola xifra per 9. Això és un algorisme!

De manera semblant, les tècniques que vas aprendre per la suma portant-ne, la resta agafant-ne i les divisions llargues són totes algorismes. Una de les característiques dels algorismes és que no requereixen cap intel·ligència per a dur-los a terme. Són processos mecànics en els quals cada pas segueix a l'anterior d'acord a un conjunt senzill de normes.

En la nostra opinió, és empipador que els humans passin tant de temps a l'escola per aprendre a dur a terme uns algorismes que, literalment, no requereixen cap intel·ligència per a dur-se a terme.

D'altra banda, el procés de dissenyar algorismes és interessant, és un repte intel·lectual i és una part central d'allò que anomenem programació.

Algunes de les coses que la gent duu a terme naturalment, sense dificultats o pensament conscients, són de les més complicades d'expressar algorítmicament. Entendre el llenguatge natural és un bon exemple. Tots ho fem, però fins al moment ningú ha estat capaç d'explicar com ho fem exactament, com a mínim no en forma d'un algorisme.

Exercicis

EXERCICI 6.1 Escriu un petit script que demani un nombre a l'usuari i escrigui la taula de multiplicar d'aquest nombre en ordre invers.

Observa el següent exemple d'execució:

```
$ python taumult.py
Introdueix un nombre: 8
La taula de multiplicar del 8 es:
8 * 10 = 80
8 * 9 = 72
8 * 8 = 64
8 * 7 = 56
8 * 6 = 48
8 * 5 = 40
8 * 4 = 32
8 * 3 = 27
8 * 2 = 16
8 * 1 = 8
8 * 0 = 0
```

EXERCICI 6.2 Què fa el següent programa? Analitza què passa quan la dada és un nombre positiu i quan és un nombre negatiu. Escriviu una traça assumint que a l'entrada es llegeix l'enter 12.

```
n=input("Introdueix un nombre: ")
s=0
i=1
while i<=n:
    s=s+i
    i=i+1
print s
```

EXERCICI 6.3 Quin és el resultat d'executar següent codi? Escriviu la traça corresponent.

```
a=1
while a<3:
    while a<3:
        print "O",
    a=a+1
```

Exercici 6.4 Quin és el resultat d'executar següent codi? Escriviu la traça corresponent.

```
a=1
while a<3:
    if a % 2 == 0:
        b=1
        while b < 3:
            print "X",
            b = b + 1
        print "O",
    a = a + 1
```

Exercici 6.5 Escriu un petit script que calculi el factorial d'un nombre. Un exemple d'execució pot ser:

```
$ python fact.py
Introdueix un nombre: 5
El factorial de 5 es 120
```

Encapsuleu el càlcul del factorial en una funció. Doteu-la d'un joc de proves escaient i assegureu-vos que la verificació és correcta.

Exercici 6.6 Dissenyeu una funció que donats m i n retorni $\binom{m}{n}$. Acompanyeu-la d'un joc de proves raonable.

Exercici 6.7 Escriviu una funció amb un paràmetre enter n que retorni el nombre de digits d'n. Doteu-la d'un joc de proves convenient i comproveu que el passa.

Exercici 6.8 Escriviu una funció que rep un paràmetre enter n i retorna la suma dels dígits d'n. Doteu-la d'un joc de proves convenient i comproveu que el passa.

Exercici 6.9 Definiu la funció condensa(n) que rep un paràmetre natural n i retorna un valor enter $0 \leq r \leq 9$. Aquesta funció fa el següent: calcula la suma s_0 dels dígits d'n; si s_0 té més d'una xifra de llargada, calcula s_1 com la suma dels dígits d's_0 i així successivament fins que s'arriva a una suma s_i que té una única xifra. Aquest valor és el resultat final.

Per exemple, per calcular condensa(3613), calcularíem la suma dels seus dígits, que és 13. Com té més d'una xifra, sumaríem de nou els dígits per obtenir 4 que és el valor que retornaria la funció. Així doncs:

```
>>> condensa(3613)
4
```

Podeu usar el resultat de l'exercici 6.8. No oblideu afegir a la funció el joc de proves corresponent.

Exercici 6.10 Escriviu una funció, amb identificador matriu, que escrigui per la terminal una matriu de 10×10 tal i com es veu en el següent exemple:

```
>>> matriu()
1   2  3  4  5  6  7  8  9 10
2   3  4  5  6  7  8  9 10 11
3   4  5  6  7  8  9 10 11 12
4   5  6  7  8  9 10 11 12 13
5   6  7  8  9 10 11 12 13 14
6   7  8  9 10 11 12 13 14 15
7   8  9 10 11 12 13 14 15 16
8   9 10 11 12 13 14 15 16 17
9  10 11 12 13 14 15 16 17 18
10 11 12 13 14 15 16 17 18 19
```

Useu dues iteracions per a fer aquesta feina.

Exercici 6.11 Aquest exercici aprofita les funcions dels exercicis 5.7 i 5.8 i implementa un programa que permet provar-les. Aquest programa cal que mostri un menú amb les següents opcions:

```
Benvinguts
==========

1. Provar exercici A
2. Provar exercici B
S. Sortir
```

El programa llegirà l'opció del menú i executarà la primera o la segona funció. Per executar la funció, primer haurà de llegir les dades necessàries. Una vegada executada la funció i mostrat el resultat, s'haurà de mostrar de nou el menú d'inici fins que l'usuari introdueixi la cadena 'S'.

Exercici 6.12 Dissenyeu una funció que llegeixi números enters del teclat fins que l'usuari escrigui un zero i finalment mostri la suma de tots els números llegits.

Exercici 6.13 Escriu un programa tal que donat un nombre enter `n` introduït per l'usuari, escrigui els `n` primers enters parells positius. El primer parell positiu és el 0. Utilitza les eines de documentació i prova de manera escaient i fes que l'execució sigui la que mostren els exemples de funcionament següents.

```
escriu un enter: 8
Els 8 primers parells positius son:
0
2
4
6
8
10
12
14

Escriu un enter: 0
0 nombres a mostrar

Escriu un enter: −2
Valor incorrecte
```

Exercici 6.14 Implementeu un programa que demani un nombre enter `n` i retorni la piràmide conformada amb els seus dígits. Per exemple, donat el nombre 1234, el programa hauria d'escriure:

```
1
1 2
1 2 3
1 2 3 4
```

Documenteu la funció convenientment usant casos de prova i verifiqueu que funciona correctament.

Exercici 6.15 Escriviu un script que llegeixi del teclat una seqüència de reals acabada en el caràcter `@`. Una vegda llegida la seqüència ha d'escriure en el terminal els següents estadístics:

- El mínim.
- El màxim.
- La mitjana.
- El recorregut

Com a exemple, considereu la següent sessió de terminal:

```
$ python estad.py
12.5
−2.0
3.21
100.001
71.3
@
maxim: 100.001
minim: −2.0
mijana: 37,0022
recorregut: 102.001
```

Exercici 6.16 Un sensor de temperatura mostreja la temperatura de l'exterior cada 30 minuts. Volem dissenyar un script que llegeixi del teclat una seqüència d'aquestes temperatures acabada amb el símbol `@` i ens indiqui quans màxims locals ha trobat. Si la seqüència de temperatures és $t_0, t_1, t_2, \ldots, t_n$, direm que t_i és un màxim local si es cumpleix que $t_{i-1} < t_i$ i $t_i > t_{i+1}$.

Observeu la següent execució a tall d'exemple:

```
$ python temperatura.py
12.9
13
14
13.5
14
15.2
16.9
16.8
@
Hi ha 2 maxims locals
```

EXERCICI 6.17 Una forma habitual de calcular funcions trascendentals en els computadors quan no es disposa de llibreria matemàtica és usar aproximacions polinòmiques calculades a partir d'una sèrie de Taylor.

Definiu una funció que calcula $\sin x$ usant la sèrie de Taylor calculada en el punt $x = 0$ que té aquesta expressió:

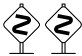
$$\sin x \approx \sum_{n=0}^{k} (-1)^n \frac{x^{2n+1}}{(2n+1)!}$$

Com sabeu, la sèrie de Taylor aproxima la funció exactament si sumem infinits termes. En el cas dels computadors ens hem d'acontentar amb una aproximació feta sumant un nombre finit de termes n tot i saber que fem un error —que pot set afitat amb la fórmula de l'error. Per determinar n, el nombre de termes que cal sumar, proveu dues estratègies:

a) Fixeu un valor fix $n = 5$.

b) Aneu sumant termes fins que la diferència entre dues aproximacions succesives sigui menor que un cert $\epsilon = 1.0 \times 10^{-6}$.

En preparar els casos de prova, serviu-vos de la llibreria matemàtica per obtenir un resultat fiable que pogueu comparar amb el vostre.



EXERCICI 6.18 Definiu al funció integra(f,a,b) que donada una funció f i dos reals a i b calcula $\int_a^b f(x)dx$. Un exemple de funcionament pot ser el següent:

```
>>> def g(x):
...     return 2 * x * x + 3
...
>>> integra(g, 0.0, 3.0)
27.000
```

regla del trapeci (empty)

A tal efecte useu l'algoritme d'integració basat en la *regla del trapeci*. Aquest mètode ens diu que:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}\big(f(a) + 2f(a+\Delta x) + 2f(a+2\Delta x) + \cdots + f(b)\big)$$
$$= \Delta x \left(\frac{f(a)+f(b)}{2} + \sum_{k=1}^{n-1} f(a + k\Delta x) \right)$$

on $\Delta x = \frac{b-a}{n}$ i n és el nombre de subdivisions. En la funció que es demana, us recomanem usar una n de 5000.

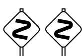
EXERCICI 6.19 Usant la funció de l'exercici 6.18, dissenyeu un script que tabula la següent funció:

$$F(s) = \int_{-10}^{s} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} dx$$

distribució normal (empty)

per valors de s que van de -9.5 a 10 de 0.5 en 0.5. $F(s)$ és la funció de *distribució normal*, fonamental en l'estadística. Malauradament no existeix una fórmula tancada per a aquesta funció i, per tant, la única forma de calcular-la és numèricament.

mètode de la bisecció (empty)



EXERCICI 6.20 El *mètode de la bisecció* és un algoritme simple per calcular de forma numèrica arrels d'equacions.

Suposem que volem resoldre l'equació $f(x) = 0$ i calcular una arrel d'f en l'intèrval $[a, b]$ assumint que f és contínua i que $f(a)$ i $f(b)$ tenen signes oposats. En aquestes condicions, pel teorema de Bolzano, és segur que hi ha una arrel dins de $[a, b]$.

El procediment és com segueix: es calcula m com el punt mig entre a i b; si $f(m) = 0$ llavors m és una arrel; en cas contrari $f(m)$ és positiu o negatiu; estrenyem l'intèrval $[a, b]$ substituint l'extrem que té el mateix signe que $f(m)$ per m i tornem a repetir el mateix procés. En cas que la mida de l'intèrval esdevingui més petita que un cert $\epsilon = 1.0 \times 10^{-5}$, considerem que el seu punt mig és l'arrel que buscàvem.

Es demana que feu el següent:

a) Estudieu el mètode i dibuixeu-ne una execució en un cas simple.

b) Implementeu la funció `root(f,a,b)`, que calcula una arrel de la funció `f` en l'interval definit per `a` i `b`. Assumiu que `f` és sempre un funció real d'una sola variable.

c) Comproveu el funcionament usant un joc de proves escaient.

7 Cadenes de caràcters

7.1 Un tipus de dades compost

tipus de dades compostos (compount datatypes)

Fins ara hem vist cinc tipus de dades: int, float, bool, NoneType i str. El tipus cadena de caràcters —o per fer-ho més breu simplement cadena— és qualitativament diferent dels altres quatre ja que els seus valors estan formats per peces més petites: els caràcters.

Els tipus que es composen de peces més petites s'anomenen *tipus de dades compostos*. Segons què estiguem fent, ens interessarà tractar un tipus compost de dades com a una sola cosa o, d'altres vegades, preferirem accedir a cadascuna de les seves parts. Aquesta dualitat entre el tot i les parts sovint és molt útil.

operador claudàtor (bracket operator)

L'*operador claudàtor* selecciona un únic caràcter d'una cadena. Fixeu-vos en l'exemple:

```
>>> fruit = "banana"
>>> letter = fruit[1]
>>> print letter
```

L'expressió fruit[1] selecciona —a vegades també es parla d'accedir— el caràcter número 1 de fruit. La variable letter fa referència al resultat. Quan mostrem el valor de letter, ens enduem una sorpresa:

```
a
```

La primera lletra de "banana" no és l'a, a no ser que siguis un enginyer informàtic. Per raons perverses, els enginyers informàtics sempre comencen a comptar des de zero. La lletra 0-èssima (0) de "banana" és b. La primera (1) lletra és la a i la segona (2) lletra és n.

Si vols la lletra 0 d'una cadena, cal posar un 0, o qualsevol expressió amb valor 0, entre els claudàtors:

```
>>> letter = fruit[0]
>>> print letter
b
```

índex (index)

L'expressió entre claudàtors s'anomena *índex*. Un índex especifica un element d'un conjunt ordenat, en aquest cas el conjunt de caràcters de la cadena. L'índex indica quin d'ells vols. Pot ser qualsevol expressió entera.

7.2 Longitud

La funció len retorna el nombre de caràcters d'una cadena:

```
>>> fruit = "banana"
>>> len(fruit)
6
```

Per tal d'obtenir la última lletra d'una cadena, pot ser que et sentis temptat a provar alguna cosa així:

```
length = len(fruit)
last = fruit[length] # ERROR!
```

No funciona! Aquest codi provoca el següent error en temps d'execució: *IndexError: string index out of range*. El motiu és que no existeix una sisena (6) lletra a "banana". Com que hem començat a comptar des de zero, les sis lletres estan numerades de 0 a 5. Per tal d'obtenir l'últim caràcter, hem de restar 1 a length:

```
length = len(fruit)
last = fruit[length-1]
```

De manera alternativa, podem usar índexos negatius, que comencen a comptar des del final de la cadena. L'expressió fruit[−1] conté la última lletra, fruit[−2] conté la penúltima, i així successivament.

7.3 Recorreguts i el bucle "for"

Molts càlculs involucren processar una cadena de caràcter en caràcter. Sovint comencen pel principi, seleccionen un caràcter a cada iteració, fan algun càlcul amb aquest caràcter i continuen fins haver esgotat els caracters de la cadena. Aquest esquema de processat s'anomena *recorregut*. Una manera de codificar un recorregut és mitjançant la sentència **while**:

recorregut (traversal)

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index += 1
```

Aquest bucle recorre la cadena i mostra cada lletra en una línia diferent. La condició d'aquest bucle és index < len(fruit), així que quan index és igual a la longitud de la cadena, la condició és falsa i el cos del bucle no s'executa més. L'últim caràcter accedit és el que té per índex len(fruit) − 1, que és el darrer caràcter de la cadena.

Utilitzar un índex per recórrer un conjunt de valors és tant freqüent que Python proporciona una sentència específica amb una sintaxi més simple: el bucle **for**.

```
for char in fruit:
    print char
```

A cada iteració del bucle, el següent caràcter de la cadena s'assigna a la variable char. El bucle continua fins que no hi ha més caràcters.

El següent exemple mostra com usar la concatenació i un bucle **for** per tal de generar sèries abecedàries. Les sèries abecedàries fan referència a sèries o llistes en les quals els elements apareixen per ordre alfabètic. Per exemple, al llibre *Make way for Ducklings* de Robert McCloskey, el nom dels ducklings són Jack, Kack, Lack, Mack, Nack, Ouack, Pack i Quack. El següent bucle escriu aquests noms per ordre:

```
prefixes = "JKLMNOPQ"
suffix = "ack"
for letter in prefixes:
    print letter + suffix
```

La sortida d'aquest programa és:

```
Jack
Kack
Lack
Mack
Nack
Oack
Pack
Qack
```

Evidentment, això no és del tot correcte perquè Ouack i Quack estan mal escrits... però il·lustra bé la sentència **for**.

7.4 Segments de cadenes

segment (slice)

Una subcadena d'una cadena s'anomena un *segment*. Seleccionar un segment és similar a seleccionar un caràcter:

```
>>> s = "Peter, Paul, and Mary"
>>> print s[0:5]
Peter
>>> print s[7:11]
Paul
>>> print s[17:21]
Mary
```

L'operador [n:m] retorna la part de la cadena des de l'n-èssim caràcter fins al m-èssim caràcter, incloent el primer però excloent el darrer. Aquest comportament, que és una mica contraintuitiu, té més sentit si t'imagines els índexos situats entre els caràcters i no damunt seu, com al diagrama de la figura 7.1. Fixa't que en aquest diagrama és clar que el segment [1:3] es refereix a la cadena "ol".

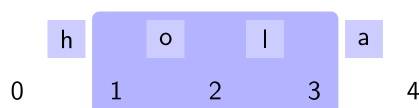


Figura 7.1: Forma d'imaginar-se els índexos quan es refereixen a segments d'una cadena.

Si omets el primer índex de l'operador de segmentació, (el d'abans dels dos punts), el segment comença al principi de la cadena. Si omets el segon índex, el segment va fins al final de la cadena. Observa l'exemple:

```
>>> fruit = "banana"
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

Què creus que significa s[:]?

7.5 Comparació de cadenes

L'operador d'igualtat també funciona per les cadenes. Per tal de veure si dues cadenes són iguals:

```
if word == "banana":
    print "Yes, we have no bananas!"
```

Altres operacions de comparació són útils per ordenar paraules en ordre lexicogràfic:

```
if word < "banana":
    print "Your word," + word + ", comes before banana."
elif word > "banana":
    print "Your word," + word + ", comes after banana."
else:
    print "Yes, we have no bananas!"
```

Aquest és similar a l'ordre alfabètic que usaries en un diccionari, excepte que totes les lletres majúscules apareixen abans que les lletres minúscules. Com a resultat:

```
Your word, Zebra, comes before banana.
```

Si la distinció entre majúscules i minúscules és un problema, una forma d'evitar-lo és convertir les cadenes a un format canònic, per exemple convertir tots els caràcters a minúscules, abans de fer les comparacions. Un problema més difícil de resoldre és fer que el programa s'adoni que les zebres no són fruites.

7.6 Les cadenes són immutables

Pot ser temptador usar l'operador [] a la banda esquerra d'una assignació, amb la intenció de canviar un caràcter en una cadena. Per exemple així:

```
greeting = "Hello, world!"
greeting[0] = 'J' # ERROR!
print greeting
```

En comptes d'escriure Jello, world!, aquest codi provoca aquest error en temps d'execució:

```
TypeError: 'str' object doesn't support item assignment
```

immutables (immutable)

Les cadenes són *immutables*, la qual cosa significa que no pots modificar una cadena que ja existeix. El millor que pots fer en aquests casos és crear una nova cadena que sigui una variació de l'original:

```
greeting = "Hello, world!"
newGreeting = 'J' + greeting[1:]
print newGreeting
```

La solució aquí és concatenar una nova primera lletra amb un segment de greeting. Aquesta operació no té cap efecte en la cadena original.

7.7 Els operadors “in” i “not in”

L’operador **in** comprova si una cadena és una subcadena d’una altra:

```
>>> 'p' in 'apple'
True
>>> 'i' in 'apple'
False
>>> 'ap' in 'apple'
True
>>> 'pa' in 'apple'
False
```

Cal notar que una cadena és subcadena d’ella mateixa:

```
>>> 'a' in 'a'
True
>>> 'apple' in 'apple'
True
```

Combinant l’operador **in** amb la concatenació de cadenes, podem escriure una funció que elimini totes les vocals d’una cadena. Mira:

```
def remove_vowels(s):
    vowels = "aeiouAEIOU"
    s_without_vowels = ""
    for letter in s:
        if letter not in vowels:
            s_without_vowels += letter
    return s_without_vowels
```

Comprova que aquesta funció fa el que volem que faci.

L’operador **not in** s’avalua exactament de forma contrària a **in**. Fixeu-vos en aquest exemple:

```
>>> "a" not in "bcdefghijk"
True
```

7.8 Una funció “find”

Què fa la següent funció?

```
def find(strng, ch):
    index = 0
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

En cert sentit, la funció find és antagònica a l’operador []. En comptes de prendre un índex i extreure’n el caràcter corresponent, prèn un caràcter i determina l’índex on apareix el caràcter. Si el caràcter no hi és, la funció retorna −1.

Aquest és el primer exemple que veiem d'una sentència **return** dins d'un bucle. Si strng[index] == ch, la funció retorna immediatament i acaba el bucle prematurament.

Si la cadena no conté el caràcter, aleshores el programa surt del bucle normalment i retorna −1.

cerca (search)

Aquest patró de computació s'anomena sovint recorregut eureka, o més seriosament *cerca*, ja que tant aviat com trobem el que busquem, podem cridar Eureka! i parar de cercar.

7.9 Bucles i comptatge

El següent programa compta el nombre de vegades que surt la lletra a en una cadena i és un altre exemple del patró de comptatge introduït a la secció 6.5, que explica com comptar dígits.

```
fruit = "banana"
count = 0
for char in fruit:
    if char == 'a':
        count += 1
print count
```

7.10 Paràmetres opcionals

Per tal de trobar les posicions de les aparicions segona i tercera d'un caràcter en una cadena, podem modificar la funció find, afegint un tercer paràmetre que indiqui l'índex inicial a partir del qual cal cercar dins de la cadena:

```
def find2(strng, ch, start):
    index = start
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

La crida find2('banana', 'a', 2) ara retorna 3, l'índex de la primera aparició d''a' a 'banana' després de l'índex 2. Què retorna find2('banana', 'n', 3)? Si respons 4, hi ha moltes possibilitats de què hagis entès com funciona find2.

Millor encara, podem combinar find i find2 usant un paràmetre opcional:

```
def find(strng, ch, start=0):
    index = start
    while index < len(strng):
        if strng[index] == ch:
            return index
        index += 1
    return -1
```

La crida find('banana', 'a', 2) amb aquesta versió de find es comporta exactament com find2, mentre que a la crida find('banana','a'), start prendrà el valor per omisió 0.

Afegint un altre paràmetre opcional a find, podem aconseguir que la funció cerqui endavant o enrere:

```
def find(strng, ch, start=0, step=1):
    index = start
    while 0 <= index < len(strng):
        if strng[index] == ch:
            return index
        index += step
    return -1
```

Si passem l'argument len(strng) − 1 al paràmetre start i l'argument −1 al paràmetre step, la funció buscarà de darrera cap a davant de la cadena en comptes de fer-ho en el sentit habitual. Cal fer notar que ens ha calgut afegir una comprovació del límit inferior d'index per tal de poder fer aquest canvi.

7.11 Operacions sobre cadenes

En els apartats anteriors ja hem vist que les cadenes es poden entendre des de dos punts de vista. Per un costat poden considerar-se com un tot i par l'altre com un conjunt de caràcters. També hem vist que hi ha algunes operacions que permeten operar cadenes enteses com un tot. Aquest és el cas, per exemple, de la concatenació de cadenes o de la funció len.

Hi ha moltes altres operacions que faciliten manipular cadenes i fer càlculs sofisticats. Observeu el següent exemple:

```
>>> s = "Una cadena"
>>> s.count('a')
3
```

count compta quantes vegades surt a la cadena s la lletra 'a'. Observeu amb atenció la sintaxi que hem usat per cridar count: el nom de la cadena seguit d'un punt seguit de la operació i finalment dels arguments. De les operacions que cridem usant aquesta sintaxi en diem *mètodes*. Segurament que entendreu millor manera de cridar als mètodes si ho interpreteu així:

mètodes (methods)

> ...demana-li a la cadena s que compti el nombre d''a'...

count sap comptar també subcadenes:

```
>>> "abracadabra".count("abra")
2
```

A banda de count les cadenes disposen de moltes més operacions:

```
>>> "abracadabra".capitalize()
"Abracadabra"
>>> "abracadabra".endswith("bra")
True
>>> "abracadabra".replace("bra","bla")
"ablacadabla"
```

Una operació interessant és find, que té un comportament similar al de la funció find que hem implementat en apartats anteriors. Si executeu el següent script:

```
estrofa = """
  Perque es alta i esvelta
  tota es sap estremir.
  Si els cabells li penjaven
  com el fruit del raim
```

```
    pels clotets de la sina
    s'hi perdien gotims!
"""
print estrofa.find("es")
print estrofa.find("es", 20)
```

El resultat serà:

```
12
23
```

7.12 Classificació de caràcters

Sovint és útil examinar un caràcter i comprovar si és una majúscula o una minúscula, o descobrir si és un número o una lletra. El mòdul `string` proporciona diverses constants que són útils per aquesta finalitat. string.digits n'és una. Com que string.digits és una cadena, la podem escriure per veure quin valor té. Noteu que prèviament haurem hagut d'importar el mòdul com és preceptiu:

```
>>> import string
>>> print string.digits
0123456789
```

string.digits conté una cadena formada per tots els caràcters que representen dígits.

A més, la cadena string.lowercase conté totes les lletres que el sistema considera que són minúscules. De manera semblant, string.uppercase conté totes les lletres majúscules. Proveu el següent i mireu què és el que obteniu:

```
print string.lowercase
print string.uppercase
```

Podem usar aquestes constants i el mètode find per tal de classificar caràcters. Per exemple, si ch.find(lowercase) retorna un valor diferent de −1, aleshores ch ha de ser una minúscula:

```
def is_lower(ch):
    return ch.find(string.lowercase) != −1
```

Com alternativa, també podeu fer servir l'operador **in**:

```
def is_lower(ch):
    return ch in string.lowercase
```

Com a tercera alternativa, és possible usar l'operador de comparació:

```
def is_lower(ch):
    return 'a' <= ch <= 'z'
```

Si ch es troba entre `'a'` i `'z'`, aleshores ha de ser una minúscula.

Finalment, podeu fer servir directament el mètode is_lower, que retorna True únicament quan la cadena a la que s'aplica està íntegrament en minúscules:

```
>>> "botifarra".is_lower()
True
>>> "Botifarra".is_lower()
False
```

Una altra constant definida al mòdul string pot ser que et sorprengui quan la escriguis:

caràcters blancs (whitespace characters)

```
>>> print string.whitespace
```

Els *caràcters blancs* mouen el cursor sense escriure res. Creen l'espai en blanc entre els caràcters visibles (si més no sobre el paper blanc). La constant string.whitespace conté tots els caràcters blancs, incloent l'espai, el tabulador ('\t') i el salt de línia ('\n').

7.13 El mètode per formatar cadenes

La manera més senzilla per formatar cadenes és usar el mètode format. Per entendre com funciona comencem amb alguns exemples:

```
s1 = "His name is {0}!".format("Arthur")
print s1

name = "Alice"
age = 10
s2 = "I am {0} and I am {1} years old.".format(name, age)
print s2

n1 = 4
n2 = 5
s3 = "2**10 = {0} and {1} * {2} = {3:f}".format(2**10, n1, n2, n1 * n2)
print s3
```

Si executes aquest script obtindràs:

```
His name is Arthur!
I am Alice and I am 10 years old.
2**10 = 1024 and 4 * 5 = 20.000000
```

format (format)

caselles (placeholders)

La idea fonamental rau en definir una cadena que determina el *format* i que conté un o més *caselles* numerades que representem així: {0},{1},{2},.... El mètode format usa el número o índex de la casella per decidir quin dels arguments del mètode cal posar en quina casella. Així, el primer argument es posarà en les caselles {0}, el segon en {1} i així successivament.

Cada casella d'un format pot contenir una especificació formal que indiqui com s'ha de posar l'argument corresponent en aquesta casella. Aquesta especificació sempre comença pels dos punts (:) i pot indicar coses com:

- La amplada que ha d'ocupar la casella en la cadena resultant.
- Si el contingut de la casella s'ha d'aliniar a l'esquerra <, centrar ^, o aliniar a l'esquerra >.
- Com cal convertir l'argument corresponent a cadena a fi i efecte de posar-lo en la casella. Noteu que en les caselles només podem posar-hi cadenes però l'argument pot ser, per exemple, un real. Si usem f, per exemple, interpretarem l'argument com un número en coma flotant. Si posem x, com un enter en hexadecimal.
- Si el tipus de conversió és f, podem espcificar també quants dígits decimals volem que surtin a la casella. Per exemple, si indiquem .2f estem dien que volem només 2 dígits decimals.

Vegem alguns exemples senzills però habituals. En la majoria de casos us trobareu amb casos com aquests. Si necessiteu definir formats més esotèrics, consulteu els detalls més gore en el manual.

```
n1 = "Paris"
n2 = "Whitney"
n3 = "Hilton"

print "Pi to three decimal places is {0:.3f}".format(3.1415926)
print "123456789 123456789 123456789 123456789 123456789 123456789"
print "|||{0:<15}|||{1:^15}|||{2:>15}|||Born in {3}|||".format(n1,n2,n3,1981)
print "The decimal value {0} converts to hex value {0:x}".format(123456))
```

Aquest script, per exemple, escriu el següent:

```
Pi to three decimal places is 3.142
123456789 123456789 123456789 123456789 123456789 123456789
|||Paris          |||    Whitney    |||         Hilton|||Born in 1981|||
The decimal value 123456 converts to hex value 1e240
```

En una cadena de format, podeu tenir diverses caselles que es refereixin al mateix argument; o fins i tot arguments que no referencia cap casella.

```
letter = '''
Dear {0} {2}.
    {0}, I have an interesting money-making proposition for you!
    If you deposit $10 million into my bank account, I can
    double your money ...
'''

print letter.format("Paris", "Whitney", "Hilton")
print letter.format("Bill", "Henry", "Gates")
```

L'exemple anterior genera el següent:

```
Dear Paris Hilton.
     Paris, I have an interesting money-making proposition for you!
     If you deposit $10 million into my bank account, I can
     double your money ...

Dear Bill Gates.
     Bill, I have an interesting money-making proposition for you!
     If you deposit $10 million into my bank account I can,
     double your money ...
```

Tai i com us podeu imaginar, si les caselles referencien arguments inexistents provocareu un error:

```
>>> "hello {3}".format("Dave")
Traceback (most recent call last):
  File "<interactive input>", line 1, in <module>
IndexError: tuple index out of range
```

L'exemple següent il·lustra la utilitat real dels formats. Primer mirem d'escriure una taula de resultats usant tabuladors però sense usar formats:

```
print("i\ti**2\ti**3\ti**5\ti**10\ti**20")
for i in range(1, 11):
    print(i, '\t', i**2, '\t', i**3, '\t', i**5, '\t',
          i**10, '\t', i**20)
```

Aquest programa escriu en una taula diverses potències dels nombres entre 1 i 10. Com l'amplada del tabulador en aquest exemple és de 8 caràcters i hi ha continguts que superen aquesta mida, la taula es descompensa:

```
i       i**2    i**3    i**5    i**10   i**20
1       1       1       1       1       1
2       4       8       32      1024    1048576
3       9       27      243     59049   3486784401
4       16      64      1024    1048576         1099511627776
5       25      125     3125    9765625         95367431640625
6       36      216     7776    60466176        3656158440062976
7       49      343     16807   282475249       79792266297612001
8       64      512     32768   1073741824      1152921504606846976
9       81      729     59049   3486784401      12157665459056928801
10      100     1000    100000  10000000000     100000000000000000000
```

Es podria canviar la mida de l'espai de tabulat però llavors la primera columna tindria un espai excessiu. La millor solució es usar formats i donar a cada columna el seu espai. Al mateix temps podem aprofitar per alinear a la dreta les columnes, com es fa sempre amb els nombres:

```
layout = "{0:>4}{1:>6}{2:>6}{3:>8}{4:>13}{5:>24}"
print layout.format('i', 'i**2', 'i**3', 'i**5', 'i**10', 'i**20')
for i in range(1, 11):
    print layout.format(i, i**2, i**3, i**5, i**10, i**20)
```

L'execució d'aquesta versió produeix un resultat molt millor:

```
   i  i**2  i**3    i**5        i**10                   i**20
   1     1     1       1            1                       1
   2     4     8      32         1024                 1048576
   3     9    27     243        59049              3486784401
   4    16    64    1024      1048576           1099511627776
   5    25   125    3125      9765625          95367431640625
   6    36   216    7776     60466176        3656158440062976
   7    49   343   16807    282475249       79792266297612001
   8    64   512   32768   1073741824     1152921504606846976
   9    81   729   59049   3486784401    12157665459056928801
  10   100  1000  100000  10000000000   100000000000000000000
```

Exercicis

EXERCICI 7.1 Encapsuleu el següent fragment de codi

```
cadena = "prova"
compta = 0
for char in cadena:
    if char == 'a':
        compta = compta + 1
    print compta
```

De tal manera que obtingueu una funció anomenada compta_lletres, i generalitzeu-la per que accepti la cadena i el caràcter com a paràmetres.

Exercici 7.2 Dissenyeu una funció que, donada una cadena, retorni el nombre de dígits que conté.

Exercici 7.3 Dissenyeu una funció que, donada una cadena, substitueixi tots els caràcters 'a' per 'A' (sense fer servir funcions o mètodes predefinits).

Exercici 7.4 Dissenyeu una funció que, donades dues cadenes, retorni el nombre d'aparicions de la segona cadena en la primera. Per exemple, donades les cadenes "pararapapa" i "pa", la funció hauria de retornar 3. Feu la implementació sense usar mètodes o funcions predefinides.

Exercici 7.5 Dissenyeu una funció que, donada una cadena, retorni aquesta mateixa cadena escrita al revés. Per exemple, donada la cadena "Hola", la funció hauria de retornar "aloH".

Exercici 7.6 Dissenyeu un programa que demani una cadena a l'usuari i, després, elimini totes les vocals. A continuació, el programa ha de demanar dos enters a l'usuari per tal d'obtenir la subcadena entre els índexos donats i, finalment, ha d'escriure per pantalla la cadena resultant, havent afegit cada tres caràcters la cadena "bogeria". Un exemple de funcionament podria ser:

```
Introdueix una cadena: "python mola molt"
El primer pas genera: pythn ml mlt
Introdueix el primer index de la subcadena: 3
Introdueix el segon index de la subcadena: 11
El segon pas genera: hn ml ml
La teva cadena resultant es: hn bogeriaml bogeriaml
```

Exercici 7.7 Els següents formats tenen errors. Arregleu-los:

a) "{0} {1} {2} {3}".format('aquest', 'aquell', 'algun')

b) "{1} {2} {3} {4}".format('si', 'no', 'dalt', 'baix')

c) "{0:d} {1:f} {2:f}".format(3, 3, 'tres')

Exercici 7.8 Es demana que dissenyeu un programa que llegeixi un text escrit en català (sense accents ni altres caràcters que no siguin lletres) i el torni a escriure en format SMS. Per exemple, si l'usuari del programa escriu:

```
Hola que tal per on passeu
```

Hauria d'escriure

```
ola ke tl x n paseu
```

A tal efecte, considereu els següents canvis a fer sobre la cadena (en l'ordre indicat):

- En els mots que comencen per `H`, suprimir l'`h`.
- Les síl·labes `"que"` o `"qui"` canviar-les per `"ke"` o `"ki"`.
- Les síl·labes `"per"` canviar-les per `"x"`.
- En els monosíl·labs d'una sola vocal i com a mínim 2 lletres, suprimir la vocal. Considereu que un monosíl·lab és un mot que només té una vocal. Naturalment les vocals de `"ke"` o `"ki"` no s'han de substituïr.
- Les dobles `s`, simplificar-les amb una sola `s`.

A tal efecte, feu servir aquells mètodes de les cadenes que us siguin útils i organitzeu el codi en funcions (per exemple, una funció per a cada canvi possible). Feu ús dels doctests per documentar i verificar el programa. Procediu de manera incremental. Primer un programa que només fa un canvi, despres un que en fa dos, i així successivament.

Exercici 7.9 Imagineu que treballeu per un departament de bioinformàtica i us demanen que busqueu en cadenes d'ADN (com la de l'Albumina, proteïna indicadora de desnutrició) un patró per detectar si aquesta té algun problema. La cadena d'ADN es representa per una cadena de caracters escollits entre `g`, `t`, `c`, `u` i `a`. A tal efecte us demanen que dissenyeu:

a) Una funció per trobar si hi ha la cadena: `'ggttaacaa**ggtttca'` on ** pot ser qualsevol parell de nucleòtids d'entre aquests: `g`, `t`, `c` o `a`.

b) Una funció que determini si existeix una cadena de 16 nucleòtids en que els 8 primers siguin reflex dels 8 següents, és a dir és una subcadena simètrica.

Exercici 7.10 Feu una funció que calculi si una paraula es cap-i-cua.

```
def es_capicua(s):
'''
    >>> es_capicua('abba')
    True
    >>> es_capicua('abab')
    False
    >>> es_capicua('tenet')
    True
    >>> es_capicua('banana')
    False
    >>> es_capicua('straw warts')
    True
'''
```

Exercici 7.11 Feu un programa que llegeixi del teclat una cadena i la escrigui de nou després de transformar qualsevol paraula a la mateixa paraula escrita en majuscules.

Exercici 7.12 Feu un programa que demani paraules a l'usuari fins que escrigui la paraula `"fi"`. Cada cop que l'usuari escriu una nova paraula cal imprimir per pantalla totes les anteriors seguida de la última escrita.

Exercici 7.13 La taula d'amortitzacions en un crèdit hipotecari es pot calcular segons diversos mètodes. En el cas de l'anomenat "mètode francès", la quota en cada període és fixa i es calcula d'acord amb la següent fórmula:

$$Q = C\frac{i}{1-(1+i)^{-n}}$$

on Q és la quota per període, C és el capital, i l'interès expressat en tant per u i n el nombre de períodes. En cada perìode, la quota es dedica a:

- Una part a pagar els interessos del capital pendent d'amortitzar.
- El sobrant de la quota —la fracció que no es dedica a pagar interessos— es dedica a amortitzar el capital, que minva després de cada quota.

Es vol que implementeu script per calcular taules d'amortització segons aquest mètode. Un exemple d'utilització seria:

```
$ python amort.py
Capital: 90000
Interes: 12.0
Periodes: 5

 ===  =============  =============  =============  =============
   N          Quota      Interesos   Amortitzacio        Capital
 ===  =============  =============  =============  =============
  1:       24966.88       10800.00       14166.88       75833.12
  2:       24966.88        9099.97       15866.90       59966.22
  3:       24966.88        7195.95       17770.93       42195.29
  4:       24966.88        5063.44       19903.44       22291.85
  5:       24966.88        2675.02       22291.85           0.00
 ===  =============  =============  =============  =============
```

Exercici 7.14 Dissenyeu una funció i el corresponent joc de proves tal que, donades dues cadenes de caràcters, retorni la subcadena comuna més llarga. Per exemple, si les cadenes són `"abcddbcbcdfabdc"` i `"xfdbcbaadd"`, llavors la subcadena comuna més llarga possible és `"dbcb"` atès que podem sobreposar una i altra cadena de forma òptima així:

```
abcddbcbcdfabdc
   |    |
  xfdbcbaadd
```

En aquest cas una i altra cadena comparteixen la subcadena `"dbcb"` i qualsevol altra cadena que comparteixen és més petita, per tant aquesta és la subcadena comuna més llarga possible.

Exercici 7.15 Dissenyeu una funció i el corresponent joc de proves que, donada una cadena, retorni el nombre de paraules que conté. A tal efecte considereu que una paraula és una seqüència de caracters que sempre acaba en un espai, en un punt o en una coma.

Exercici 7.16 Dissenyeu una funció anomenada `nent` i el corresponent joc de proves tal que, donada una cadena retorni el nombre de números naturals que conté. Observeu les següents proves per entendre-ho:

```
>>> nent('Quan tenia 12 anys vaig estar 23 dies malalta')
2
>>> nent('15 son 15 i 15 seran')
3
>>> nent('La noia cofoia mirava la boia')
0
```

Exercici 7.17 Escriviu dues funcions que facin el mateix però de diferent forma. El que han de fer és, donat un enter positiu, tornar-ne la representació en base 16. Per exemple, donat el 28 hauríen de tornar la cadena `'1C'`. Fixeu-vos que el dígit de pes 10 és `'A'`, el de pes 11, `'B'` i així successivament.

La primera funció ha de fer el càlcul basant-se en la forma tradicional de calcular un canvi de base, és a dir, calculant residus de divisions.

La segona funció ha d'usar el formatat de cadenes per calcular directament el resultat.

Comproveu que una i altra passen exactament el mateix joc de proves.

8 Llistes

Una *llista* és un conjunt ordenat de valors en que cada valor s'identifica amb un índex. Els valors que constitueixen una llista s'anomenen *elements*. Les llistes són semblants a les cadenes, que són conjunts ordenats de caràcters, excepte pel fet que els elements d'una llista poden tenir qualsevol tipus. Les llistes i les cadenes —i altres coses que es comporten com a conjunts ordenats— en l'argot de `Python` s'anomenen de forma genèrica *seqüències*.

llista (list)

elements (elements)

seqüències (sequences)

8.1 Creació de llistes

Hi ha diverses maneres de crear una llista. La manera més senzilla és incloure els elements entre claudàtors (`[` i `]`). Per exemple:

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

El primer exemple correspon a una llista de quatre enters. El segon és una llista de tres cadenes. Els elements d'una llista no han de ser obligatòriament del mateix tipus. La següent llista conté una cadena, un real, un enter i —*mirabile dictu*— una altra llista:

```
["hello", 2.0, 5, [10, 20]]
```

Per rematar el tema dels possibles elements d'una llista, cal dir que hi ha una llista especial que no conté cap element. És l'anomenada *llista buida* i s'escriu com `[]`.

llista buida (empty list)

Tal i com succeïa amb els valors numèrics 0 i amb la cadena buida, la llista buida en una expressió booleana denota el valor fals.

```
>>> if []:
... print 'Es cert."
... else:
... print 'Es fals."
...
Es fals.
>>>
```

Amb totes aquestes formes de crear llistes fora decebedor si no poguéssim assignar valors de tipus llista a variables o usar llistes com a paràmetres de funcions. De fet podem:

```
>>> vocabulary = ["ameliorate", "castigate", "defenestrate"]
>>> numbers = [17, 123]
>>> empty = []
>>> print vocabulary, numbers, empty
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []
```

8.2 Accés als elements

La sintaxi per accedir als elements d'una llista és la mateixa que per accedir als caràcters d'una cadena —l'operador claudàtor (`[]`), que no heu de confondre amb la llista buida. L'expressió de dins els claudàtors especifica l'índex. Recordeu que l'índex del primer element és 0.

```
>>> print numbers[0]
17
```

Es pot usar com a índex qualsevol expressió de tipus enter. Usar expressions d'altres tipus provocaria un error d'execució:

```
>>> numbers[9-8]
5
>>> numbers[1.0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers
```

Si proveu d'accedir a un element inexistent, també provocareu un error d'execució:

```
>>> numbers[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Si un índex té valor negatiu, compta cap enrere des del final de la llista:

```
>>> numbers[-1]
5
>>> numbers[-2]
17
>>> numbers[-3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

En aquest exemple, `numbers[-1]` és el darrer element de la llista, `numbers[-2]` és el penúltim i `numbers[-3]` no existeix.

És habitual usar com a índex d'una llista la variable associada a una iteració.

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
while i < 4:
  print horsemen[i]
  i += 1
```

En aquesta iteració **while**, la i varia entre 0 i 4. Quan la variable val 4, la condició falla —s'avalua a False— i la iteració acaba. Per tant el cos del bucle s'executa pels valors d'i 0, 1, 2 i 3.

recorregut d'una llista (list traversal)

A cada iteració la variable i s'usa com a índex per accedir als elements de la llista i s'escriu l'element i-èssim. Aquest patró de programació es coneix com a *recorregut d'una llista*.

8.3 Longitud d'una llista

La funció len retorna la longitud d'una llista, és a dir el nombre d'elements. És una bona idea usar aquest valor com a fita superior en les iteracions en comptes d'un valor constant. D'aquesta manera, si la longitud de la llista canvia, no hauràs de passejar-te pel programa canviant totes les iteracions. Funcionaran correctament per a qualsevol longitud de llista:

```
horsemen = ["war", "famine", "pestilence", "death"]

i = 0
num = len(horsemen)
while i < num:
    print horsemen[i]
    i += 1
```

La darrera vegada que s'executa el cos de la iteració, i val len(horsemen) − 1, que és l'índex del darrer element. Quan i és igual a len(horsemen), la condició falla i el cos no s'executa, cosa que és ben desitjable, ja que len(horsemen) no és un índex admissible.

Encara que una llista pot contenir una altra llista, la llista continguda compta com un sol element. Per exemple, la longitud de la següent llista és 4:

```
['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]
```

8.4 Pertinença a una llista

in és un operador booleà que calcula la pertinença a una seqüència. L'hem usat prèviament amb les cadenes, però també funciona amb llistes i altres tipus de dades que siguin seqüències:

```
>>> horsemen = ['war', 'famine', 'pestilence', 'death']
>>> 'pestilence' in horsemen
True
>>> 'debauchery' in horsemen
False
```

Atès que pestilence és un element de la llista horsemen, l'operador **in** torna True. Anàlogament, com que debauchery no és membre de la llista, **in** retorna False.

Podem usar **not** combinant-ho amb **in** per comprovar si un element no pertany a una llista:

```
>>> 'debauchery' not in horsemen
True
```

8.5 Operacions amb llistes

L'operador + concatena llistes:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
>>> print c
[1, 2, 3, 4, 5, 6]
```

De manera similar, l'operador $*$ repeteix la llista un determinat nombre de vegades:

```
>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

En el primer exemple es repeteix [0] quatre vegades. El segon repeteix la llista [1,2,3] tres vegades.

8.6 Segments de llista

Les operacions de segmentació que hem vist en el capítol de cadenes també funcionen en el cas de llistes:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3]
['b', 'c']
>>> a_list[:4]
['a', 'b', 'c', 'd']
>>> a_list[3:]
['d', 'e', 'f']
>>> a_list[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

8.7 La funció "range"

Les llistes que contenen enters consecutius són molt habituals. Per això `Python` contempla una forma simplificada per a crear-les:

```
>>> range(1, 5)
[1, 2, 3, 4]
```

La funció range pren dos arguments i retorna una llista que conté els enters que van del primer al segon, incloent el primer però *no el segon*. Així doncs, range(a,b) correspon al conjunt d'enters definit per l'interval $[a, b) \subset \mathbb{Z}$.

Hi ha dues formes més de range. Amb un únic argument es crea una llista que comença per 0. Per exemple:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si hi ha un tercer argument, aquest denota l'espai entre valors successius de la llista. És el que s'anomena la *mida del pas* o simplement *pas*. El següent exemple compta d'1 fins a 10 amb passos de 2

```
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
```

Si el pas és negatiu, llavors el "començament" ha de ser més gran que el "final".

```
>>> range(20, 4, -5)
[20, 15, 10, 5]
```

o, en cas contrari, s'obtindria una llista buida.

```
>>> range(10, 20, -5)
[]
```

8.8 Les llistes són mutables

mutables (mutable)

Al contrari que les cadenes, les llistes són *mutables*, la qual cosa significa que podem canviar els seus elements. Usant l'operador claudàtors a l'esquerra d'una assignació, hom pot modificar un dels elements d'una llista:

```
>>> fruit = ["banana", "apple", "quince"]
>>> fruit[0] = "pear"
>>> fruit[-1] = "orange"
>>> print fruit
['pear', 'apple', 'orange']
```

L'operador claudàtors aplicat a una llista pot sortir en qualsevol lloc dins una expressió. Quan surt a l'esquerra d'una assignació el que fa és modificar un dels elements de la llista. Així, el primer element de fruit ha passat de banana a pear, i el darrer element de quinze a orange. L'assignació a un element no funciona en el cas de les cadenes:

```
>>> my_string = 'TEST'
>>> my_string[2] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

però ho fa correctament en el cas de les llistes:

```
>>> my_list = ['T', 'E', 'S', 'T']
>>> my_list[2] = 'X'
>>> my_list
['T', 'E', 'X', 'T']
```

Si a més hi combinem l'operador de segmentació, podem modificar diversos elements simultàniament:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3] = ['x', 'y']
>>> print a_list
['a', 'x', 'y', 'd', 'e', 'f']
```

Amb aquestes eines podem esborrar elements d'una llista assignant-los-hi la llista buida:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> a_list[1:3] = []
>>> print a_list
['a', 'd', 'e', 'f']
```

I també podem afegir elements a una llista encabint-los en un segment buit en la posició desitjada:

```
>>> a_list = ['a', 'd', 'f']
>>> a_list[1:1] = ['b', 'c']
>>> print a_list
['a', 'b', 'c', 'd', 'f']
>>> a_list[4:4] = ['e']
>>> print a_list
['a', 'b', 'c', 'd', 'e', 'f']
```

8.9 Esborrat en llistes

Emprar segments per esborrar elements d'una llista pot ser rebuscat i, per tant, propens a errors. `Python` preveu una alternativa més senzilla: la sentència **del**.

```
>>> a = ['one', 'two', 'three']
>>> del a[1]
>>> a
['one', 'three']
```

Tal i com es pot esperar, **del** admet índexs negatius i provoca un error d'execució en cas que l'índex sigui fora de rang. Pot usar-se un segment com argument de l'operació **del**:

```
>>> a_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del a_list[1:5]
>>> print a_list
['a', 'f']
```

Com es usual, els segments fan referència a tots els elements fins, però sense incloure, el segon índex.

8.10 Objectes i valors

Si executem aquestes assignacions:

```
a = "banana"
b = "banana"
```

sabem que a i b es referiran a una cadena amb les lletres banana. Tot i això no podem saber si ambdues variables fan referència a la *mateixa* cadena. Hi ha dues configuracions possibles tal i com es veu a la figura 8.1.

Figura 8.1: Relació entre variable, objecte i valor: hi ha dues configuracions possibles.

En un cas, a i b fan referència a dues coses diferents que tenen el mateix valor. En el segon cas, ambdues referencies la mateixa cosa. Aquestes «coses» tenen nom, s'anomenen *objectes*. Un objecte és qualsevol

objectes (objects)

cosa que una variable pot referenciar.

Cada objecte té un *identificador* únic que podem obtenir usant la funció `id`. Escrivint l'identificador que correspon als valors de les variables `a` i `b`, podem conèixer si referencien el mateix objecte.

```
>>> id(a)
135044008
>>> id(b)
135044008
```

En efecte obtenim el mateix identificador, la qual cosa significa que `Python` només ha creat un objecte de tipus cadena i tant `a` com `b` referencien el mateix objecte. Noteu que els identificadors, en el vostre cas, no han de coincidir amb els de l'exemple, és clar!

És interessant veure que les llistes es comporten de manera diferent. Quan creem dues llistes, obtenim dos objectes:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
135045528
>>> id(b)
135041704
```

En aquest cas, el diagrama d'objectes corresponent és el de la figura 8.2. En aquest cas, `a` i `b` tenen el mateix valor però no referencien el mateix objecte.

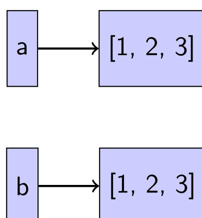


Figura 8.2: Diagrama d'objectes corresponent a dues llistes.

8.11 Àlies

Atès que les variables referencien objectes, si assignem una variable a una altra, ambdues acaben referenciant el mateix objecte:

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(a) == id(b)
True
```

En aquest cas, el diagrama d'objectes acaba com s'indica a la figura 8.3.

Com que el mateix objecte llista té dos noms diferents, `a` i `b`, diem que un és *àlies* de l'altre. Naturalment, els canvis que es fan a través d'un àlies afecten l'altre:

àlies (alias)

```
>>> b[0] = 5
>>> print a
[5, 2, 3]
```

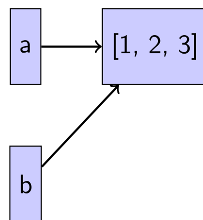


Figura 8.3: Una variable és àlies de l'altra.

Tot i que aquest comportament és útil, sovint és indesitjable o inesperat. En general, és més segur evitar els àlies quan es treballa amb objectes mutables. Òbviament, en objectes immutables els àlies no provoquen cap conflicte. Aquesta és la raó per la que, com hem vist abans, `Python` genera àlies de les cadenes de manera automàtica amb la intenció d'optimitzar l'espai de memòria emprat.

8.12 Clonat de llistes

clonat (clone)

Si volem modificar una llista i, al mateix temps, conservar una còpia de la llista original, hem de poder duplicar l'objecte llista, no només copiar-ne una referència. Aquest procés s'anomena *clonat* per evitar l'ambigüitat del mot «còpia». La forma més senzilla de clonar una llista és amb l'operador de segmentació:

```
>>> a = [1, 2, 3]
>>> b = a[:]
>>> print b
[1, 2, 3]
```

Quan es calcula un segment d'`a` es crea una nova llista. En aquest exemple concret el segment resulta ser la llista completa. Ara es poden fer canvis a `b` sense afectar `a`:

```
>>> b[0] = 5
>>> print a
[1, 2, 3]
```

8.13 Llistes i l'iterador "for"

L'iterador **for** també pot usar-se amb llistes. La sintaxi generalitzada per aquesta construcció és la següent:

```
for VARIABLE in LIST:
    BODY
```

Aquesta sentència es equivalent a la següent:

```
i = 0
while i < len(LIST):
    VARIABLE = LIST[i]
    BODY
    i += 1
```

La iteració **for** és més sintètica ja que permet eliminat la variable i de la iteració. Observeu el resultat de reescriure el fragment de codi de l'apartat ?? emprant una iteració **for**:

```
for horseman in horsemen:
    print horseman
```

Gairebé sembla llenguatge natural! Proveu de llegir-lo en veu alta: per a cada horseman en (la llista de) horsemen, escriviu (el nom del) horseman.

En una construcció **for** es pot usar qualsevol expressió de tipus llista:

```
for number in range(20):
    if number % 3 == 0:
        print number

for fruit in ["banana", "apple", "quince"]:
    print "I like to eat " + fruit + "s!"
```

El primer exemple escriu tots els múltiples de 3 entre 0 i 19. El segon exemple descriu el que ens agraden diverses fruites.

Com que les llistes són mutables és freqüent voler recórrer-les tot i modificant cadascun dels seus elements. El següent fragment de codi eleva al quadrat cadascun dels seus elements

```
numbers = [1, 2, 3, 4, 5]

for index in range(len(numbers)):
    numbers[index] = numbers[index]**2
```

Fixeu-vos un moment en l'expressió range(len(numbers)) i mireu d'entendre què significa exactament. Si us hi fixeu veureu que en aquesta iteració estem tant interessats en l'*índex* com en el *valor* dels elements de la llista ja que ambdós són necessaris en el cos del **for**.

Aquesta situació és tant freqüent que Python ofereix una forma simplificada d'escriure-la:

```
numbers = [1, 2, 3, 4, 5]

for index, value in enumerate(numbers):
    numbers[index] = value**2
```

enumerate és una funció que genera tant l'índex com el valor associat durant el recorregut de la llista. Observeu l'exemple següent per entendre millor com funciona enumerate:

```
>>> for index, value in enumerate(['banana', 'apple', 'pear', 'quince']):
... print index, value
...
0 banana
1 apple
2 pear
3 quince
>>>
```

8.14 Les llistes com a paràmetres

Passar una llista com argument d'una funció comporta realment el pas d'una referència a la llista. Atès que les llistes són mutables, els canvis fets sobre el paràmetre també afecten l'argument. Per exemple, la funció següent pren una llista com argument i multiplica cada element d'aquesta llista per 2:

```
def double_stuff(a_list):
    for index, value in enumerate(a_list):
        a_list[index] = 2 * value
```

Si escriviu la funció double_stuff en un fitxer anomenat `ch09.py`, podeu comprovar el seu funcionament amb l'intèrpret fent:

```
>>> from ch09 import double_stuff
>>> things = [2, 5, 'Spam', 9.5]
>>> double_stuff(things)
>>> things
[4, 10, 'SpamSpam', 19.0]
>>>
```

En aquest exemple, el paràmetre a_list i la variable things són àlies del mateix objecte. La figura 8.4 il·lustra aquesta situació.

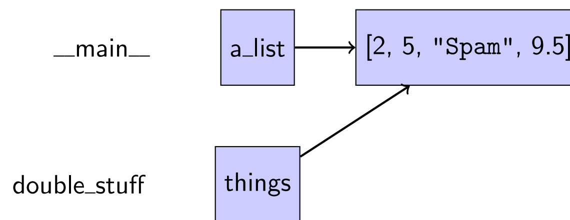


Figura 8.4: Els arguments de tipus mutables comporten la creació d'àlies.

Si la funció modifica el paràmetre, qui la invoca es veu afectat pel canvi.

8.15 Funcions pures i accions

Les funcions que prenen llistes com arguments i les modifiquen durant la seva execució s'anomenen *accions* i els canvis que provoquen s'anomenen *efectes laterals*.

accions (modifiers)

efectes laterals (side effects)

Una *funció pura* és aquella que no provoca efectes laterals. Es comunica amb el programa que la invoca només a través dels seus paràmetres, que mai modifica, i del valor de retorn. A continuació podeu veure la funció double_stuff implementada ara com a funció pura:

funció pura (pure function)

```
def double_stuff(a_list):
    new_list = []
    for value in a_list:
        new_list += [2 * value]
    return new_list
```

Aquesta versió de double_stuff no modifica els seus arguments. Comproveu-ho:

```
>>> from ch09 import double_stuff
>>> things = [2, 5, 'Spam', 9.5]
>>> double_stuff(things)
[4, 10, 'SpamSpam', 19.0]
>>> things
[2, 5, 'Spam', 9.5]
>>>
```

Per usar aquesta versió pura de la funció double_stuff i modificar la variable things cal assignar el valor de retorn de la funció a la variable:

```
>>> things = double_stuff(things)
>>> things
[4, 10, 'SpamSpam', 19.0]
>>>
```

Qualsevol cosa que pot fer-se amb accions també es pot fer amb funcions pures. De fet, alguns llenguatges de programació només permeten funcions pures. Hi ha evidències que indiquen que els programes que usen funcions pures es desenvolupen més de pressa i són més robustos davant els errors. Malgrat tot, les accions a vegades són convenients i, en certs casos, els programes funcionals són menys eficients.

programació funcional (functional programming)

En general recomanem que useu funcions pures sempre que sigui raonable fer-ho i que només empreu accions si això implica una avantatge fonamental. Aquest estil de programació bé podria dir-se'n estil de *programació funcional*.

8.16 Llistes niuades

llista niuada (nested list)

Una *llista niuada* és una llista que, a la vegada, és un element d'una altra llista. Per exemple, en la següent llista l'element d'índex 3 és una llista niuada:

```
>>> nested = ["hello", 2.0, 5, [10, 20]]
```

Si escrivim nested[3], obtenim [10, 20]. Per extreure un element de la llista niuada, podem fer-ho en dos passos:

```
>>> elem = nested[3]
>>> elem[0]
10
```

O també podem combinar els dos passos i fer:

```
>>> nested[3][1]
20
```

L'operador claudàtor s'avalua d'esquerra a dreta. Per tant l'expressió de l'exemple obté el tercer element de nested i n'extreu l'element d'índex 1.

Els operadors claudàtor s'avaluen d'esquerra a dreta, per tant aquesta expressió obté l'element 3 de nested i n'extreu l'element 1.

8.17 Matrius

Les llistes niuades s'usen sovint per representar matrius. Per exemple, la matriu següent:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

es pot representar com:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

matrix és una llista de tres elements en que cada element és una fila de la matriu. Podem accedir a una fila sencera de la matriu de la forma normal:

```
>>> matrix[1]
[4, 5, 6]
```

O podem accedir a un element de la matriu usant la indexació doble de la forma habitual:

```
>>> matrix[1][1]
5
```

El primer índex tria la fila i el segon la columna. Tot i que aquesta forma de representar matrius és corrent, no és l'única possibilitat. Una petita variació consisteix a emmagatzemar la matriu com una llista de columnes en comptes de fer-ho com una llista de files. Més endavant veurem una tercera forma de representar-les radicalment diferent usant diccionaris.

8.18 Mètodes del tipus llista

De la mateixa manera que succeia per les cadens, els objectes de tipus llista tenen una col·lecció molt important de mètodes que permeten manipular-los:

```
>>> mylist = []
>>> mylist.append(5)
>>> mylist.append(27)
>>> mylist.append(3)
>>> mylist.append(12)
>>> mylist
[5, 27, 3, 12]
>>>
```

append és un mètode de les llistes que afegeix l'argument al final de la llista. Continuant amb aquest exemple, mostrem altres mètodes de les llistes:

```
>>> mylist.insert(1, 12)
>>> mylist
[5, 12, 27, 3, 12]
>>> mylist.count(12)
2
>>> mylist.extend([5, 9, 5, 11])
>>> mylist
[5, 12, 27, 3, 12, 5, 9, 5, 11])
>>> mylist.index(9)
```

```
6
>>> mylist.count(5)
3
>>> mylist.reverse()
>>> mylist
[11, 5, 9, 5, 12, 3, 27, 12, 5]
>>> mylist.sort()
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 12, 27]
>>> mylist.remove(12)
>>> mylist
[3, 5, 5, 5, 9, 11, 12, 27]
>>>
```

Experimenta amb els mètodes de les llistes que surten en aquest exemple fins que n'estiguis segur que entens com treballen.

desenvolupament guiat per tests (test driven development)

8.19 Desenvolupament guiat per tests

El *desenvolupament guiat per tests*, és una tècnica de desenvolupament de programari en que s'arriba a l'objectiu desitjat a través d'una sèrie de petits passos, iterativament. Cadascun d'aquest passos estan guiats per proves automatitzades que *s'escriuen primer* i que expressen refinaments successius de l'objectiu.

Els doctests permeten emprar la metodologia del TDD fàcilment. Suposem que volem dissenyar una funció que crea una matriu de rows per columns.

El primer que fem és escriure la capçalera i el doctest corresponent en el fitxer `matrices.py`:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """
```

L'execució d'aquesta prova falla:

```
$ python -m doctest matrices.py
**********************************************************************
File "matrices.py", line 3, in __main__.make_matrix
Failed example:
    make_matrix(3, 5)
Expected:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
Got nothing
**********************************************************************
1 items had failures:
   1 of 1 in __main__.make_matrix
***Test Failed*** 1 failures.
```

La prova falla per que el cos de la funció és buit, per tant retorna None. La nostra prova indicava que volíem que retornés una matriu de 3×5 zeros.

La regla per usar TDD és *emprar la solució més simple* quan esteu escrivint una solució que passa la prova. En aquest cas doncs podem retornar senzillament el resultat desitjat:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
    """
    return [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

La prova ara passa correctament però la implementació de make_matrix sempre retorna el mateix resultat, que clarament no és el que volíem. Per avançar fem necessària la millora afegint un nou cas de prova:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
    """
    return [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
```

que com esperàvem, falla:

```
$ python -m doctest matrices.py
**********************************************************************
File "matrices.py", line 5, in __main__.make_matrix
Failed example:
    make_matrix(4, 2)
Expected:
    [[0, 0], [0, 0], [0, 0], [0, 0]]
Got:
    [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
**********************************************************************
1 items had failures:
   1 of 2 in __main__.make_matrix
***Test Failed*** 1 failures.
```

guiada per proves (test driven)

Aquesta tècnica s'anomena *guiada per proves* per que el codi cal escriure'l únicament per satisfer els casos de prova. Motivats per el nou cas que falla ara podem aportar una solució més general:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
    """
    return [[0] * columns] * rows
```

Aquesta solució sembla correcta i podem pensar que hem acabat la implementació. Succeeix, però, que quan usem la funció més endavant descobrim un error:

```
>>> from matrices import *
>>> m = make_matrix(4, 3)
>>> m
[[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>> m[1][2] = 7
>>> m
[[0, 0, 7], [0, 0, 7], [0, 0, 7], [0, 0, 7]]
>>>
```

Volíem assignar a l'element de la segona fila tercera columna el valor 7. En comptes d'això, *tots* els elements de la tercera columna valen 7.

Després d'una reflexió, ens n'adonem que en la nostra solució cada fila és un *àlies* de les altres files. Definitivament això no és el que es volia, per tant encarem la solució del problema *primer escrivint un cas de prova que falli*:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
      >>> m = make_matrix(4, 2)
      >>> m[1][1] = 7
      >>> m
      [[0, 0], [0, 7], [0, 0], [0, 0]]
    """
    return [[0] * columns] * rows
```

Ara que tenim un cas que falla i cal esmenar proposem una solució millor:

```
def make_matrix(rows, columns):
    """
      >>> make_matrix(3, 5)
      [[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
      >>> make_matrix(4, 2)
      [[0, 0], [0, 0], [0, 0], [0, 0]]
      >>> m = make_matrix(4, 2)
      >>> m[1][1] = 7
      >>> m
      [[0, 0], [0, 7], [0, 0], [0, 0]]
    """
    matrix = []
    for row in range(rows):
        matrix += [[0] * columns]
    return matrix
```

Usar TDD aporta diversos beneficis al procés de desenvolupament de programari:

- Ens ajuda a pensar d'una forma concreta sobre el problema que volem resoldre abans de resoldre'l.
- Ens encoratja a trencar un problema en problemes més petits i senzills. Al mateix temps ens condueix a resoldre'ls pas a pas d'una forma incremental.
- Ens assegura que tindrem un conjunt de casos de prova automàtics ben desenvolupats per al nostre programari. Això en facilita les modificacions i millores posteriors.

8.20 Cadenes i llistes

Python te una operació anomenada list que, donat un paràmetre de tipus seqüència, crea una llista amb els seus elements.

```
>>> list("Crunchy Frog")
['C', 'r', 'u', 'n', 'c', 'h', 'y', ' ', 'F', 'r', 'o', 'g']
```

També existeix la funció str, que donat un paràmetre de qualsevol tipus retorna una representació en forma de cadena del mateix.

```
>>> str(5)
'5'
>>> str(None)
'None'
>>> str(list("nope"))
"['n', 'o', 'p', 'e']"
```

Com podem veure en el darrer exemple, str no pot usar-se per concatenar una llista de caràcters. Per fer això podem usar el mètode join de les cadenes.

```
>>> char_list = list("Frog")
>>> char_list
['F', 'r', 'o', 'g']
>>> ''.join(char_list)
'Frog'
```

Dos dels mètodes més útils del tipus cadena estan relacionades amb les llistes. split trenca una cadena en una llista de mots. L'opció predeterminada és considerar qualsevol nombre d'espais blancs com el separador dels mots:

```
>>> song = "The rain in Spain..."
>>> song.split()
['The', 'rain', 'in', 'Spain...']
```

Un paràmetre opcional del mètode, anomenat delimiter, permet especificar quins caràcters cal usar com a separadors de mots. L'exemple següent fa servir la cadena "ai" com a separador:

```
>>> song.split('ai')
['The r', 'n in Sp', 'n...']
```

Noteu que el separador no apareix a la llista.

join és el mètode invers d'split. Pren una argument: una llista de cadenes i s'aplica a una cadena separadora. El separador serà afegit entre cada element de la llista al formar la cadena resultant.

```
>>> words = ['crunchy', 'raw', 'unboned', 'real', 'dead', 'frog']
>>> ' '.join(words)
'crunchy raw unboned real dead frog'
>>> '**'.join(words)
'crunchy**raw**unboned**real**dead**frog'
```

Exercicis

EXERCICI 8.1 Dissenyeu un programa que llegeixi 10 nombres enters i els emmagatzemi en una llista. El programa ha d'assegurar que tots els nombres introduïts per l'usuari són positius. Quan un nombre sigui negatiu, el programa ho indicarà amb un missatge i demanarà a l'usuari introduir un nou nombre quantes vegades sigui necessari.

EXERCICI 8.2 Dissenyeu un programa que llegeixi un enter n i escrigui el quadrat dels nombres enters entre 1 i n. Seguiu els següents passos:

- Definiu la funció construeix_llista que, donat n, retorna la llista dels enters de 1 a n. (Pista: estudieu la funció range).
- Definiu la funció modifica_llista que té com a paràmetre una llista d'enters i retorna una llista amb els quadrats de la primera.
- Definiu la funció escriure_llista, que escriu de manera agradable una llista d'enters.
- Finalment, usant les tres funcions anteriors, escriu el programa que es demana.

EXERCICI 8.3 Dissenyeu una funció que elimini d'una llista tots els elements amb índex parell.

EXERCICI 8.4 Dissenyeu una funció tal que, donada una llista de reals en calculi la mitjana.

EXERCICI 8.5 Dissenyeu una funció tal que, donada una llista de punts al pla, calculi quants d'aquests punts són dins del cercle de radi 1 centrat a l'orígen. Assumiu que la llista té una estructura com la següent:

```
[[x0,y0], [x1,y1], [x2,y2],....]
```

EXERCICI 8.6 Dissenyeu la funció llegeix_matriu(m,n) que gestioni la lectura d'una matriu d'm files per n columnes. Deseu la funció en el mòdul `matrius.py`.

EXERCICI 8.7 Dissenyeu la funció producte_matriu_escalar(a,k) que donada una matriu d'm files i n columnes la multipliqui pel nombre k. El resultat de multiplicar una matriu per un nombre consisteix a multiplicar cada element de la matriu per aquest nombre. Deseu la funció en el mòdul `matrius.py`.

EXERCICI 8.8 Dissenyeu una funció que rebi com a paràmetre una matriu quadrada d'enters, representada com a llista de files, i determini si és un quadrat màgic. Una matriu quadrada M és un quadrat màgic si existeix un enter k tal que la suma de tota fila, columna i diagonal d'M és exactament k.

EXERCICI 8.9 Dissenyeu un programa que permeti determinar si una matriu és o no. Una matriu A és prima si la suma dels elements de qualsevol de les seves files és igual a la suma dels elements de qualsevol de les seves columnes.

Utilitzeu la funció llegeix_matriu del mòdul `matrius.py` i dissenyeu una nova funció matriu_prima(m), que determini si la matriu m és prima. Finalment, escriviu l'script `provaMatriuPrima.py` que faci el que es demanava.

EXERCICI 8.10 Dissenyeu un programa que llegeixi dues matrius i calculi la diferència entre la primera i la segona. Utilitzeu la funció llegeix_matriu del mòdul `matrius.py`. Dissenyeu una nova funció matriu_diferencia(a,b), que calculi la diferencia entre la matriu a i la matriu b. Finalment, escriviu l'script `provaDiferenciaMatrius.py` que fa el que es volia.

Exercici 8.11 L'objectiu d'aquest exercici és dissenyar i implementar un programa que permeti analitzar mostres estadístiques. A tal efecte seguirem els següents passos:

- Dissenyeu i implementeu una funció que llegeixi del teclat una seqüència de reals i retorna la llista corresponent. Assumiu que els valors són sempre positius o zero i que el primer negatiu que es llegeix actúa de sentinella.
- Dissenyeu i implementeu una funció tal que donada una llista de reals de mida arbitraria en calcula la suma (sense usar la funció `sum` de `Python`). Documenteu la funció i afegiu els doctests convenients.
- Dissenyeu i implementeu una funció tal que donada una llista de reals de mida arbitrària en calcula el màxim (sense usar la funció `max` de `Python`). Documenteu la funció i afegiu els doctests convenients.
- Dissenyeu i implementeu una funció tal que donada una llista de reals de mida arbitrària en calcula el mínim (sense usar la funció `min` de `Python`). Documenteu la funció i afegiu els doctests convenients.
- Dissenyeu i implementeu una funció tal que donada una llista de reals de mida arbitrària en calcula el recorregut. Documenteu la funció i afegiu els doctests convenients.
- Dissenyeu i implementeu una funció tal que donada una llista de reals de mida arbitrària en calcula la mitjana. Documenteu la funció i afegiu els doctests convenients.
- Dissenyeu i implementeu una funció tal que donada una llista de reals de mida arbitrària en calcula la variància. Documenteu la funció i afegiu els doctests convenients.

A continuació, escriviu un programa que presenti un menú a l'usuari i li permeti executar les següents opcions:

- Llegir dades.
- Calcular el recorregut.
- Calcular la mitja.
- Calcular la variància.
- Acabar l'execució.

Haureu construit el vostre propi calculador estadístic! Com ampliació us proposem que afegiu al calculador les funcionalitats de calcular la moda i la mediana.

Exercici 8.12 Dissenyeu un programa que llegeixi paraules fins a la paraula `'fi'` i després les escrigui ordenades alfabeticament.

Exercici 8.13 Què escriurà el següent script:

```
this = ['Jo', 'soc', 'una', 'llista']
that = ['Jo', 'soc', 'una', 'llista']
print "Test 1: %s" % (id(this) == id(that))
that = this
print "Test 2: %s" % (id(this) == id(that))
```

Exercici 8.14 Definiu una funció amb capçalera **def** `replace(s, old, new)` que substitueixi la lletra `old` per la lletra `new` en la cadena `s` usant els mètodes `split` i `join`. Hauria de superar els següents casos de prova:

```
"""
>>> replace('Manresa', 'a', '4')
'M4nres4'
>>> s = "M'agrada molt ballar!"
>>> replace(s, 'ba', 'ca')
'M'agrada molt cantar!'
>>> replace(s, 'cantar', 'saltar')
'M'agrada molt saltar!'
"""
```

Exercici 8.15 Dissenyeu una funció que indiqui quina posició ocupa una lletra en una paraula sense usar cap mètode de cadenes.

Exercici 8.16 Escriviu un programa que permeti entrar paraules fins a la paraula `'fi'`, un cop finalitzada l'entrada de dades ha d'indicar si hi ha alguna paraula repetida.

Exercici 8.17 Dissenyeu un programa tal que donat una seqüència d'enters acabada en 0 (que actua de sentinella), escrigui els que són superiors a la mitjana.

A tal efecte, us caldrà dissenyar les funcions per fer les tasques que s'especifiquen a continuació:

- Emmagatzemar els elements en una llista. Feu-ho utilitzant una funció que demani a l'usuari la seqüència d'enters i retorni una llista d'enters.
- Calcular la mitjana d'elements d'una llista. Feu-ho utilitzant una funció que donada una llista, retorni la mitjana dels elements de la llista.
- Obtenir els elements superiors a la mitjana. Feu-ho utilitzant una funció que, donada una llista i un valor, retorni en una altra llista els elements superiors a aquest valor.

Exercici 8.18 La cadena de producció d'una fàbrica de components electrònics vol dissenyar unes eines per al control de qualitat de les peces que fabriquen. Per poder sortir al mercat, les peces fabicades han de tenir un pes d'entre 100 i 150 grams. Els programes que es necessiten són aquests:

a) Un programa tal que, donada la seqüència de pesos (en grams) d'una partida de peces, detecti si totes passen el control de qualitat. La seqüència acaba amb un pes fictici -1 que fa de sentinella.

b) Un programa tal que, donada la seqüència de pesos (en grams) determini quantes no passen el control de qualitat. La seqüència acaba amb un pes fictici -1 que fa de sentinella.

Exercici 8.19 Dissenyeu un programa tal que donada una seqüència de paraules acabada amb la paraula `'X'`, escrigui aquelles que tenen la mateixa mida que la paraula més llarga.

A tal efecte heu de considerar el següent:

- Cal dissenyar una funció tal que donada una paraula comprovi si es tracta de la paraula `'X'`. Si és el cas retornarà cert, altrament retornarà fals.
- Cal dissenyar una funció que llegeixi paraules fins arribar a la paraula `'X'` i les retorni en una llista.
- Cal dissenyar una funció tal que donada una llista de paraules, calculi la mida de la paraula més gran.
- Cal dissenyar una funció tal que donada una llista de paraules `l` i un valor `v`, retorni la llista de paraules de l que tenen mida `v`.

Exercici 8.20 Dissenyeu una funció tal que, donada una llista d'enters, retorni cert ssi la llista és creixent. Un cop la tingueu, dissenyeu un programa que llegeixi una llista de 10 enters i determini si aquesta és creixent.

Exercici 8.21 Dissenyeu una funció tal que, donada una llista de cadenes, determini si totes les comencen per la lletra `'a'`. Un cop la tingueu, dissenyeu un programa que vagi llegint cadenes fins a trobar la cadena sentinella `'final'`, i a continuació comprovi si totes les cadenes de la llistan comencen per la lletra `'a'`.

Exercici 8.22 Dissenyeu una funció que, donada una llista de cadenes, determini quantes comencen per la lletra `'a'`.

Exercici 8.23 Dissenyeu una funció que, donada una cadena, retorni cert ssi la cadena és cap-i-cua.

Exercici 8.24 Dissenyeu una funció que llegeixi una cadena i retorni cert ssi conté dues vegades la lletra `'m'`.

Exercici 8.25 Dissenyeu una funció anomenada en_general_positiva tal que, donada una llista de reals, retorni cert ssi hi ha més d'un 50% de valors positius.

Exercici 8.26 Dissenyeu una funció que, donada una llista de reals, retorni una llista amb tres elements enters: el primer ha de correspondre al nombre d'elements negatius de la llista, el segon al nombre d'elements iguals a zero i el tercer, ha de ser el nombre d'elements positius.

9 Mòduls i fitxers

9.1 Mòduls

biblioteca estàndard (standard library)

Un mòdul és un fitxer que conté definicions i sentències `Python` que s'usaran altres programes `Python`. Hi ha molts mòduls de `Python` que formen part de la *biblioteca estàndard* de `Python`.

9.2 Alguns mòduls estàndard

9.2.1 El mòdul "random"

Sovint hi ha aplicacions en les que és necessari usar nombres aleatoris. Algun casos d'aquests són:

1. Quan implementem un joc d'atzar amb el computador i hem de triar una ma de cartes, un número arbitrari o llençar una moneda a l'aire.
2. Quan en un joc cal decidir si és el moment per que aparegui una nau espacial disparant-te.
3. Per simular les condicions de pluja en un model simulat per determinar l'impacte ambiental de la construcció d'una presa.
4. Per encriptar una sessió de treball amb el teu banc via Internet.

`Python` disposa del mòdul `random` per donar suport a aquests tipus de tasques. Podeu mirar-vos la documentació, però de moment aquí teniu alguns exemples del que pot fer:

```
import random

# create a black box object that generates random numbers
rng = random.Random()

dice_throw = rng.randrange(1,7) # return an int, one of 1,2,3,4,5,6
delay_in_seconds = rng.random() * 5.0
```

El mètode randrange retorna un enter en l'interval definit pels seus paràmetres. Usa la mateixa semàntica que la funció range: el límit inferior hi és inclòs i el superior exclòs. Tots els valors tenen la mateixa probabilitat de ser triats, és a dir els resultats segueixen una distribució uniforme.

El mètode random retorna un real en el rang $[0.0, 1.0)$. Sovint el valor que retorna s'escala usant un producte per tal d'obtenir un niu valor en el rang que ens interessa. Els valors que retorna també es distribueixen uniformement. És doncs tant probable que retorni 0.265301 com 0.10101.

El següent exemple mostra com es pot barrejar aleatòriament una llista. Primer es genera una llista d'enters i després és modifica per tal que els elements acabin tenint una disposició aleatòria:

```
cards = list(range(52)) # generate ints 0..51,
                        # representing a pack of cards.
rng.shuffle(cards) # shuffle the pack
```

Els generadors de valors aleatoris es basen en un algoritme determinista i, per tant, predictible i repetible. Per això s'anomenen realment generadors pseudo-aleatoris. La seqüència pseudo-aleatoria depèn d'un valor anomenat *llavor*. Si la llavor és igual, la seqüència és la mateixa.

llavor (seed)

Aquesta capacitat de repetir la mateixa seqüència pseudo-aleatòria és essencial en moltes aplicacions. Per exemple en la generació de casos de proves, atès que és fonamental poder repetir-los exactament igual tantes vegades com sigui necessari.

```
drng = random.Random(123) # create generator with known starting state
```

Aquesta variant en la creació d'un objecte generador d'aleatoris permet triar la llavor, en aquest cas 123 i assegurar-vos que la seqüència generada per aquest objecte sempre serà la mateixa. Si no la trieu, el sistema en triarà una de forma arbitrària.

9.2.2 El mòdul "time"

Una preocupació natural a mida que anem treballant amb programes més grans i sofisticats és respondre la pregunta "és eficient aquest codi?". Una forma d'apreciar l'eficiència és mesurar el temps que triguen les operacions. El mòdul `clock` té una funció anomenada `clock` que pot ajudar. Cada vegada que es crida retorna un real que representa el nombre de segons que han passat des que el programa ha començat a funcionar.

La forma d'usar aquesta funció consisteix en emmagatzemar el valor que torna en un instant, fer un càlcul, cridar-la de nou i observar la diferència entre el resultat de la darrera i la primera crida: és el temps que ha transcorregut entre crides.

Provem un petit exemple. `Python` te una funció predefinida anomenada `sum` que permet sumar els elements d'una llista. Nosaltres també en podem escriure una. Quina funció penseu que serà més ràpida? la predefinida o la nostra? Fem-les-hi sumar la mateixa llista a cadascuna i vegem quan triguen:

```
import time

def do_my_sum(xs):
    sum = 0
    for v in xs:
        sum += v
    return sum

sz = 10000000 # lets have 10 million elements in the list
testdata = range(sz)

t0 = time.clock()
my_result = do_my_sum(testdata)
t1 = time.clock()
print("my_result = {0} (time taken = {1:.4f} seconds)"
                                    .format(my_result, t1-t0))

t2 = time.clock()
their_result = sum(testdata)
t3 = time.clock()
print("their_result = {0} (time taken = {1:.4f} seconds)"
                                .format(their_result, t3-t2))
```

En un portàtil corrent, s'obtenen els següents resultats:

```
my_sum = 49999995000000 (time taken = 1.5567 seconds)
their_sum = 49999995000000 (time taken = 0.9897 seconds)
```

Així doncs la nostra funció és aproximadament un 57% més lenta que la predefinida. No està pas malament sumar deu milions d'element en menys d'un segon!

9.2.3 El mòdul "math"

Com sabeu el mòdul `math` conté la mena de funcions i constants matemàtiques que trobeu típicament en una calculadora científica (sinus, cosinus, tangent, pi, logaritme natural, etc.).

```
>>> import math

>>> math.pi # constant attribute for pi
3.141592653589793
>>> math.e # constant natural log base
2.718281828459045
>>> math.sqrt(2.0) # square root function
1.4142135623730951
>>> math.radians(90) # convert 90 degrees to radians
1.5707963267948966
>>> math.sin(math.radians(90)) # find sin of 90 degrees.
1.0
>>> math.asin(1.0) * 2 # Double arcsin of 1.0 to get pi
3.141592653589793
```

Noteu que com en la majoria de llenguatges de programació, els angles s'expressent en radians. Això no és problema per que també ofereix funcions de conversió entre graus i radians.

La majoria de funcions del mòdul són funcions pures. Això no era així en els dos mòduls que hem vist anteriorment com a exemples, que tenien una estructura una mica més complicada.

9.3 Creació de mòduls

La creació de mòduls no és una tasca misteriosa. De fet, separar diferents parts d'un programa en mòduls és una manera molt convenient de desenvolupar programari. Tot allò que cal per crear un mòdul és un fitxer de text amb un nom acabat amb `.py`:

```
# seqtools.py
#
def remove_at(pos, seq):
    return seq[:pos] + seq[pos+1:]
```

Ara podem utilitzar el nostre mòdul tant en scripts com en l'intèrpret interactiu de `Python`. Per usar el nostre mòdul, primer hem d'importar-lo. Hi ha dues maneres fer-ho:

```
>>> from seqtools import remove_at
>>> s = "A string!"
>>> remove_at(4, s)
'A sting!'
```

O bé:

```
>>> import seqtools
>>> s = "A string!"
>>> seqtools.remove_at(4, s)
'A sting!'
```

En el primer exemple, cridem la funció remove_at com ho hem fet fins ara. En el segon exemple escrivim el nom del mòdul i un punt (.) abans del nom de funció.

sentència d'importació (import statement)

Observa que en cap cas incloem el .py del nom del fitxer quan importem el mòdul. Python s'espera que els noms dels fitxers que contenen mòduls acabin sempre en .py, i, per tant, el sufix .py no s'ha d'incloure en la *sentència d'importació*.

L'ús de mòduls permet dividir programes grans en parts d'una mida més fàcil de gestionar i, alhora, mantenir agrupades les parts relacionades.

9.4 Àmbits

àmbit (namespace)

Un *àmbit* és un contenidor sintàctic que possibilita que usem el mateix nom (de paràmetre, de variable, de funció, etc.) en funcions o mòduls diferents.

Cada mòdul té associat de manera automàtica el seu propi àmbit, que és diferent de tota las resta d'àmbits del programa. Així podem utilitzar el mateix nom en més d'un mòdul sense que hi hagi un problema de col·lisió d'identificadors.

```
# module1.py

question = "What is the meaning of life, the Universe, and everything?"
answer = 42
```

```
# module2.py

question = "What is your quest?"
answer = "To seek the holy grail."
```

Ara podem importar ambdós mòduls i accedir a les variables question i answer de cadascun d'ells sense conflictes:

```
>>> import module1
>>> import module2
>>> print module1.question
What is the meaning of life, the Universe, and everything?
>>> print module2.question
What is your quest?
>>> print module1.answer
42
>>> print module2.answer
To seek the holy grail.
>>>
```

col·lisió de noms (naming collision)

Si haguéssim utilitzat **from** module1 **import** * i **from** module2 **import** * en comptes d'**import**, tindríem una *col·lisió de noms* i ens resultaria impossible accedir a les variables question i answer del mòdul module1.

Com ja havíem vist, les funcions també defineixen el seu propi àmbit i no hi ha cap problema en tenir paràmetres o variables locals amb el mateix nom:

```
def f():
    n = 7
    print "printing n inside of f: %d" % n

def g():
    n = 42
    print "printing n inside of g: %d" % n

n = 11
print "printing n before calling f: %d" % n
f()
print "printing n after calling f: %d" % n
g()
print "printing n after calling g: %d" % n
```

L'execució d'aquest programa produeix la sortida següent:

```
printing n before calling f: 11
printing n inside of f: 7
printing n after calling f: 11
printing n inside of g: 42
printing n after calling g: 11
```

No hi ha col·lisió entre les tres variables `n` perquè cadascuna pertany a un àmbit diferent.

Els àmbits permeten treballar a diversos programadors en el mateix projecte evitant el problema de la col·lisió de noms.

9.5 Atributs i l'operador punt

Les variables definides en un mòdul s'anomenen *atributs* del mòdul. S'hi accedeix utilitzant l'*operador punt* (.). A l'atribut `question` de `module1` i `module2` s'hi accedeix utilitzant `module1.question` i `module2.questio`

atributs (attributes)

operador punt (dot operator)

Els mòduls poden contenir tant funcions com també atributs. A unes i altres s'hi accedeix amb l'operador punt de la mateixa manera. Per exemple, `seqtools.remove_at` es refereix a la funció `remove_at` del mòdul `seqtools`. El mòdul `math` que hem vist abans és un bon exemple per estudiar.

9.6 Llegir i escriure fitxers de text

Mentre un programa s'està executant, les seves dades s'emmagatzemen en la memòria d'accés directe (RAM). La memòria RAM és ràpida i barata, però també és *volàtil*, la qual cosa significa que quan el programa finalitza, o l'ordinador s'atura, les dades de la memòria RAM desapareixen. Per tal de disposar de les dades la propera vegada que engeguis el teu l'ordinador i executis el teu programa, has d'escriure les dades en un dispositiu d'emmagatzematge *no-volàtil* o permanent, com un disc dur, una memòria USB, o un CD-RW.

volàtil (volatile)

no-volàtil (non-volatile)

En els dispositius d'emmagatzematge no-volàtils les dades s'emmagatzemen en unes entitats que s'anomenen *fitxers*. Els fitxers s'identifiquen per un nom. Els programes poden conservar informació entre diferents execucions llegint i escrivint les dades en fitxers.

fitxers (files)

Treballar amb fitxers s'assembla a treballar amb una llibreta. Per tal d'utilitzar una llibreta, l'has d'obrir. Quan has acabat, l'has de tancar. Mentre tens la llibreta oberta, tant pots escriure-hi com llegir-hi. En qualsevol dels dos casos, saps en quin punt de la llibreta et trobes. Pots llegir la llibreta sencera en el seu ordre natural o pots saltar-te'n parts.

Tot això també s'aplica als fitxers. Per tal d'obrir un fitxer, cal especificar el seu nom i indicar si vols llegir-hi o escriure-hi.

Obrir un fitxer crea un objecte file. En aquest exemple, la variable myfile conté el nou objecte file.

```
>>> myfile = open('test.dat', 'w')
>>> print myfile
<open file 'test.dat', mode 'w' at 0x2aaaaab80cd8>
```

mode (mode)

La funció open rep dos arguments. El primer és el nom del fitxer, i el segon és el *mode*. El mode 'w' significa que estem obrint el fitxer per escriptura.

Si no hi ha cap fitxer anomenat test.dat, es crearà automàticament. Si ja n'hi ha un, es reemplaçarà pel fitxer que estem escrivint.

Quan escrivim un objecte file, veiem el nom del fitxer, el mode, i la ubicació de l'objecte en memòria.

Per escriure dades en el fitxer invoquem el mètode write de l'objecte file:

```
>>> myfile.write("Now is the time")
>>> myfile.write("to close the file")
```

Tancant el fitxer diem al sistema que ja hem acabat d'escriure i el fitxer queda disponible per poder-lo llegir:

```
>>> myfile.close()
```

Ara podem obrir el fitxer una altra vegada, aquest cop en mode lectura, i llegir els continguts en una cadena. Aquest cop, l'argument mode és 'r', lectura:

```
>>> myfile = open('test.dat', 'r')
```

Si intentem obrir un fitxer que no existeix, es produeix un error:

```
>>> myfile = open('test.cat', 'r')
IOError: [Errno 2] No such file or directory: 'test.cat'
```

Com és d'esperar, el mètode read llegeix dades del fitxer. Sense arguments, llegeix els continguts sencers del fitxer en una única cadena:

```
>>> text = myfile.read()
>>> print text
Now is the timeto close the file
```

No hi ha cap espai entre *time* i *to* perquè no vam escriure cap espai entre les dues cadenes.

read també pot rebre un argument que indica quants caràcters ha de llegir:

```
>>> myfile = open('test.dat', 'r')
>>> print myfile.read(5)
Now i
```

Si no queden prou caràcters al fitxer, read retorna els caràcters restants. Quan arribem al final del fitxer, read retorna la cadena buida:

```
>>> print myfile.read(1000006)
s the timeto close the file
>>> print myfile.read()

>>>
```

La funció següent copia un fitxer. Va llegint i escrivint en blocs de cinquanta caràcters alhora. El primer argument és el nom del fitxer original; el segon és el nom del fitxer nou:

```
def copy_file(oldfile, newfile):
    infile = open(oldfile, 'r')
    outfile = open(newfile, 'w')
    while True:
        text = infile.read(50)
        if text == "":
            break
        outfile.write(text)
    infile.close()
    outfile.close()
    return
```

Aquesta funció itera, llegint 50 caràcters d'infile i escrivint els mateixos 50 caràcters a outfile fins que s'assoleix el final d'infile, en aquest punt la cadena text és buida i s'executa la sentència **break**.

9.7 Fitxers de text

fitxer de text (text file)

Un *fitxer de text* és un fitxer que conté caràcters imprimibles i espais en blanc, organitzats en línies separades per caràcters salt de línia. Com que Python està expressament dissenyat per processar fitxers de text, proporciona mètodes que fan la feina fàcil.

Per tal de demostrar-ho, crearem un fitxer de text amb tres línies de text separat per salts de línia:

```
>>> outfile = open("test.dat","w")
>>> outfile.write("line one\nline two\nline three\n")
>>> outfile.close()
```

El mètode readline llegeix tots els caràcters fins al proper caràcter salt de línia inclòs:

```
>>> infile = open("test.dat","r")
>>> print infile.readline()
line one

>>>
```

El mètode readlines retorna totes les línies que queden com a llista de cadenes:

```
>>> print infile.readlines()
['line two\012', 'line three\012']
```

En aquest cas, la sortida és en format de llista, la qual cosa significa que les cadenes apareixen entre cometes i el caràcter salt de línia apareix com la seqüència d'escapada \012.

Al final del fitxer, readline retorna la cadena buida i readlines retorna la llista buida:

```
>>> print infile.readline()

>>> print infile.readlines()
[]
```

El següent exemple és un programa que processa el fitxer línia a línia. La funció filter fa una còpia de oldfile, ometent qualsevol línia que comenci amb #:

```
def filter(oldfile, newfile):
    infile = open(oldfile, 'r')
    outfile = open(newfile, 'w')
    while True:
        text = infile.readline()
        if text == "":
            break
        if text[0] == '#':
            continue
        outfile.write(text)
    infile.close()
    outfile.close()
    return
```

La sentència **continue** acaba la iteració actual del bucle, però deixa que el bucle continuï iterant. Quan es troba una sentència **continue**, el flux d'execució s'interromp i salta a la capçalera del bucle, comprova la condició, i procedeix conseqüentment.

Així, si text és la cadena buida, el bucle acaba. Si el primer caràcter de text és un sostingut, el flux d'execució va a la part superior del bucle. Només si ambdues condicions fallen copiem text al fitxer nou.

9.8 Directoris

Els fitxers emmagatzemats en dispositius d'emmagatzematge no-volàtils s'organitzen en una estructura de dades anomenada *sistema de fitxers*. Els sistemes de fitxers estan constituïts per fitxers i *directoris*, que són contenidors tant de fitxers com d'altres directoris.

sistema de fitxers (file system)

directoris (directori-es)

Quan crees un fitxer nou obrint-lo i escrivint-hi, el nou fitxer va al *directori de treball* actual, onsevulga que eres quan vas executar el programa. De manera semblant, quan obres un fitxer per lectura, Python el busca en el directori de treball actual.

directori de treball (working directory)

Si vols obrir un fitxer en algun altre lloc, has d'especificar el *camí* al fitxer, que aproximadament és el nom del directori en on es troba el fitxer:

camí (path)

```
>>> wordsfile = open('/usr/share/dict/words', 'r')
>>> wordlist = wordsfile.readlines()
>>> print wordlist[:6]
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

Aquest exemple obre un fitxer de nom words que està en un directori anomenat dict, el qual està dins de share, el qual està dins de usr, el qual està dins del directori de nivell superior del sistema, anomenat /. Llavors llegeix cada línia en una llista utilitzant el mètode readlines, i escriu els primers 5 elements d'aquesta llista.

No pots utilitzar el caràcter / dins del nom d'un fitxer perquè està reservat com a *delimitador* entre

delimitador (delimiter)

noms de directoris i de fitxers.

El fitxer /usr/share/dict/words hauria d'existir en qualsevol sistema basat en Unix, i conté una llista de paraules en ordre alfabètic.

9.9 Comptatge de lletres

La funció ord retorna la representació com a enter d'un caràcter:

```
>>> ord('a')
97
>>> ord('A')
65
>>>
```

Aquest exemple explica perquè `'Apple'< 'apple'` s'avalua a True. La funció chr és la inversa de ord. Donat un enter com a argument retorna la seva representació com a caràcter:

```
>>> for i in range(65, 71):
... print chr(i)
...
A
B
C
D
E
F
>>>
```

El programa següent, countletters.py, compta quantes vegades ocorre cada caràcter en el llibre *Alice in Wonderland* assumint que el llibre està escrit en el fitxer `alice_in_wonderland.txt`:

```
#
# countletters.py
#

def display(i):
    if i == 10: return 'LF'
    if i == 13: return 'CR'
    if i == 32: return 'SPACE'
    return chr(i)

infile = open('alice_in_wonderland.txt', 'r')
text = infile.read()
infile.close()

counts = 128 * [0]

for letter in text:
    counts[ord(letter)] += 1

outfile = open('alice_counts.dat', 'w')
outfile.write("%-12s%s\n" % ("Character", "Count"))
```

```
outfile.write("=================\n")

for i in range(len(counts)):
    if counts[i]:
        outfile.write("%-12s%d\n" % (display(i), counts[i]))

outfile.close()
```

Executa aquest programa i mira l'aspecte del fitxer resultant que ha generat utilitzat un editor de texts. L'anàlisi del programa es farà en els exercicis del final del capítol.

9.10 El mòdul "sys" i l'atribut "argv"

El mòdul sys conté funcions i variables que proporcionen accés a l'entorn en el qual s'executa l'intèrpret de Python.

L'exemple següent mostra els valors d'algunes d'aquestes variables en un dels nostres sistemes:

```
>>> import sys
>>> sys.platform
'linux2'
>>> sys.path
['', '/home/jelkner/lib/python', '/usr/lib/python25.zip', '/usr/lib/python2.5',
'/usr/lib/python2.5/plat-linux2', '/usr/lib/python2.5/lib-tk',
'/usr/lib/python2.5/lib-dynload', '/usr/local/lib/python2.5/site-packages',
'/usr/lib/python2.5/site-packages', '/usr/lib/python2.5/site-packages/Numeric',
'/usr/lib/python2.5/site-packages/gst-0.10',
'/var/lib/python-support/python2.5', '/usr/lib/python2.5/site-packages/gtk-2.0',
'/var/lib/python-support/python2.5/gtk-2.0']
>>> sys.version
'2.5.1 (r251:54863, Mar 7 2008, 04:10:12) \n[GCC 4.1.3 20070929 (prerelease)
(Ubuntu 4.1.2-16ubuntu2)]'
>>>
```

Jython (empty)

Executant *Jython* —una implementació de Python diferent— en la mateixa màquina produeix valors diferents per les mateixes variables:

```
>>> import sys
>>> sys.platform
'java1.6.0_03'
>>> sys.path
['', '/home/jelkner/.', '/usr/share/jython/Lib', '/usr/share/jython/Lib-cpython']
>>> sys.version
'2.1'
>>>
```

línia d'ordres (command line)

arguments de línia d'ordres (command line arguments)

Naturalment, sobre el vostre computador els resultats seran diferents.

La variable argv conté una llista de cadenes obtingudes de la *línia d'ordres* quan s'executa un script de Python. Aquests *arguments de línia d'ordres* s'utilitzen per passar informació a un programa en el moment d'invocar-lo.

```
#
# demo_argv.py
#
import sys

print sys.argv
```

Executant aquest programa des de la línia d’ordres de unix exemplifica el significat de sys.argv:

```
$ python demo_argv.py this and that 1 2 3
['demo_argv.py', 'this', 'and', 'that', '1', '2', '3']
$
```

argv és una llista de cadenes. Tingues en compte que el primer element és el nom del programa. Els arguments s’escriuen separats per espais en blanc, i la llista resultat és la mateixa que s’obtindria cridant al mètode split de les cadenes. Si algun argument ha de contenir espais en blanc, cal escriure’l entre cometes:

```
$ python demo_argv.py "this and" that "1 2" 3
['demo_argv.py', 'this and', 'that', '1 2', '3']
$
```

Usant argv es poden escriure programes útils que agafen la seva entrada directament de la línia d’ordres. Per exemple, aquí teniu un programa que troba la suma d’una sèrie de números:

```
#
# sum.py
#
from sys import argv

nums = argv[1:]

for index, value in enumerate(nums):
    nums[index] = float(value)

print sum(nums)
```

En aquest programa utilitzem l’estil d’importar **from** <module> **import** <attribute>, així que argv s’incorpora a l’espai de noms principal del mòdul.

Executem ara el programa des de la línia d’ordres de la següent manera:

```
$ python sum.py 3 4 5 11
23
$ python sum.py 3.5 5 11 100
119.5
```

Exercicis

EXERCICI 9.1 Les variables locals d’una funció succeix que estan definides en un àmbit restringit al cos de la funció i això permet que els seus noms no entrin en conflicte amb variables d’altres àmbits. A banda d’aquesta propietat, però, tenen una altra característica fonamental. Quina?

Exercici 9.2 Creeu un mòdul anomenat `modul1`. En el mòdul, definiu els atributs `edat` amb valor inicial la vostra edat, i `any`, amb valor incial l'any actual. Creeu un altre mòdul anomenat `modul2`. Afegiu els atributs `edat` amb valor inicial 0, i `any` amb valor l'any que vau nèixer. Ara creeu un script anomenat `test_espai_de_noms.py`. Importeu-hi els dos mòduls anteriors i escriviu la següent sentència:

```
print (modul2.edat - modul1.edat) == (modul2.any - modul1.any)
```

Executeu l'script `test_espai_de_noms.py` i expliqueu quin és el resultat que obteniu i per què.

Exercici 9.3 Escriviu un script anomenat `mitjana.py` que prengui una seqüència de nombres de la línia de comandes i retorni la mitjana dels seus valors. Una sessió del vostre programa funcionant amb la mateixa entrada ha de generar la mateixa sortida que a continuació:

```
$ python mitjana.py 3 4
3.5
$ python mitjana.py 3 4 5
4.0
$ python mitjana.py 11 15 94.5 22
35.625
```

Exercici 9.4 Escriviu un programa anomenat `mediana.py` que prengui una seqüència de nombres de la línia de comandes i retorni la mediana dels seus valors. Una sessió del vostre programa funcionant amb la mateixa entrada ha de generar la mateixa sortida que a continuació:

```
$ python mediana.py 19 85 121
85
$ python mediana.py 11 15 16 22
15.5
```

Exercici 9.5 `unsorted_fruits.txt` conté una llista de 26 fruites, cada una d'elles amb un nom que comença per una lletra diferent de l'abecedari. Escriviu un programa anomenat `ordena_fruites.py` que llegeixi les fruites del fitxer `unsorted_fruits.py` i els escrigui en ordre alfabètic en un fitxer anomenat `fruites_ordenades.txt`.

Exercici 9.6 Escriviu un programa en un script anomenat `tres_linies.py` que llegeixi les fruites del fitxer `unsorted_fruits.txt` i les escrigui en grups de tres (tres fruites / línia en blanc / tres fruites / linia en blanc / etc.) en un fitxer anomenat `paquets_fruita.txt`.

Exercici 9.7 Escriviu un programa en un script anomenat `canvia_lletres.py` que prengui una lletra per la línia de comandes, llegeixi les fruites del fitxer `unsorted_fuits.txt` i les escrigui en un fitxer anomenat `canvi_vocals_fruites.txt`, canviant totes les vocals que trobi per la lletra passada per la línia de comandes.

Exercici 9.8 Dissenyeu un programa que obtingui els 100 primers nombres primers i els guardi en un fitxer de text anomenat `primers.txt`.

Exercici 9.9 Dissenyeu un programa que donada una llista d'enters proporcionada pel teclat en calculi el nombre més repetit de la llista.

Exercici 9.10 Dissenyeu una funció anomenada `data_llarga` que retorni la data en format textual. Per exemple, si l'entrada és `"11/11/2009"`, hauria de tornar `"11 de novembre de 2009"`.

Exercici 9.11 Dissenyeu un programa que compti el nombre de caràcters d'un fitxer de text, incloent els salts de línia. El nom del fitxer es demana a l'usuari per teclat.

10 Tuples i mutabilitat

10.1 Tuples i mutabilitat

Fins ara, hem vist dos tipus compostos: les cadenes, que estan compostes de caràcters, i les llistes, que estan compostes d'elements de qualsevol tipus. Una de les diferències que hem apreciat rau en el fet que els elements d'una llista poden ser modificats mentre que els caràcters d'una cadena no. En altres paraules, les cadenes són immutables i les llistes mutables.

Un tuple, com una llista, és una seqüència d'elements de qualsevol tipus. Contràriament a les llistes, els tuple són immutables. Des del punt de vista sintàctic un tuple és una seqüència de valors separats per comes:

```
>>> tup = 2, 4, 6, 8, 10
```

Tot i que no és necessari, és molt habitual tancar els tuples entre parèntesis:

```
>>> tup = (2, 4, 6, 8, 10)
```

Per crear un tuple amb un únic element, cal incloure la coma al final:

```
>>> tup = (5,)
>>> type(tup)
<type 'tuple'>
```

Sense la coma, `Python` tractaria el (5) com si fos un enter entre parèntesis:

```
>>> tup = (5)
>>> type(tup)
<type 'int'>
```

Deixant al marge els aspectes sintàctics, els tuples admeten les mateixes operacions de seqüència que les cadenes i llistes. L'operador d'indexat, per exemple, permet accedir a un element d'un tuple:

```
>>> tup = ('a', 'b', 'c', 'd', 'e')
>>> tup[0]
'a'
```

i l'operador de segment permet accedir a un rang d'elements:

```
>>> tup[1:3]
('b', 'c')
```

això no obstant, si provem d'assignar un valor a un dels elements d'un tuple per modificar-lo obtindrem un error. No pot ser d'altra manera si tenim en compte que són immutables:

```
>>> tup[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Naturalment, tot i que no podem modificar els elements d'un tuple sempre podem substituir-lo per un nou tuple:

```
>>> tup = ('X',) + tup[1:]
>>> tup
('X', 'b', 'c', 'd', 'e')
```

de forma alternativa també el podem convertir en una llista, modificar-lo, i convertir-lo de nou en un tuple:

```
>>> tup = ('X', 'b', 'c', 'd', 'e')
>>> tup = list(tup)
>>> tup
['X', 'b', 'c', 'd', 'e']
>>> tup[0] = 'a'
>>> tup = tuple(tup)
>>> tup
('a', 'b', 'c', 'd', 'e')
```

10.2 Assignació de tuples

De quan en quan és útil intercanviar el valor de dues variables. Amb sentències d'assignació convencionals cal emprar una variable temporal. Per exemple, per intercanviar els valors d'`a` i de `b` fem:

```
temp = a
a = b
b = temp
```

Aquesta solució és una mica lletja. `Python` ofereix una forma més neta basada en l'assignació de tuples:

```
a, b = b, a
```

en aquesta assignació, l'esquerra és un tuple de variables i la dreta un tuple de valors. Cada valor s'assigna a la variable corresponent. Com és habitual en l'assignació, les expressions de la dreta s'avaluen abans de fer l'assignació. Això fa de l'assignació de tuples una sentència molt versàtil.

Naturalment, el nombre de variables a l'esquerra i el d'expressions a la dreta han de coincidir:

```
>>> a, b, c, d = 1, 2, 3
ValueError: need more than 3 values to unpack
```

10.3 Tuples com a valors de retorn

Les funcions poden retornar tuples. Per exemple, podem escriure una funció que intercanvia dos paràmetres:

```
def swap(x, y):
    return y, x
```

i ara la podem invocar i assignar el valor que retorna a un tuple de dues variables:

```
a, b = swap(a, b)
```

L'exemple precedent, la funció no suposa una gran avantatge. De fet existeix el risc de voler encapsular en excés l'intercanvi de valors i caure en el següent error:

```
def swap(x, y): # incorrecte
    x, y = y, x
```

si cridem aquesta funció fent:

```
swap(a, b)
```

aleshores a i x són àlies del mateix valor. La modificació d'x dins de la funció swap fa que x es refereixi a un valor diferent però no té cap efecte en a. De forma paral·lela, modificar y no té cap efecte sobre b. Aquesta funció s'executa sense que hi hagi cap missatge d'error, però no fa el que es volia. És un exemple d'*error semàntic*.

error semàntic (semantic error)

10.4 De nou les accions i funcions pures

A l'apartat 8.15 es presenten les funcions pures i les accions en relació a les llistes. Com que els tuples són immutables no podem definir accions en les que els únics paràmetres siguin tuples.

Aquí teniu una acció que insereix un nou valor al mig d'una llista:

```
#
# seqtools.py
#
def insert_in_middle(val, lst):
    middle = len(lst)/2
    lst[middle:middle] = [val]
```

Si provem d'executar-la usant l'intèrpret interactiu per veure com funciona succeirà el següent:

```
>>> from seqtools import *
>>> my_list = ['a', 'b', 'd', 'e']
>>> insert_in_middle('c', my_list)
>>> my_list
['a', 'b', 'c', 'd', 'e']
```

si ho provem amb un tuple, però, observarem el següent error:

```
>>> my_tuple = ('a', 'b', 'd', 'e')
>>> insert_in_middle('c', my_tuple)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "seqtools.py", line 7, in insert_in_middle
    lst[middle:middle] = [val]
TypeError: 'tuple' object does not support item assignment
>>>
```

El problema és que els tuples són immutables i no suporten l'assignació a un segment. Una solució senzilla consisteix en transformar insert_in_middle en una funció pura:

```
def insert_in_middle(val, tup):
    middle = len(tup)/2
    return tup[:middle] + (val,) + tup[middle:]
```

aquesta versió ara funciona per tuples però no ho fa per a llistes i cadenes. Si volem una implementació que funcioni per a tots els tipus de natura seqüencial necessitem encapsular el valor de retorn en el tipus correcte. El truc rau en la funció type. Definim una funció auxiliar que faci la feina:

```
def encapsulate(val, seq):
    if type(seq) == type(""):
        return str(val)
    if type(seq) == type([]):
        return [val]
    return (val,)
```

ara podem reescriure insert_in_middle usant la funció encapsulate:

```
def insert_in_middle(val, seq):
    middle = len(seq)/2
    return seq[:middle] + encapsulate(val, seq) + seq[middle:]
```

Les dues darreres versions de insert_in_middle són funcions pures: no tenen efectes laterals. Si afegim les funcions insert_in_middle i encapsulate al mòdul `seqtool.py` el podem provar fent:

```
>>> from seqtools import *
>>> my_string = 'abde'
>>> my_list = ['a', 'b', 'd', 'e']
>>> my_tuple = ('a', 'b', 'd', 'e')
>>> insert_in_middle('c', my_string)
'abcde'
>>> insert_in_middle('c', my_list)
['a', 'b', 'c', 'd', 'e']
>>> insert_in_middle('c', my_tuple)
('a', 'b', 'c', 'd', 'e')
>>> my_string
'abde'
```

els valors de my_string, my_list i my_tuple no han canviat. Si volem que insert_in_middle les canvii cal assignar el valor de retorn de nou a la mateixa variable:

```
>>> my_string = insert_in_middle('c', my_string)
>>> my_string
'abcde'
```

10.5 Excepcions

excepció (exception)

volcat de pila (traceback)

Quan s'esdevé un error d'execució es crea una *excepció*. El programa s'atura en aquest punt i `Python` escriu un *volcat de pila* que acaba amb l'excepció ocorreguda.

Per exemple, si dividim per zero es crea una excepció:

```
>>> print 55/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>>
```

El mateix succeeix si accedim a un element inexistent d'una llista:

```
>>> a = []
>>> print a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

o si provem d'assignar a un element d'un tuple:

```
>>> tup = ('a', 'b', 'd', 'd')
>>> tup[2] = 'c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

En tots els casos el missatge d'error de la darrera línia té dues parts:

1. El tipus d'error —abans dels dos punts.
2. Aspectes específics de l'error —després dels dos punts.

Algunes vegades volem executar una operació que pot causar una excepció però no desitgem que el programa s'aturi. Podem tractar l'excepció usant les sentències **try** i **except**.

Per exemple, imaginem que demanem a l'usuari el nom d'un fitxer i volem després obrir el fitxer. Si el fitxer no existeix no volem que el programa s'aturi. Volem tractar l'excepció:

```
filename = raw_input('Enter a file name: ')
try:
    f = open (filename, "r")
except:
    print 'There is no file named', filename
```

La sentència **try** executa les sentències en el primer bloc. Si no es produeix cap excepció durant la seva execució, el bloc corresponents a la sentència **except** s'ignora. D'altra banda, si es produeix una excepció durant l'execució del primer bloc, aleshores s'executaran de manera automàtica les sentències del bloc **except**.

Podem encapsular aquesta capacitat en una funció. exists és un predicat amb un paràmetre que és un nom de fitxer i ens indica si el fitxer existeix o no:

```
def exists(filename):
    try:
        f = open(filename)
        f.close()
        return True
    except:
        return False
```

aixecar (raise)

Si un programa detecta uns situació d'error es pot provocar *aixecar* una excepció. Aquí teniu un exemple que obté un enter de l'usuari i comprova que és positiu:

```
#
# learn_exceptions.py
#
def get_age():
```

```
    age = input('Please enter your age: ')
    if age < 0:
        raise ValueError, '%s is not a valid age' % age
    return age
```

La sentència **raise** pren dos arguments: el tipus d'excepció i una cadena que explica el problema específic. ValueError és un tipus d'excepció predefinit que s'adiu al tipus d'error que es produeix en aquesta funció. El llistat complet el podeu trobar en l'apartat *Built-in Exceptions* de la *Python Library Reference*.

```
>>> get_age()
Please enter your age: 42
42
>>> get_age()
Please enter your age: -2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "learn_exceptions.py", line 4, in get_age
    raise ValueError, '%s is not a valid age' % age
ValueError: -2 is not a valid age
>>>
```

el missatge d'error inclou el tipus d'excepció i la informació addicional que s'indica en la sentència **raise**.

llista definida per intensió (list comprehension)

10.6 Llistes definides per intensió

Una *llista definida per intensió* és una construcció sintàctica que permet la creació de llistes a partir d'altres llistes usant una notació compacta i de natura matemàtica. La idea és adoptar la notació matemàtica clàssica per descriure conjunts d'elements que permet, per exemple, descriure el conjunt dels quatre primers números naturals com $T = \{x \mid 0 < x < 5\}$ o bé el dels quatre primers quadrats perfectes com $P = \{q^2 \mid q \in T\}$.

Fixeu-vos en els següents exemples:

```
>>> numbers = [1, 2, 3, 4]
>>> [x**2 for x in numbers]
[1, 4, 9, 16]
>>> [x**2 for x in numbers if x**2 > 8]
[9, 16]
>>> [(x, x**2, x**3) for x in numbers]
[(1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64)]
>>> files = ['bin', 'Data', 'Desktop', '.bashrc', '.ssh', '.vimrc']
>>> [name for name in files if name[0] != '.']
['bin', 'Data', 'Desktop']
>>> letters = ['a', 'b', 'c']
>>> [n*letter for n in numbers for letter in letters]
['a', 'b', 'c', 'aa', 'bb', 'cc', 'aaa', 'bbb', 'ccc', 'aaaa', 'bbbb', 'cccc']
>>>
```

La sintaxi general d'una llista definida per intensió és la que segueix:

```
[expr for item1 in seq1 for item2 in seq2 ... for itemx in seqx if condition]
```

aquesta intensió té el mateix efecte que:

```
output_sequence = []
for item1 in seq1:
    for item2 in seq2:
        ...
            for itemx in seqx:
                if condition:
                    output_sequence.append(expr)
```

Com es pot observar, les definicions de llistes per intensió són molt més compactes.

Exercicis

EXERCICI 10.1 Dissenyeu una funció tal que donades una primera tupla de cadenes de caràcters on cadascuna d'elles representa el nom i cognom d'una persona i una llista on cada element d'aquesta conté el nom d'una persona com a primer element, el cognom com a segon element i la població on resideix com a tercer element, retorni una tupla on només hi hagi els noms de les persones de les quals tenim informació de la població on resideix.

EXERCICI 10.2 Escriviu sengles expressions, basades en llistes definides per intensió, que calculin:

a) Donada una llista de reals, la llista dels seus cubs.

b) Donada una llista de mots l, la llista dels mots d'l que comencen amb majúscula.

c) Donades dues llistes d'enters l i m, la llista que conté tots els possibles productes entre un element d'l i un d'm amb la condició que el producte sigui superior a 20.

EXERCICI 10.3 Escriviu el codi iteratiu equivalen a cadascuna de les següents expressions:

a) `[name for name in files if name[0] != '.']`

b) `[x**3 for x in nums]`

c) `[x**2 for x in nums if x**2 !=4]`

d) `[(x,y) for x in nums for y in nums]`

e) `[llista1[i]*llista2[i] for i in range(len(llista1))]`

EXERCICI 10.4 Definiu una funció tal que donada una llista de números reals, retorni aquesta mateixa llista però canviant l'element de la posició 0 per l'element de la posició 1, l'element de la posició 2 per l'element de la posició 3,... i així successivament.

EXERCICI 10.5 Un departament universitari vol gestionar l'organització del personal que treballa en un projecte. El projecte s'estructura en base a equips i cada equip el formen un conjunt de persones. Una persona pot pertànyer a més d'un equip. Cada equip té un nom i té també un pressupost assignat.

El programa de gestió ha de presentar un menú a l'usuari que permeti fer les següents accions:

- Afegir una nova persona.

  En cap cas s'ha de poder afegir dues vegades la mateixa persona.

- Afegir un nou equip.

  En cap cas s'ha de poder afegiu un equip dues vegades. En donar d'alta un equip, cal indicar el seu nom, pressupost i persones en formen part. Les persones que en formen part han d'haver estat prèviament donades d'alta.

- Llistar les persones.
- Llistar els equips i les persones que els formen.
- Esborrar una persona.

  Cal tenir en compte que s'ha d'esborrar la persona d'aquells equips dels que forma part. Si un equip es queda sense persones, cal donar-lo de baixa.

- Esborrar un equip.

Per representar aquesta informació cal emprar la següent estructura:

- El conjunt de tota la informació, que anomenarem "base de dades", es representa mitjançant un tuple `(p,g)` on `p` és una llista de noms de persones i `q` és una llista de grups.
- Un grup es representa mitjançant un tuple `(ng, pr, lp)` en el que `ng` representa el nom del grup, `pr` representa el pressupost i `lp` la llista de les persones que formen part del grup.

Exercici 10.6 Afegiu a l'exercici anterior la opció de llegir i escriure la informació en un fitxer. Ha d'apareixer una nova opció al menú que digui llegir les dades d'un fitxer i una altra opció que escrigui les dades en un fitxer. En tots dos casos cal indicar el nom del fitxer on ho guardarem.

Exercici 10.7 Dissenyeu una funció que, donat el nom d'un fitxer de text, retorni una tupla formada pel nom del fitxer, la lletra que més vegades apareix en el fitxer i el nombre de vegades que hi apareix.

Exercici 10.8 Dissenyeu una funció que, donat el nom d'un fitxer, retorni una tupla formada pel nom del fitxer, la frase més llarga del fitxer i la frase més curta.

Exercici 10.9 Dissenyeu una funció tal que, donada una llista d'elements `l` i un enter `n`, retorni una tupla amb els elements de la llista `l` que ocupin posicions múltiples de `n`.

Exercici 10.10 Donada la llista d'enters `[1, 2, 3, 4]`, useu una intensió per tal de crear una nova llista que contingui, per cada element de la llista original, una tupla amb l'enter, la conversió a real de l'enter i la conversió a cadena de l'enter. Així, la llista resultant hauria de ser `[(1,1.0,"1"), (2,2.0,"2"), (3,3.0,"3"), (4,4.0,"4")]`.

Exercici 10.11 Donades les llistes `numbers = [1, 2, 3, 4]`, `paraules = ["hola", "adeu"]` i `l = [["no", "nop"], ["si", "sip"`
què retorna la següent expressió?

```
[(n*elem)+value for n in numbers for elem in paraules
                for subl in l for value in subl
                if ("h" in elem and len(value)%2==0)]
```

Raoneu el resultat sense fer servir l'intèrpret de `Python`.

11 Diccionaris

Tots els tipus de dades composts que hem estudiat en detall fins ara —cadenes, llistes i tuples— s'agrupen sota el paraigües del que anomenem seqüències. Intuïtivament, un tipus de dades compost té natura de seqüència si els seus elements es disposen "en fila índia". Els tipus de natura seqüencial utilitzen enters com a índexs per accedir als valors que contenen.

diccionaris (dictionary)

Els *diccionaris* són un tipus compost diferent. Són els *tipus associatius* predefinits de `Python`. Associen *claus*, que poden ésser de qualsevol tipus immutable, a valors, els quals poden ser de qualsevol tipus com passava amb els valors d'una llista o d'un tuple.

tipus associatius (mapping type)

claus (key)

Com a exemple, crearem un diccionari per a traduir paraules angleses al Català. En aquest diccionari, les claus són cadenes.

Una manera de crear un diccionari és començar amb el diccionari buit i afegir-hi parells *clau-valor*. El diccionari buit es denota per {}:

clau-valor (key-value)

```
>>> eng2cat = {}
>>> eng2cat['one'] = 'un'
>>> eng2cat['two'] = 'dos'
```

La primera assignació crea un diccionari anomenat `eng2cat`; les altres assignacions afegeixen parells clau-valor al diccionari. Podem veure el contingut actual del diccionari de la forma habitual:

```
>>> print eng2cat
{'two': 'dos', 'one': 'un'}
```

El parells clau-valor del diccionari es separen per comes. Cada parell conté una clau i un valor separada per dos punts.

L'ordre dels parells pot ser diferent del que ens esperem. `Python` utilitza algorismes complicats per a determinar on es desen els parells clau-valor en un diccionari i aquests algorismes no respecten l'ordre. Per a nosaltres, n'hi ha prou amb què pensem que aquest ordre és imprevisible.

Una altra forma de crear un diccionari és a partir d'una llista de parelles clau-valor, fent servir la mateixa sintaxi que la del resultat de l'exemple anterior:

```
>>> eng2cat = {'two': 'dos', 'one': 'un', 'three' : 'tres'}
```

L'ordre amb què escrivim els parells és indiferent. Als valors d'un diccionari s'hi accedeix mitjançant les claus, no pas amb índexs, per tant no ens ha de preocupar l'ordenació.

Vegem com fer servir una clau per accedir al valor associat:

```
>>> print eng2cat['two']
'dos'
```

La clau `'two'` ha donat com a resultat el valor `'dos'`.

11.1 Operacions dels diccionaris

La sentència **del** elimina un parell clau-valor d'un diccionari. Per exemple, el diccionari següent conté els noms de diverses fruites i el nombre de peces de fruita que tenim en estoc:

```
>>> inventari = {'pomes': 430, 'platans': 312, 'taronges': 525,
    'peres' : 217}
>>> print inventari
{'peres': 217, 'pomes': 430, 'taronges': 525, 'platans': 312}
```

Si algú compra totes les peres, podem esborrar l'entrada del diccionari:

```
>>> del inventari['peres']
>>> print inventari
{'pomes': 430, 'taronges': 525, 'platans': 312}
```

O si estem a l'espera que ens proveeixen amb més peres, podem simplement canviar el valor associat a les peres:

```
>>> inventari['peres'] = 0
>>> print inventari
{'peres': 0, 'pomes': 430, 'taronges': 525, 'platans': 312}
```

La funció len també és vàlida per als diccionaris i retorna el nombre de parells clau-valor:

```
>>> len(inventari)
4
```

11.2 Mètodes dels diccionaris

Els diccionaris tenen un bon nombre de mètodes predefinits.

El mètode keys donat un diccionari retorna una llista de claus:

```
>>> eng2cat.keys()
['three','two','one']
```

Com ja hem vist amb les cadenes i les llistes, els mètodes dels diccionaris fan servir la notació del punt, que especifica el nom del mètode a la dreta del punt i el nom de l'objecte al qual s'aplica el mètode a l'esquerra, just abans del punt. Els parèntesis indiquen que aquest mètode no té paràmetres.

invocació (invocation)

La crida a un mètode s'anomena *invocació*; en aquest cas, direm que hem invocat el mètode keys de l'objecte eng2cat. L'objecte del qual s'invoca un mètode és de fet el primer argument del mètode.

El mètode values és semblant; retorna la llista dels valors del diccionari:

```
>>> eng2cat.values()
['tres', 'dos', 'un']
```

El mètode items retorna claus i valors, com una llista de tuples —un per cada parell clau-valor:

```
>>> eng2cat.items()
[('three', 'tres'), ('two', 'dos'), ('one', 'un')]
```

El mètode has_key donada una clau retorna True si la clau és al diccionari i False en cas contrari:

```
>>> eng2cat.has_key('one')
True
>>> eng2cat.has_key('deux')
False
```

Aquest mètode pot pot ser mol útil, ja que si consultem una clau que no existeix al diccionari es produeix un error d'execució:

```
>>> eng2cat['dog']
Traceback (most recent call last):
  File "", line 1, in
KeyError: 'dog'
>>>
```

11.3 Àlies i còpies

Com que els diccionaris són mutables, hem d'anar amb compte amb els àlies. Quan dues variables facin referència al mateix objecte, els canvis de l'una afectaran l'altra.

Si volem modificar un diccionari mantenint una còpia de l'original, farem servir el mètode copy. Per exemple, opposites és un diccionari que conté parelles de contraris:

```
>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}
>>> alias = opposites
>>> copy = opposites.copy()
```

alias i opposites fan referència al mateix objecte; copy fa referència a una nova còpia del mateix diccionari. Si modifiquem alias, opposites també canvia:

```
>>> alias['right'] = 'left'
>>> opposites['right']
'left'
```

Si modifiquem copy, opposites no canvia:

```
>>> copy['right'] = 'privilege'
>>> opposites['right']
'left'
```

11.4 Matrius disperses

matriu dispersa (sparse matrix)

En un capítol anterior hem fet servir una llista de llistes per a representar una matriu. Aquesta era una bona opció per a matrius en les que la majoria d'elements són diferents de zero. Però considerem una *matriu dispersa* com aquesta:

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \end{pmatrix}$$

La seva representació com a llista conté molts zeros. En certa manera estem malbaratant molt espai de memòria:

```
matrix = [[0, 0, 0, 1, 0],
          [0, 0, 0, 0, 0],
          [0, 2, 0, 0, 0],
          [0, 0, 0, 0, 0],
          [0, 0, 0, 3, 0]]
```

Una alternativa és fer servir un diccionari. Com a claus, fem servir tuples que contenen el número de la fila i el de la columna. Heus ací la mateixa matriu representada amb un diccionari:

```
matrix = {(0, 3): 1, (2, 1): 2, (4, 3): 3}
```

Només calen tres parells clau-valor, un per a cada element de la matriu que no és nul. Les claus són tuples i els valors són nombres enters.

Per accedir a un element de la matriu, podem fer servir l'operador []:

```
matrix[(0, 3)]
1
```

Noteu que la sintaxi per al diccionari no és la mateixa que la de les llistes niuades. Enlloc de fer servir dos índexs enters, n'hem fet servir un, que és un tuple d'enters.

Però hi ha un problema. Si especifiquem un element nul, es produirà un error, ja que el diccionari no conté una entrada per a la clau associada:

```
>>> matrix[(1, 3)]
KeyError: (1, 3)
```

El mètode get resol aquest problema:

```
>>> matrix.get((0, 3), 0)
1
```

El primer argument és la clau; el segon argument és el valor que retornarà get si no es troba la clau al diccionari:

```
>>> matrix.get((1, 3), 0)
0
```

Definitivament, get millora la semàntica de l'accés a matrius esparses. Llàstima de la sintaxi.

11.5 Enters grans

Python disposa d'un tipus anomenat long per a processar enters de qualsevol mida (només limitada per la quantitat de memòria de què disposi l'ordinador).

Hi ha tres maneres de crear un valor long. La primera és fent un càlcul aritmètic d'una expressió que no càpiga en un enter habitual, un int. Per exemple, si avaluem l'expressió 10**100. Una altra manera és escriure un enter amb la lletra majúscula L al final del nombre:

```
>>> type(1L)
```

La tercera manera és cridar a long amb el valor que volem convertir com a argument. long, igualment com int i float, converteix int's, float's i fins i tot cadenes de dígits a enters grans:

```
>>> long(7)
7L
>>> long(3.9)
3L
>>> long('59')
59L
```

11.6 Comptatge de lletres

En el capítol ??, hem dissenyat una funció que comptava el nombre d'ocurrències d'una lletra en una cadena. Una versió més general d'aquest problema consisteix en fer un histograma de les lletres de la cadena, és a dir, quantes vegades apareix cada lletra en la cadena.

Un histograma com aquest pot ser útil per a comprimir un text. Com que cada lletra apareix amb una freqüència diferent, podem comprimir un fitxer utilitzant codis més curts per les lletres més comunes i codis més llargs per a les lletres menys freqüents.

Els diccionaris proporcionen una manera elegant de generar un histograma:

```
>>> letter_counts = {}
>>> for letter in "Mississippi":
... letter_counts[letter] = letter_counts.get (letter, 0) + 1
...
>>> letter_counts
{'M': 1, 's': 4, 'p': 2, 'i': 4}
```

Al començament tenim un diccionari buit. Per cada lletra de la cadena, consultem quantes vegades apareix fins al moment a la cadena (potser zero) i incrementem aquest comptador. Al final, el diccionari conté parells de lletres i llurs freqüències.

Pot semblar-nos més bonic mostrar l'histograma en ordre alfabètic. Ho podem fer amb els mètodes items i sort:

```
>>> letter_items = letter_counts.items()
>>> letter_items.sort()
>>> print letter_items
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]
```

Exercicis

Exercici 11.1 Escriu un programa que llegeixi una cadena de l'intèrpret de comandes i retorni una taula de les lletres que conté la cadena, ordenada alfabèticament, juntament amb el nombre de vegades que apareix cadascuna de les lletres. No s'ha de distingir entre majúscules i minúscules. Un exemple de com hauria de funcionar el programa podria ser aquest:

```
$ python letter_counts.py "ThiS is String with Upper and lower case Letters."
a 2
c 1
d 1
e 5
g 1
h 2
```

```
i 4
l 2
n 2
o 1
p 2
r 4
s 5
t 5
u 1
w 2
$
```

Exercici 11.2 Determina quina serà la sortida de l'intèrpret de `Python` en mode interactiu després d'executar cada sentència de la sèrie següent. Assumeix que s'executen en una sessió contínua.

a)
```
>>> d = {'apples': 15, 'bananas': 35, 'grapes': 12}
>>> d['banana']
```

b)
```
>>> d['oranges'] = 20
>>> len(d)
```

c)
```
>>> d.has_key('grapes')
```

d)
```
>>> d['pears']
```

e)
```
>>> d.get('pears', 0)
```

f)
```
>>> fruits = d.keys()
>>> fruits.sort()
>>> print fruits
```

g)
```
>>> del d['apples']
>>> d.has_key('apples')
```

Exercici 11.3 Assegura't que entens el perquè es produeix cadascun dels resultats. Després, aplica el que has après per a omplir el cos de la funció següent:

```
def add_fruit(inventory, fruit, quantity=0):
    """
    Adds quantity of fruit to inventory.

    >>> new_inventory = {}
    >>> add_fruit(new_inventory, 'strawberries', 10)
    >>> new_inventory.has_key('strawberries')
    True
    >>> new_inventory['strawberries']
    10
    >>> add_fruit(new_inventory, 'strawberries', 25)
    >>> new_inventory['strawberries']
"""
```

La teva solució ha de passar els doctests.

Exercici 11.4 Dissenya un programa anomenat alice_words.py que crei un fitxer de text de nom alice_words.txt amb una llista de les paraules de alice_in_wonderland.txt [1] juntament amb el nombre de vegades que hi apareix cada paraula. Les 10 primeres línies del fitxer produït haurien de ser quelcom així:

```
Word Count
========================
a 631
a−piece 1
abide 1
able 1
about 94
above 3
absence 1
absurd 2
```

Quantes vegades apareix la paraula alice al llibre?

Exercici 11.5 Quina és la paraula més llaga d'Alice in Wonderland? Quants caràcters té?

Exercici 11.6 Escriviu un programa anomenat `aparicionsLletres.py` que llegeixi una cadena donada per la línia de comandes i retorni una taula amb les lletres de l'abecedari per ordre alfabètic que apareguin en la cadena donada, amb el nombre d'aparicions per cada lletra. Un exemple d'execució del vostre programa podria ser, donada la següent entrada, el vostre programa hauria de retornar:

```
$python aparicionsLletres.py "hola adeu"
a 2
d 1
e 1
h 1
l 1
0 1
u 1
```

Exercici 11.7 Escriviu un script anomenat `aparicionsLletresFitxer.py` que llegeixi un fitxer donat per la línia de comandes i escrigui en un fitxer anomenat resultat.txt una taula amb les lletres de l'abecedari per ordre alfabètic que apareguin en el fitxer donat, amb el nombre d'aparicions per cada lletra.

Exercici 11.8 Escriviu una funció anomenada diccioCadenes que, donada una llista de cadenes, retorni un diccionari on la clau per cada valor sigui la primer paraula de cada cadena. Per exemple, donada la següent llista:

```
["una frase", "dos frases", "tres frases"]
```

La funció diccioCadenes hauria de retornar el següent diccionari:

```
{"una": "una frase", "dos": "dos frases", "tres": "tres frases"}
```

Exercici 11.9 Escriviu una funció anomenada diccioLlistes que, donada una llista de llistes, retorni un diccionari on la clau per cada valor sigui el primer element de cada subllista, i el valor emmagatzemat sigui la resta de la llista. Per exemple, donada la següent llista:

[1] Podeu obtenir el fitxer de `http://openbookproject.net//thinkCSpy/resources/ch10/alice_in_wonderland.txt`

```
[[1, 2, 3], ["un", "dos", "tres"]]
```

La funció diccioLlistes hauria de retornar el següent diccionari:

```
{1: [2, 3], "un": ["dos", "tres"]}
```

Exercici 11.10 Escriviu un programa que, donat un nom de fitxer per la línia de comandes, retorni un diccionari on la clau per cada element, sigui la primera paraula de la línia, i el valor sigui un nou diccionari amb clau les lletres de l'abecedari i valor el nombre d'aparicions de cada lletra en la línia.

Exercici 11.11 Escriviu una funció anomenada ordenaDiccio que, donat un diccionari, retorna el elements del diccionari ordenats respecte la seva clau.

Exercici 11.12 Escriviu un programa anomenat complicacioFinal.py que llegeixi per la línia de comandes el nom de 3 fitxers i retorni un diccionari amb clau el nom de cada fitxer i per valor una tupla amb el primer valor el nombre de lletres del fitxer, amb segon valor el nombre de vocals del fitxer, per tercer valor el nombre de paraules del fitxer, per quart valor la llista de paraules del fitxer, per cinquè valor la llista de línies del fitxer i per sisè valor una cadena amb el contingut del fitxer.

Exercici 11.13 Volem muntar un sistema d'autentificació d'usuaris. Per tal de fer això cal desenvolupar un menú on ens permeti crear, esborrar i editar usuaris amb tota la seva informació referent.

A partir d'un diccionari d'usuaris hem de guardar per cada un el nom, l'adreça i la paraula clau que farà servir per accedir al sistema. En el menú tindrem les següents opcions :

- Afegir un usuari : Es demana el nom d'usuari, el nom complert, el correu electrònic, el telèfon, l'adreça i la paraula clau.
- Esborrar un usuari : Es demana el nom d'usuari.
- Modificar un usuari : Es demana el nom d'usuari i es permet canviar les dades.

Un cop tenim la funcionalitat bàsica afegirem guardar a disc tota la informació de manera que poguem tenir unes dades. Per això afegirem al menú :

- Guardar a disc : Es demana el nom del fitxer que s'usarà i es guarda en ell la informació del diccionari.
- Llegir de disc : Es demana el nom del fitxer que es llegira i crearà el diccionari

Finalment crearem un altra programa en python que ens demani una nom d'usuari i una paraula clau i ens indiqui si està validat. Per tal de saber quin fitxer de dades s'usa es passarà per parametre.

Exercici 11.14 Tracteu de familiaritzar-vos amb els diccionaris. Primer de tot, mireu quin nom de tipus dona Python als diccionaris ("type name"). Executeu alguna comanda d'ajuda per tal de saber els mètodes disponibles en Python per al tractament d'objectes de tipus diccionari. Intenteu conèixer el funcionament de varis d'aquests mètodes, per exemple: values, clear, get, update, itervalues, ... Amb l'ajuda comproveu quins arguments requereixen i quin valor i tipus retornen.

Exercici 11.15 Mitjançant el disseny d'una funció, realitzeu i guardeu en una variable de tipus diccionari totes les taules de multiplicar dels 100 primers números enters. Crideu aquesta funció fent que el resultat l'imprimeixi en un fitxer de sortida, el nom del qual l'haurà entrat a l'usuari dins de la línea de comandes, com a paràmetre a l'hora de cridar al script.

EXERCICI 11.16 Dissenyeu un programa que simuli el control de inventari del nostre magatzem de ferreteria. Utilitzeu un diccionari per guardar els productes i les quantitats que disposem en inventari de cadascun d'aquests productes.

Inicialment tindrem els següents productes i quantitats per defecte: 25 cargols M5x20, 67 cargols TORX M4x20 i 109 femelles M6. Creeu un menú d'entrada on:

- L'usuari podrà crear un nou producte (comprovant que abans no existia) o eliminar un d'existent.
- L'usuari podrà consultar el inventari d'un producte.
- L'usuari podrà afegir o treure inventari d'un producte.
- L'usuari podrà extreure, en un fitxer que especificarà, un informe de tots els productes i quantitats existents.

EXERCICI 11.17 Dissenyeu una funció en `Python` que donat una llista de 5 números enters i una tupla que indiqui posicions en fila i columna d'aquests 5 números, crei una matriu dispersa, utilitzant diccionaris per a representar-la. La mida de la matriu serà sempre de 5 files i 5 columnes.

EXERCICI 11.18 Modifiqueu la funció anterior per tal de que la matriu pugui tenir tantes files i columnes com l'usuari cregui convenient. També feu una funció que escrigui la matriu per la terminal.

EXERCICI 11.19 Dissenyeu el una funció que realitzi l'intercanvi del parell valor-clau per a tots els elements d'un diccionari. Aquesta funció retornarà un altre diccionari on els valors inicials siguin les claus del diccionari a retornar i les claus del diccionari inicial seran els valors del diccionari a retornar.

EXERCICI 11.20 L'objectiu d'aquest exercici és implementar un mòdul destinat a facilitar el càlcul amb permutacions. De les diverses formes possibles per representar una permutació, usarem un diccionari en que les claus corresponen a la primera fila de la permutació i els valors a la segona. Per exemple, la permutació

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 2 & 0 \end{pmatrix}$$

la representarem com un diccionari que associa a al clau 0 el valor 1, a la clau 1 el valor 4 i així successivament. Si d és un diccionari que representa la permutació σ, llavors d[i]=j significa que $\sigma(i) = j$.

El mòdul, que anomenarem permutacio, ha d'implementar les següents funcions:

a) permutacio(l)

   l és una tupla o una llista indistintament que correspon a la segona fila d'una permutació. Retorna la permutació que s'escau.

b) identitat(n)

   Retorna la permutació identitat de mida n. Recordeu que la permutació identitat és la que compleix que $\forall i : 1 \leq i \leq n : \sigma(i) = i$.

c) producte(p1,p2)

   Donades dues permutacions p1 i p2 retorna la permutació que correspon al producte de p1 per p2.

d) iguals(p1,p2)

   Retorna True sii p1 i p2 són la mateixa permutació.

e) cicles(p)

Donada una permutació p, retorna la llista dels seus cicles. Un cicle es representa com una tupla d'enters. Per a exemplificar-ho tingueu en compte el següent cas de prova:

```
>>> pm = permutacio([1,4,3,2,0])
>>> cicles(pm)
[(0,1,4), (2,3)]
```

f) inversa(p)

Donada una permutació p retorna la seva inversa. Recordeu que la inversa σ^{-1} d'una permutació σ és la permutació que compleix que $\sigma \cdot \sigma^{-1} = 1$ essent 1 la permutació identitat.

g) aleatoria(n)

Retorna una permutació aleatoria de longitud n. Per implementar aquesta funció tingueu en compte la funció shuffle() del mòdul `random` de la llibreria estàndart de `Python`.

h) escriu(p,f)

Escriu la permutació p en el fitxer f. f ha de ser un fitxer obert en mode escriptura. Compte que f no és un nom de fitxer sinó un objecte fitxer!. Per implementar aquesta funció, primer cal decidir com es representa una permutació en un fitxer de text.

i) llegeix(f)

Llegeix una permutació del fitxer f i la retorna. f ha de ser un fitxer obert en mode lectura. Ha de ser capaç de llegir correctament una permutació escrita per la funció escriu que s'ha definit a l'apartat anterior.

EXERCICI 11.21 Aquest exercici té com objectiu crear un sistema senzill de codificació de missatges basant-se en permutacions. Com és sabut, les funcions ord() i chr() permeten obtenir el codi enter corresponent a un caracter o bé el caracter corresponent al codi enter. Per tant, podem transformar de forma senzilla una cadena de caracters en la corresponent llista d'enters i viceversa.

Una permutació σ pot ser entesa com una funció $\sigma : \mathbb{N} \to \mathbb{N}$. Siguin σ_1 i σ_2 dues permutacions arbitràries de longitud 256. Donat un missatge $m = m_0 m_1 \cdots m_l$ entès com una cadena de caracters, podem obtenir un missatge codificat aplicant al codi de cada lletra el producte de les dues permutacions anteriors i obtenint el missatge resultant. Més formalment diríem que el missatge codificat r és $r = \sum_{i=0}^{l} chr(\sigma_1 \cdot \sigma_2(ord(m_i)))$. De la mateixa forma, donat un missatge codificat r podem obtenir l'original aplicant la permutació $(\sigma_1 \cdot \sigma_2)^{-1}$.

Aquesta tècnica permet implementar missatges encriptats amb varies claus. Imagineu que la persona P_1 posseeix la clau σ_1 i la persona P_2 la clau σ_2. P_1 i P_2 es poden posar d'acord i, usant les seves claus, encriptar un missatge. Aquest missatge només pot ser desencriptat si cadascún d'ells cedeix de nou la seva clau.

En base a això, es desitja implementar un programa anomenat `encripta` que encripti el contingut d'un fitxer de text donades dues permutacions de longitud 265. L'encriptació s'ha de fer línia per línia. La comanda s'ha de poder usar d'acord amb el següent exemple:

```
$ python encripta clau1.key clau2.key dades.txt encriptat.txt
```

En l'exemple, `clau1.key` i `clau2.key` són els fitxers que contenen les permutacions corresponents a la clau de cada usuari. `dades.txt` és el fitxer original i `encriptat.txt` és el fitxer on es desarà el contingut encriptat.

De forma similar cal implementar un programa anomenat `desencripta` que faci just el contrari.

GNU Free Documentation License

Version 1.3, 3 November 2008
Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0 PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such,"Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3 COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C State on the Title page the name of the publisher of the Modified Version, as the publisher.

D Preserve all the copyright notices of the Document.

E Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H Include an unaltered copy of this License.

I Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this

License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11 RELICENSING

“Massive Multiauthor Collaboration Site” (or “MMC Site”) means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A “Massive Multiauthor Collaboration” (or “MMC”) contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.