

Python-en Tutoretza

Axlor Elorza Eizagirre

<axlor@zundan.com>

2005.eko apirilaren 8

Gaien Aurkibidea

1	Sarrera	5
2	Datu motak	7
2.1	Zenbakiak	7
2.2	Kateak	7
2.3	Zerrendak (array-ak)	10
2.3.1	Funtzioak	12
2.4	Tupla / sekuentziak	16
2.5	Hiztegiak	17
3	Fluxu kontrola	19
3.1	if sententzia	19
3.2	while sententzia	19
3.3	for sententzia	20
3.4	break, continue eta else eraketak bukletan	21
3.5	Baldintzei buruz	21
3.6	Sekuentzia eta beste mota batzuen arteko konparazioa	22
4	Funtzioak	25
4.1	lambda egiturak	30
4.2	Dokumentazio kateak	30
5	Moduluak	31
5.1	Moduluei buruz gehiago	32
5.2	dir() funtzioa	33
5.3	Paketeak	34
5.3.1	Pakete batetik * inportatu	36

6	Sarrera irteerak	37
6.1	Irteera formatu hobetua	37
6.2	Fitxategien idazketa eta irakurketa	40
6.2.1	Fitxategi objetuen metodoak	40
6.2.2	<code>pickle</code> modulua	42
7	Erroreak eta eszepzioak	45
7.1	Sintaxi erroreak	45
7.2	Eszepzioak	45
7.3	Eszepzioen kudeaketa	46
7.4	Eszepzioak jaurtiarazi	48
7.5	Erabiltzaileak zehazturiko eszepzioak	48
7.6	Garbiketa ekintzak	49
8	Klaseak	51
8.1	Izen eremu eta esparruak	51
8.2	Klaseen definizioen sintaxia	52
8.3	Klase objektuak	53
8.4	Instantzia objektuak	54
8.5	Metodo objektuak	54
8.6	Herentzia	56
8.6.1	Herentzia anizkoitza	56
8.7	Aldagai pribatuak	57
8.8	Amaitzeko	58
8.8.1	Eszepzioak ere klaseak izan daitezke	58

1 Sarrera

Python programazio hizkuntza interpretatua 90. hamarkada hasieran sortu zen GUIDO VAN ROSSUM-en eskutik. Python-ek maila altuko datu estrukturak eta objetuei zuzenduriko programazioarekiko soluzio simple eta eraginkorrak ditu, honela script-ak eta aplikazioen garapen azkarrerako hizkuntza egokia delarik.

Python-ek aginduen lerroak (shell) baino laguntza eta azpiegitura gehiago eskaintzen ditu. Bestalde C-k baino errore egiaztapen gehiago du eta, maila altuko hizkuntza izatean, matri-ze malgu edota hiztegien moduko maila altuko datu motak barneratzen ditu. Erraza delarik funtzio berri eta C edo C++-n landuriko datu motak erabiliz Python interpretea hedatzea eragiketa kritikoak abiadura handiengan burutzeko edo Python programak modu bitarrean dauden liburutegiekin lotzeko. Honela Python interpretea C-n idatziriko aplikazio bati lotu daiteke eta aplikazio honentzat makroen hizkuntza moduan erabili.

Python-en programak C edo C++-en baino motzagoak dira, honako arrazoiak direla medio:

- Maila altuko datuen erabilerak eragiketa konplexuak lerro bakar batean idazten laguntzen du.
- Sententzien taldekatzea koskatzea erabiliz egiten da, `begin/end` edo `giltzen {/}` ordez.
- Argumentuak eta aldagaiak deklaratu beharrik ez dago.

Modu elkarreragilean ere erabili daiteke, aginduak irakurri eta exekutatu. Hizkuntzaren ezau-garriekin frogak egin daitezke larrik, programa baten azpi-funtzioak adibidez. Gaitasun handiko kalkulagailu bezala ere erabili daiteke.

2 Datu motak

Datu mota ezberdinekin frogak egiteko interpretea erabiltzea gomendatzen da. Interpretea abiaraztean gonbit nagusia “>>>” ageri arte itxaron (ez luke asko tardatu behar).

Sarrerak gonbit edo indikatzaileak (“>>>” eta “. . .”) ageri direnean emango dira, gonbitik ageri ez den lerroetan interpretearen irteera emango delarik. Adibide askotan iruzkinak azalduko dira, beti ere “#” karaktereaz hasiko dira, lerroaren amaieraraino luzatuko direlarik. Iruzkinak lerro haseran edo kodearen atzetik joango dira eta ez konstante literal baten barruan. Kate baten barruko traol (#) hutsa da, besterik gabe.

2.1 Zenbakiak

Interpreteak kalkulagailu xume bat bezala jokatu du: adierazpena idatzi eta berak emaitza erakutsiko du: +, -, * eta / operadoreak beste hizkuntzetan bezala jokatzen dute. Parentesiak erabili daitezke operazioak taldekatzeko.

Zenbakiak adierazteko modu ezberdinak:

```
123456789          # osoa
0x003f             # sistema hamartarraz gain, besteen erabilera
0776               # aurretik 0 bat izanda ere berdina da
10L                # long
2134865345657L    # long
0.5                # erreala
5e10               # exponentziala
1.5+0.3j           # zenbaki konplexuak
```

2.2 Kateak

Kateak adierazteko modu ezberdinak:

```
>>> 'elurra ari du'
'elurra ari du'
>>> 'L\'Hospitalet'
```

2 Datu motak

```
"L'Hospitalet"  
>>> "L'Hospitalet"  
"L'Hospitalet"  
>>> '"Bai," esan zuen.'  
'"Bai," esan zuen.'  
>>> "\"Bai,\" esan zuen."  
'"Bai," esan zuen.'  
>>> '"L\'Hospitalet," esan zuen.'  
'"L\'Hospitalet," esan zuen.'
```

Kateek lerro ezberdinak bete ditzakete modu ezberdinetara. Lerro fisikoaren amaiera lerro logikoaren amaieraz izatea lortu daiteke alderantzizko barra erabiliz:

```
kaixo = "Hau lerro bat baino gehiago dituen\n\  
testua da, C izango balitz bezala.\n\  
    Ohar zaitez lerro haserako zuriunea\  
    esanguratsua dela.\n"  
print kaixo
```

Hau erakutsiko du:

```
Hau lerro bat baino gehiago dituen  
testua da, C izango balitz bezala.  
    Ohar zaitez lerro haserako zuriunea esanguratsua dela.
```

Gako hirukoitzak `"""` edo `'''` erabiliz ez dago alderantzizko barrarik erabili beharrik lerro jauziak adierazteko.

```
print """ Erabilera: gauzatxoa [AUKERAK]  
    -h                               Erabilera mezua erakutsi  
    -H ZerbitzariIzena               Konektatu beharreko zerbitzariaren izena  
    """
```

Hau erakutsiko du:

```
Erabilera: gauzatxoa [AUKERAK]  
    -h                               Erabilera mezua erakutsi  
    -H ZerbitzariIzena               Konektatu beharreko zerbitzariaren izena
```

Kateak lotzeko + operadorea erabiliko dugu eta errepikatzeko *:

```
>>> hitza = 'Laguntza' + 'Z'
>>> hitza 'LaguntzaZ'
>>> '<' + hitza*4 + '>'
'<LaguntzaZLaguntzaZLaguntzaZLaguntzaZ>'
```

Kateak lotzeko beste modu batzuk:

```
>>> 'kat' 'ea' # <- Honek balio du 'katea'
>>> import string
>>> string.strip('kat') + 'ea' # <- Honek balio du 'katea'
>>> string.strip('kat') 'ea' # <- Honek ez du balio
File "<stdin>", line 1
    string.strip('kat') 'ea'
                        ^
SyntaxError: invalid syntax
```

Kateak indexatu daitezke eta lehenengo karakterearen indizea 0 izango da. Ez dago karaktere mota ezberdinik; karakterea bat luzerako katea da. Azpikateak mozketan oharpen bitartez zehazten dira: bi puntuz bananduriko bi indize.

```
>>> hitza[4]
'n'
>>> hitza[:2] # Lehen bi karaktereak
'La'
>>> hitza[2:] # Lehen biak ezik beste guztiak
'guntzaZ'
```

Ondorioz:

```
>>> hitza[:3]+hitza[3:] # Lehen hiruak + beste guztiak
'LaguntzaZ'
>>> hitza[2:4] # Lehen 4 karakteretatik lehen biak ez besteak
'gu'
```

Python-en kateak ezin dira aldatu. Posizio indexatu bat asignatzen saiatzen bagara errorea emango digu.

```
>>> hitza[0]='X'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

2 Datu motak

Eduki bateratuarekin kate berri bat sortzea erraza eta eraginkorra da:

```
>>> 'X' + hitza[1:]
'XaguntzaZ'
```

Indizeak negatiboak ere izan daitezke, zenbatzea atzetik hasi dadin:

```
>>> hitza[-3]      # Azken hirugarren karakterea
'z'
>>> hitza[-3:]    # Azken hiru karaktereak
'zaZ'
>>> hitza[: -3]   # Azken hiru karaktereak ez beste guztiak
'Lagunt'
>>> hitza[-100:]
'LaguntzaZ'
>>> hitza[-10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

`len()` barne-funtzioak katearen luzera itzultzen du:

```
>>> len(hitza)
9
```

2.3 Zerrendak (array-ak)

Python-ek datu mota *konposatu* ezberdinak erabiltzen ditu, beste balio batzuk taldekatzeko. Joko gehiena eskaintzen dutenak zerrendak dira, kortxete artean komaz bananduz idatziko dira balio edo elementuak. Zerrendako elementu guztiek ez dute zertan mota berdinekoak izan behar.

```
>>> a = ['arkatza', 100.9, 1234]
>>> a
['arkatza', 100.90000000000001, 1234]
```

Zerrenden indizeak kateenak bezalaxe 0-rekin hasten dira. Zerrendak ere moztu, lotu, etb. egin daitezke.

2.3 Zerrendak (array-ak)

```
>>> a[0]
'arkatza'
>>> a[3]
1234
>>> a[-2]
100.90000000000001
>>> a[1:-1]
100.90000000000001
>>> a[:2] + ['elurra', 2*2]
['arkatza', 100.90000000000001, 'elurra', 4]
>>> 3*a[:3] + ['Eurre!']
['arkatza', 100.90000000000001, 1234, 'arkatza', 100.90000000000001,
 1234, 'arkatza', 100.90000000000001, 1234, 'Eurre!']
```

Kateek ez bezala zerrendek aldaketak jasan ditzakete:

```
>>> a
['arkatza', 100.9, 1234]
>>> a[3] = a[3] + 23
>>> a
['arkatza', 100.90000000000001, 1257]
```

Mozketa bat ere asigmatua izan daiteke, zerrendaren tamaina ere aldatua izan daitekeelarik:

```
>>> # Elementuak aldatu:
... a[0:2] = [1, 12]
>>> a
[1, 12, 1234]
>>> # Elementuak kendu:
... a[0:2] = []
>>> a
[1234]
>>> # Elementuak gehitu:
... a[1:1] = ['ZimZum', 'Duran']
>>> a
[1234, 'ZimZum', 'Duran']
>>> a[:0] = a      # Kopia bat sartu bere haseran
>>> a
[1234, 'ZimZum', 'Duran', 1234, 'ZimZum', 'Duran']
```

2 Datu motak

`len()` funtzioaren erabilera kateetan bezalakoxea da:

```
>>> len(a)
6
```

Zerrendak habitzea (elementu bezala zerrendak dituzten zerrendak) ere posible da, adibidez:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('extra')      # Consulte la sección 5.1
>>> p
[1, [2, 3, 'extra'], 4]
>>> q
[2, 3, 'extra']
```

Ohar zaitetz azken adibidean `p[1]` eta `q` objektu berataz ari direla.

2.3.1 Funtzioak

range() Barne-funtzio hau zenbaki sekuentzia baten barneko elementuak banan bana emaitz bezala kaleratzeko oso baliagarria da. Progresio arimetikoekin zerrendak sortzen ditu, adibidez:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Adierazitako azken puntua ez da sorturiko zerrendaren parte izango, hau da, `range(10)`-ek 10-eko luzerako zerrenda baten indizeen balioak sortuko ditu. Nahi izanez gero zerrenda beste zenbaki batean hasi dezakegu edo gehikuntza ezberdin bat zehaztu (step):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Sekuentzia baten indizeak hartzen joateko, `range()` eta `len()` modu honetara konbinatu daitezke:

```
>>> a = ['Cure', 'for', 'pain']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Cure
1 for
2 pain
```

Lista motako datuek dituzten metodoak

append(x) Zerrenda baten amaieran elementu bat sartu; `a[len(a):] = [x]` -en baliokidea litzateke.

extend(L) Zerrenda luzatzeko beste zerrenda bat erantsiz; `a[len(a):] = L` -en baliokidea.

insert(i, x) Elementu bat emaniko posizio batean gehitzen du, eta ez ordeztu.

remove(x) `x`-en balioa duen zerrendako lehen elementua ezabatuko du. Elementu hori existitzen ez bada errorea emango du.

index(x) Zerrendan `x` balioa duen lehen elementuaren indizea itzuliko digu.

count(x) Zerrendan `x` balioa zenbat aldiz dagoen itzuliko digu.

sort() Zerrenda txikitik handira ordenatuko du. Zerrenda aldatua geldituko da.

reverse() Zerrenda alderantzikatuko du.

Zerrendak pila moduan (LIFO)

```
>>> pila = [3, 4, 5]
>>> pila.append(6)
>>> pila.append(7)
>>> pila
[3, 4, 5, 6, 7]
>>> pila.pop() 7
>>> pila [3, 4, 5, 6]
>>> pila.pop()
6
```

2 Datu motak

```
>>> pila.pop()
5
>>> pila
[3, 4]
```

Zerrendak kola moduan (FIFO)

```
>>> cola = ["Patton", "Fantomas", "Lonbardo"]
>>> cola.append("Buzz")
>>> cola.append("Dunn")
>>> cola.pop(0)
'Patton'
>>> cola.pop(0)
'Fantomas'
>>> cola
['Lonbardo', 'Buzz', 'Dunn']
```

Programazio funtzionaleko herremintak Zerrendekin lan egiteko hiru funtzio baliagarri ditugu: `filter()`, `map()` eta `reduce()`.

filter(funtzioa, zerrenda) Funtzioa(elementua) egia deneko elementuen zerrenda itzuliko digu. Zenbaki lehen batzuk kalkulatzeko adibidea:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map(funtzioa, zerrenda) Zerrendako elementu bakoitzari funtzioa aplikatuz lorturiko zerrenda itzuliko digu.

```
>>> def kubo(x): return x*x*x
...
>>> map(kubo, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

reduce(funtzioa, zerrenda) Funtzioari zerrendaren lehen bi elementuak pasako dizkio hasieran, gero lehen bi hauen artean lortuiko emaitza eta hurrengo elementua, eta horrela hurrenez hurren azken emaitza lortu arte. Adibidez 1-etik 10-erako zenbakien arteko batura kalkulatzeko:

```
>>> def batura(x,y): return x+y
...
>>> reduce(batura, range(1, 11))
55
```

Zerrendan elementu bakarra badago, bere balioa itzuliko digu; zerrenda hutsik badago, exzepzio bat jaurtiko du.

Hirugarren argumentu bat pasa geniezaiokegu haserako balioa zehazteko. Kasu honetan balio hau itzuliko luke zerrenda hutsik balego eta funtzioa lehen elementuari aplikatuko zaio, gero bigarrenari eta horrela hurrenez hurren. Adibidez:

```
>>> def batu(sekuentzia):
...     def batu(x,y): return x+y
...     return reduce(batu, sekuentzia, 0)
...
>>> batu(range(1, 11))
55
>>> batu([])
0
```

LC-ak LC-ek `map()`, `filter()` edota lambda funtzioak erabili gabe zerrendak sortzeko modu zehatza dira. LC-en egitura adierazpen bat, `for` sententzia bat eta `for` edo `if` (zero edo gehiago) adierazpenez eratuak daude. Emaitza izango den zerrenda adierazpenaren ondoren datozen `for` eta `if` sententzietan adierazpena ebaluatu ondoren lortuko da. Adierazpenak emaitz bezala tupla bat itzuli behar badu parentesi artean itxi beharko da.

```
>>> fruitufreskoa = [' platano', ' sagarra ', 'marrubia ']
>>> [arma.strip() for arma in fruitufreskoa]
[' platano', ' sagarra ', 'marrubia ']
>>> bec = [2, 4, 6]
>>> [3*x for x in bec]
[6, 12, 18]
>>> [3*x for x in bec if x > 3]
[12, 18]
>>> [3*x for x in bec if x < 2]
[]
>>> [{x: x**2} for x in bec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in bec]
```

2 Datu motak

```
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in bec] # error - tupletan parentesia behar da
File "<stdin>", line 1
    [x, x**2 for x in bec]
      ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in bec]
[(2, 4), (4, 16), (6, 36)]
>>> bec1 = [2, 4, 6]
>>> bec2 = [4, 3, -9]
>>> [x*y for x in bec1 for y in bec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in bec1 for y in bec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

del sententzia Indizea edo mozketara pasata elementuak zerrendatik ezabatzeko balio du, baita aldagai osoa ezabatzeko ere. Adibidez:

```
>>> a [-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
>>> del a
```

2.4 Tupla / sekuentziak

Zerrenda eta kateak bezala tuplak ere datu mota sekuentzialak dira. Tuplek zenbait balio komaz bananduak izaten dituzte:

```
>>> t = 12345, 54321, 'Rosemary'
>>> t[0]
12345
>>> t
(12345, 54321, 'Rosemary')
>>> # Tuplak habitzea posible da:
```

```
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'Rosemary'), (1, 2, 3, 4, 5))
```

Tuplak oso erabilgarriak dira, adibidez koordenada bikoteetan (x,y), datu base baten langileen erregistroetan, etab. Tuplak kateak bezala, aldaezinak dira: elementu indibidualei ezin zaie baliorik asignatu (nahiz eta mozketara eta kateatzea erabili daitezken). Zerrendak bezalako objektu aldakorrez osaturiko tuplak posible dira.

Elementu bakarreko edo elementurik gabeko tuplak adierazteko parentesi hutsak eta elementu bakarraren ondoren koma bat gehituz lortuko da.

```
>>> hutsa = ()
>>> vendetta = 'charade', # <-- Begiratu amaierako koma
>>> len(hutsa)
0
>>> len(vendetta)
1
>>> vendetta
('charade',)
```

2.5 Hiztegiak

Hiztegiak Python-eko barne-datu mota erabilgarriak dira. Sekuentziekin alderatuz, zenbakiez indexatu beharrean *gakoak* erabiliz indexatuko dira, edozein mota aldaezinekoak izango direlarik, zenbaki eta kateak bezala. Tupla gako bezala erabiltzeko baldintza tupla, kate edo zenbakiez bakarrik eratzea da. Tupla batek zuzenean edo zeharka objektu aldakor bat badu ezingo da gako bezala erabili. Listak ezingo dira gako bezala erabili `append()` edo `extend()` metodoak erabiliz aldatzeko gaitasuna baitute, mozketara asignazio eta asignazio aniztuez gain.

Egokiena hiztegiak ordenarik gabeko *gakoa* : *balioa* bikoteen talde bezala hartzea da, gakoak hiztegi barruan bakarrik izango diren baldintzarekin. `{}` giltz pareak hiztegi huts bat adierazten du eta *gakoa* : *balioa* pare bat sartuz gero *gakoa* : *balioa* pare haseratuak sartzen dira hiztegian.

Hiztegien eragiketa nagusiak bertan gako bat erabiliz balio bat gorde eta gako bat emanik balioak ateratzea dira. Hauxe gain `del` erabiliz *gakoa* : *balioa* bikote bat ezabatu daiteke. Iada hiztegian dagoen gako bat sartuz gero lehendik dagoen balioa ezabatu egingo du. Existitzen ez den gako bat ateratzen saiatzeak errorea emango du.

Objetu baten `keys()` metodoak hiztegiaren gako guztiak itzuliko dizkigu ausazko ordena batean (ordenaturik nahi badira `sort()` metodoa aplikatu beharko zaio gako zerrendari).

Gako bat hiztegian ba al dagoen jakiteko hiztegiaren `has_key()` metodoa erabili behar da.

2 Datu motak

Hona hemen hiztegi bat erabiltzen duen adibide bat:

```
>>> tel = {'mikel': 4098, 'xabier': 4139}
>>> tel['alex'] = 4127
>>> tel
{'alex': 4127, 'mikel': 4098, 'xabier': 4139}
>>> tel['mikel']
4098
>>> del tel['xabier']
>>> tel['axlor'] = 4127
>>> tel
{'alex': 4127, 'mikel': 4098, 'axlor': 4127}
>>> tel.keys()
['alex', 'mikel', 'axlor']
>>> tel.has_key('axlor')
true
```

3 Fluxu kontrola

3.1 if sententzia

Nahi adina `elif` eta `else` izan ditzake, aukerakoa delarik hauen erabilera. `if ... elif ... elif...` sekuentziak beste hizkuntza batzuetako `switch` edo `case` bezala jokatzen du.

Formatua:

```
>>> if baldintza1: sententzia1
    elif baldintza2: sententzia2
    ...
    else: sententziaN
```

Adibidea:

```
>>> x = int(raw_input("Sartu zenbaki bat: "))
>>> if x < 0:
...     x = 0
...     print 'Zenbaki negatiboa zerora bihurtu da.'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Bat'
... else:
...     print 'Gehiago'
... 
```

3.2 while sententzia

Formatua:

```
while baldintza :
    sententzia
```

3 Fluxu kontrola

Adibidea:

```
>>> # Fibonacci seriea
... # Aurreko bi zenbakiren baturak emango digu hurrengoaren balioa
... a, b = 0, 1
>>> while b < 10:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8
```

Python-en, C-n bezala, zeroren ezberdina den edozein balio egiazkoa da eta 0 berriz gezurrezkoa. Baldintza, zerrenda edo beste edozein sententzia izan daiteke, edozer gauz zero ez den luzerarekin egiazkoa da eta sekuentzia hutsak berriz gezurrezkoak.

3.3 for sententzia

for sententziak C edo Pascal-ekiko ezberdintasun bat du. Progresio arimetiko bateko elementu denak banan bana pasa beharrean edo programatzaileari initalizazio, egiaztapen eta urrats-jauzia aukeratzeko askatasuna utzi beharrean, Python-en sekuentzia baten (zerrenda edo kate bat, adibidez) elementu guztiak banan bana pasako ditu, sekuentzian azaltzen diren ordenean.

Formatua:

```
>>> for elementua in zerrenda:
...     sententzia
```

Adibidea:

```
>>> # kateakñehurtzeko:
... a = ['katu', 'lehioa', 'hitzluzeagoa']
>>> for x in a:
...     print x, len(x)
...
katu 4 lehioa 6 hitzluzeagoa 12
```

Kasuren batean for sententziaren barruan banan bana pasatzen ari garen sekuentzian (zerrendak adibidez) aldaketak egin behar baditugu kopia bat pasatu behar da. Hau errazteko mozketak oharpena erabili dezakegu.

```
zerrenda[:]
```

3.4 *break, continue eta else eraketak bukletan*

```
>>> for x in a[:]: # lista osoaren mozketan kopia egin behar da
...     if len(x) > 7: a.insert(0, x)
...
>>> a
['hitzluzeagoa', 'katu', 'lehioa', 'hitzluzeagoa']
```

3.4 **break, continue eta else eraketak bukletan**

`break` sententziak, C-n bezala, aribidean dagoen barrueneko `for` edo `while` bukletik irteera bultzatuko du.

`continue` sententziak buklearen hurrengo errepikapenean jarraitzera behartzen du.

Bukleko eraketek `else` klausula izan dezakete. Hau, buklean izanez gero, bukletik irteteen baldintza gezurrezkoa delako (`while-en`) edo zerrenda amaierara iritsi denean (`for-en`) exekutatuko da, baina ez `break` sententziarekin amaitu bada.

3.5 Baldintzei buruz

Lehenago deskribaturiko `for` eta `while` eraketetan erabilitako baldintzek konparazioez gain beste baldintza mota batzuk erabili ditzakete.

`in` eta `not in` konparazio operadoreek balio bat sekuentzi baten barrun ba al dagoen begiratzten dute. `is` eta `is not` bi objektu benetan bera al diren begiratzten dute. Hauek listak bezalako objektu aldakorretan bakarrik dute garrantzia. Konparazio operadore guztiek lehentasun bera dute, zenbakizko operadoreena baino baxuagoa.

Konparazioak elkartu daitezke: Adibidez, $a < b == d$, $a \text{ b}$ baino txikiagoa al den eta b eta d berdinak al diren egiaztatzen du.

`and` eta `or` operadore logikoak erabiliz ere egin daitezke konparazioak, `not`-ek emaitza ezeztatuko lukeelarik. Konparazio operadoreek baino lehentasun baxuagoa dute. Beraien artean lehentasuna `not`-ek izango du, baxuenak `or`-ek izango lukeelarik, ondorioz $A \text{ and } \text{not } B \text{ or } C$, $(A \text{ and } (\text{not } B)) \text{ or } C$ -ren berdina da. Emaitza ezkerretik eskubira ebaluatuko da eta emaitza zehaztu orduko ebaluazioa moztu egingo da. Adibidez, A egia da eta B gezurra, ondorioz $A \text{ and } B \text{ and } D$ -k ez du ebaluatuko D adierazpena. Orokorrean, lasterbide deituriko operadore hauek itzuliko duten balioa, balio orokor bezala erabiltzean eta ez balio logiko bezala, ebaluatuturiko azken argumentua izango da:

```
>>> str1, str2, str3 = "", 'Norbegia', 'Mailu dantza'
>>>ñon_null = str1 or str2 or str3
>>>ñon_null
'Norbegia'
```

Python-en, C-n ez bezala, ezin da adierazpen baten barruan asignaziorik egin.

3.6 Sekuentzia eta beste mota batzuen arteko konparazioa

Sekuentzia objektuak mota berdineko sekuentziez alderatu daitezke, ordenazio *lexikografikoa* erabiliko delarik.: Lehenendabizi lehen bi elementuak konparatuko ditu, ezberdinak badira emaitza hauek emango digute, eta berdinak badira hurrenez hurren sekuentzia bakoitzeko elementuak alderatzen joango da, sekuentzietako bat amaitu arte. Elementuetako bat sekuentzia bat bada, konparazio lexikografiko habitua bat emango da. Konparazio lexikografikoa egiteko ASCII kodearen ordena erabiliko da. Hona hemen mota berdineko sekuentzien arteko alderapen batzuk:

```
(1, 2, 3) < (1, 2, 4)
[1, 2, 3] < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4) < (1, 2, 4)
(1, 2) < (1, 2, -1)
(1, 2, 3) == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Mota ezberdineko konparazioak egitean motak izenez ordenatzen dira. Honela, lista edo zerrenda bat kate bat baino txikiagoa izango da, katea tupla baino txikiagoa, etab. Mota ezberdinetako zenbakizko balioen artean zenbakizko balio hoiek konparatuko dira, 0 0.0-ren berdina izango delarik.

Adibidea:

Honako adibide honek telefono zenbakiak gordeko ditu hiztegi batetan:

```
#
# Zenbakiak.py fitxategiaren hasera
#
def menua():
    print '1. Telefono zenbakiak erakutsi'
    print '2. Zenbaki berri bat sartu'
    print '3. Zenbaki bat ezabatu'
    print '4. Zenbaki bat bilatu'
    print '5. Irten'
    print
```

3.6 Sekuentzia eta beste mota batzuen arteko konparazioa

```
zenbakiak = {}
aukera = 0
menua()
while aukera != 5:
    aukera = input("Sartu aukera (1-5):")
    if aukera == 1:
        print "Zenbaki telefonikoak: "
        for x in zenbakiak.keys():
            print "Izena: ",x," \tZenbakia: ",zenbakiak[x]
        print
    elif aukera == 2:
        print "Sartu izena eta zenbakia"
        izena = raw_input("Izena: ")
        telefonoa = raw_input("Zenbakia: ")
        zenbakiak[izena] = telefonoa
    elif aukera == 3:
        print "Ezabatu izena eta zenbakia"
        izena = raw_input("Izena: ")
        if zenbakiak.has_key(izena):
            del zenbakiak[izena]
        else:
            print "Ezin izan da ",ñombre, " aurkitu"
    elif aukera == 4:
        print "Zenbakia bilatu"
        izena = raw_input("Nombre:")
        if zenbakiak.has_key(izena):
            print "Zenbakia: ",zenbakiak[izena]
        else:
            print "No pude encontrar a ",izena
    elif aukera != 5:
        menua()
# Zenbakiak fitxategiaren amaiera
```

3 Fluxu kontrola

4 Funtzioak

Adibidea:

```
>>> def fib(n):
...     "n-rainoko Fibonacci seriea idatzi"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Orain funtzio definitu berriari deituko diogu:
... fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

def hitz-gakoak funtzio baten definiziora garamatza, hau funtzioaren izenaz eta parentesi artean parametro formalen zerrendak jarraituko dute. Funtzioaren gorputza osatzen duten sententziak hurrengo hilaran hasten dira eta koskatuak egon behar dute. Funtzioaren gorputzeko lehen sententzia kate-konstante bat izan liteke, non funtzioaren zeregina argituko ligukeen. Hau dokumentazio idatzi/elektronikoa sortzeko edo erabiltzailea kodean zehar interktiboki nabigatzeko baliagarria da.

Funtzio baten exekuzioak Python-en aldagai lokalentzat sinbolo-taula berri bat sortzen du. Zehazki, funtzio baten aldagaien asignazio guztiek balioa sinbolo-taula lokalean gordetzen dute; honela, aldagaien erreferentziek lehenengo sinbolo-taula lokalean begiratuko dute, gero sinbolo-taula orokorrean eta, azkenik, barneko izen-taulan. Arrazoi honexegatik ezin izango diogu aldagai orokor bati baliorik ezarri funtzio baten barruan (global sententzi batean aipatzen ez bada behintzat), nahiz eta beraiei erreferentzia egin daitekeen.

Funtzio bati argumentuak balioz pasatuko zaizkio, balioa beti objeturi erreferentzia bat izango delarik eta ez objektuaren balioa.

Funtzioaren definizioak funtzioaren izena indarrean dagoen sinbolo-taulan gordeko du, izenaren balioa interpreteak onarturiko mota izango du erabiltzaileak definituriko funtzio bezala. Honela berizendapen-generiko bezala balio duelarik:

```
>>> fib
```

4 Funtzioak

```
<function fib at 0x403f6bc4>
>>> f = fib
>>> f (50)
1 1 2 3 5 8 13 21 34
```

Python-en prozedurak baliorik itzultzen ez duten funtzioak dira. Teknikoki ordea, None balioa itzuliko lukete.

```
>>> fib(0)
None
```

Zenbakiak idatzi beharrenean, hauek itzuliko dizkigun funtzioa:

```
>>> def fib2(n):
...     "n-rainoko Fibonacci seriea itzuli"
...     a, b = 0, 1
...     emaitza = []
...     while b < n:
...         emaitza.append(b)    # Edo: emaitza = emaitza + [b]
...         a, b = b, a+b
...     return emaitza
...
>>> f10 = fib2(10)    # funtzioari deitu
>>> f10              # emaitza idatzi
[1, 1, 2, 3, 5, 8]
```

Funtzioak argumentu kopuru aldakorrez definitzea ere zilegi litzateke, konbinatu daitezken hiru modu daude:

Balio lehenetsiak argumentuetan

Modurik errazena da argumentu bat edo gehiagori balio lehenetsiak zehazteko. Honela dituen baina argumentu gutxiagorekin deitu dezakegun funtzioa sortu dezakegu:

```
def baieztatu(adierazlea, ahaleginak=3, kexua='Bai edo ez!'):
    while 1:
        erantzuna = raw_input(adierazlea)
        if erantzuna in ('b', 'bai'): return 1
        if erantzuna in ('e', 'ez'): return 0
```

```
ahaleginak = ahaleginak - 1
if ahaleginak < 0: raise IOError, 'Erabiltzailea ukatua'
print kexua
```

Funtzioari honela deitu diezaiokegularik: baieztatu ('Irteterik nahi?') edo: baieztatu ('Ezabatu fitxategia?', 2).

Balio lehenetsiak funtzioa defintzean ebaluatuko dira:

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

Honek 5 erakutsiko liguke.

Kontuz! Argumentu lehenetsiak behin bakarrik ebaluatzen dira. Emaitza ezberdinak lortuko ditugu balio lehenetsiak zerrenda edo hiztegien moduko balio aldakorrak direnean. Adibidez:

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

Honek erakutsiko ligukena:

```
[1]
[1, 2]
[1, 2, 3]
```

Ez badugu nahi balio lehenetsia hurrendo deiekin konpartitzerik honela egin genezake:

```
def f(a, l=None):
    if l is None:
        l = []
    l.append(a)
    return l
```

Argumentuak gako

"gako = balio" modua erabiliz ere deitu diezaiokegu funtzioari. Adibidez:

```
def txoria(tentsioa, egoera='frijitua', ekintza='lehertu', mota='Grisa'):  
    print "-- Txori honek ezingo luke ", ekintza,  
    print tentsioa, " boltio jasanaraziz ere."  
    print "-- Lumatza ederra ", mota,"rena."  
    print "-- ", egoera, " dago!"
```

Modu hauetara deitu diezaiokegu:

```
txoria(13000)  
txoria(ekintza = "DANBA!", tentsioa = 1000)  
txoria('bi mila', egoera = 'antzarrak perratzen')  
txoria('milioika', 'hilda', 'salto egin')
```

Honako hauek berriz ez lirateke ondo egongo:

```
txoria() # Derrigorrezko argumentu bat falta da  
txoria(tentsioa=5.0, 'hilda') # Gako argumentua argumentu ez-gakoaren  
# ondoren  
txoria(110, tentsioa=220) # Argumentuaren balioa bikoiztua  
txoria(hegokopurua='bat') # Gako ezezaguna
```

Orokorrean, argumentu zerrenda batek argumentu posizionalak gakoaren argumentuz jarraiturik izango ditu, non gakoak parametro formalen izenen arabera aukeratuko diren. Parametro formalek balio lehenetsia izan edo ez izateko aukera dute. Argumentuek ezin dute balio bat baino gehiago jaso (argumentu posizionalei dagozkien parametro formalen izenak ezin dira erabili funtzioari deitzean). Hona hemen arazo honexengatik errorea ematen duen adibide bat:

```
>>> def funtzioa (a):  
...     pass  
...  
>>> funtzioa(0, a=0)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
TypeError: funtzioa() got multiple values for keyword argument 'a'
```

Azken parametro formalak ****izena** egitura duenean parametro formalekin bat ez datozen gako argumentu guztiak dituen hiztegi bat jasoko du. Hau ***izena** egitura duen parametro formal batekin konbinatu daiteke, honek parametro formalen zerrendatik at dauden tupla jasoko du. Beti ere ***izena**-ak ****izena**-ren aurretik agertu beharko du. Adibidez:

```
def garagardotegia(mota, *argumentuak, **gakoak):
    print "-- Ba al duzu", mota, "-ik?"
    print "-- Barkatu, ez dugu", mota, "-ik"
    for arg in argumentuak: print arg
    print '-'*30
    for kw in gakoak.keys(): print kw, ':', gakoak[kw]
```

Honela deitu diezaiokegularik:

```
garagardotegia ('duff', "Samielak ditugu, jauna.",
                "Zuk halañahi izanez gero", bezeroa='dooteo',
                dendaria='axlor')
```

Honakoa erakutsiko luke:

```
-- Ba al duzu duff -ik?
-- Barkatu, ez dugu duff -ik
Samielak ditugu, jauna.
Zuk halañahi izanez gero
-----
bezeroa : dooteo
dendaria : axlor
```

Aukerazko argumentu zerrendak

Azkenik, gutxien erabiltzen den aukera funtzio bati zehaztugabeko argumentu kopuru batekin deitu diezaiokegunekoa da. Argumentu hauek tupla batean bilduko dira. Zehaztugabeko argumentu kopuruaren aurretik argumentu arruntak izan ditzake, bat ere ez edo nahi adina.

```
def fprintf(file, formatua, *args):
    file.write(formatua % args)
```

4.1 lambda egiturak

Erabiltzaileen beharren arabera, Python-en Lisp programazio hizkuntza funtzionaletako ezau-garri batzuk gehitu dira. `lambda` hitza erabiliz funtzio ezezagun txikiak gehitu daitezke. Funtzio honek bere bi argumentuen batura itzuliko digu: `lambda a, b: a+b`. Lambda egiturak funtzio objektu bat behar direnean bakoitzean erabil daitezke. Sintaktikoki adierazpide simple batetara mugaturik daude. Funtzio habiaratuak (anidadas) bezala lambda egiturek ezin dute dutuen esparruko aldagaiei erreferentziarik egin, nahiz eta hori, lehenetsiriko argumentuen balioak erabiliz konpondu daitekeen, adibidez:

```
def muntatu_gehitzaillea(n):  
    return lambda x, gehi=n: x+gehi
```

4.2 Dokumentazio kateak

Nahiz eta ez den derrigorrezkoa, komenigarria da funtzioen lehen hilaran funtzioaren zeregina azalduko digun esaldi labur bat sartzea Maiuskulaz hasiz eta `'.'` puntuz amaituz. Funtzio baten dokumentazio katea ikusteko:

```
>>> print funtzioizena.__doc__  
Dokumentazio katea.
```

5 Moduluak

Python interpretetik irten eta berriz sartzean, lehenxeago eginiko definizioak (funtzio eta aldagaiak) galdu egin direla ohartuko zara. Horregatik, programa luzeago bat idatzi nahi bada, hobe izango da testu editore batetan prestatzea interpreterako sarrera eta fitxategi horrekin sarrera moduan exekutatzeko. Honi gidoi bat sortzea deitzen zaio. Gehitzen doazen elean fitxategi gehiagotan banatzea komeniko da hauen mantenimendua errazteko.

Honetarako, Python-ek badu definizioak fitxategi batetan utzi eta gidoi batetan edo interpretearen ekinaldi interaktibo batetan erabiltzeko modu bat. Fitxero hori *modulua* deitzen da; modulu baten funtzioak beste modulu batzuetara edo modulu *nagusira* (goragoko mailatik eta kalkulagailu moduan exekutatuak gidoi batetik sarbidedun den aldagai bilduma) *inportatu* daitezke.

Modulu bat Python sententziak eta definizioak dituen fitxategi bat da. Fitxategiaren izena moduluarena izango da **.py** atzizkiarekin. Modulu baten barruan, moduluaren izena (kate bezala) `__name__` aldagai orokorrak emango digu. Adibidez, erabili gustokoen duzun testu editorea **fib.py** izeneko fitxategi bat sortzeko uneko direktorioan, hurrengo edukinarekin:

```
# Fibonacci zenbakien modulua
def fib(n):    #ñ-rainoko Fibonacci seriea idatzi
    a, b = 0, 1
    while b <ñ:
        print b,
        a, b = b, a+b
def fib2(n):  #ñ-rainoko Fibonacci seriea itzuli
    emaitza = []
    a, b = 0, 1
    while b <ñ:
        emaitza.append(b)
        a, b = b, a+b
    return emaitza
```

Orain sartu Python-en interpretean eta inportatu modulu hau honela:

```
>>> import fibo
```

5 Moduluak

Honekin ez dira `fibon` definituriko funtzioen izenak zuzenean uneko sinboloen taulan sartuko; `fibon` moduluaren izena bakarrik. Moduluaren izena erabiliz erabil ditzakegu funtzioak:

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibon'
```

Funtzio bat askotan erabili behar bada, lekuko izen bati ezarri dakioke:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

5.1 Moduluei buruz gehiago

Modulu batek funtzioen definizioez gain, modulu haseratzeko balio duten sententziak izan ditzake. Hauek inportatzen diren lehenengo aldian bakarrik exekutatu dira.

Modulu bakoitzak bere sinbolo-etaula du, moduluan definituriko funtzio guztiek sinbolo-etaula orokor bezala erabiliko dutelarik. Ondorioz, modulu baten egileak moduluaren barnean aldagai orokorrak erabili ditzake moduluaren erabiltzaile baten aldagai orokorrekin izan ditzakeen arazoekin kezkatu gabe. Bestalde, modulu baten aldagai orokorrak aldatzeko `modIzena.elemIzena` erabil genezake.

Moduluek beste modulu batzuk inportatu ditzakete. Ez da derrigorrezko ohitura `import` sententzia guztiak modulu (edo gidoiaren) haseran jartzea. Inportaturiko moduluaren izenak inportatzen duen modulu orokorraren sinbolo-etaula orokorrean kokatzen dira.

Bada `import` sententziaren bariazio bat, non modulu baten izenak inportatzen duen moduluaren sinbolo taulara zuzenean inportatzen diren:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Honek ez du sinbolo-etaula lokalean inportatu diren elementuen moduluaren izena sartuko (adibidean `fibon` ez dago definitua).

Gainera badago modulu batek definituriko izen guztiak inportatzen dituen aldakuntza bat:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Honek azpi-marrarekin (`_`) hasten ez diren izen guztiak inportatuko ditu.

5.2 `dir()` funtzioa

`dir()` barne-funtzioak modulo batek zein izen zehazten dituen jakiteko erabiltzen da. Kate zerrenda ordenatu bat itzultzen du:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
 'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile',
 'settrace', 'stderr', 'stdin', 'stdout', 'version']
```

Argumenturik gabe unean (norberak edo sistemak) zehazturiko izen zerrenda idazten du:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Izen mota guztiak erakusten ditu: aldagai, modulu, funtzio, etab.

`dir()`-ek funtzio eta barne-aldagaien izenik ez du itzuliko. Izen hoiek lortzea nahi bada, `__builtin__` modulu estandarrean daude zehaztuta:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError',
 'IOError', 'ImportError', 'IndexError', 'KeyError',
 'KeyboardInterrupt', 'MemoryError', 'NameError', 'None',
 'OverflowError', 'RuntimeError', 'SyntaxError',
 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
 'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr',
```

5 Moduluak

```
'cmp', 'coerce', 'compile', 'dir', 'divmod', 'eval',  
'execfile', 'filter', 'float', 'getattr', 'hasattr',  
'hash', 'hex', 'id', 'input', 'int', 'len', 'long',  
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range',  
'raw_input', 'reduce', 'reload', 'repr', 'round',  
'setattr', 'str', 'type', 'xrange']
```

5.3 Paketeak

Paketeak Python-eko moduluen izendatze-formak egituratzeko bidea dira, "modulu izenak puntuaz" erabiliz. Adibidez A.B modulua izenak "A" izeneko modulu baten "B" izeneko azpi-modulu bati egiten dio erreferentzia.

Adibide bezala, soinu fitxategi eta soinu datuak lantzeko modulu talde bat (paketea) diseinatzea nahi da. Soinu fitxategi mota ezberdinak daude (luzapenez bereiztua, .wab, .aiff edo .au) eta ondorioz formatu ezberdinen artean bihurtzeko modulu taldea sortu eta mantendu beharko genuke. Soinu datuen gain eragiketa ezberdinak daude (nahastu, ohiartzuna gehitu, ekualizatu edo efektuak sartu), eragiketa hauek burutzeko hainbat modulu idazten ari garelarik. Hona hemen paketearen egituraketa posible bat:

Soinua/	Maila goreneko paketea
__init__.py	Soinuen paketea inzializatu
Formatuak/	Fitxategien formatuen eragiketaren azpipaketea
__init__.py	
irakurwav.py	
idatziwav.py	
irakuraiff.py	
idatziaiff.py	
irakurau.py	
idatziau.py	
...	
Efektuak/	Soinu efektuen azpipaketea
__init__.py	
ohiartzuna.py	
surround.py	
alderantzizkatu.py	
...	
Filtroak/	Filtroen azpipaketea

```

__init__.py
ekualizadorea.py
vocoder.py
karaoke.py
...

```

__init__.py fitxategiak Python-ek direktorioak paketeen edukiontzi bezala jokatzeko beharrezkoak dira. Honela izen arrunt dun direktorioek, adibidez "test", bilaketa bidean agertzen diren moduluak ustekabez ezkutatzea saiesten du. **__init__.py** fitxategi huts bat izan daiteke, paketearen initalizazioa landuko duen kodea eduki edo **__all__** aldagaia eguneratu dezake.

Paketearen erabiltzaileek paketearen modulu bakanak inportatu ditzakete:

```
import Soinua.Efektuak.ohiartzuna
```

Honela `Soinua.Efektuak.ohiartzuna` azpimodulua kargatu daiteke. Izen osoari egin behar ko zaio erreferentzia:

```
Soinua.Efektuak.ohiartzuna.ohiartzunfiltroa(sarrera, irteera,
      atzerapena=0.3, aten=5)
```

Azpimodulu bat inportatzeko beste modu bat:

```
from Soinua.Efektuak import ohiartzuna
```

Honela `ohiartzuna` azpimodulua ere kargatu daiteke eta paketearen aurreizena erabili gabe:

```
ohiartzuna.ohiartzunfiltroa(sarrera, irteera, atzerapena=0.3, aten=5)
```

Nahi den funtzio edo aldagaia zuzenean inportatuz:

```
from Soinua.Efektuak.ohiartzuna import ohiartzunfiltroa
```

Berrito kargatuko da `ohiartzuna` azpimodulua baina honela `ohiartzunfiltroa` funtzioa zuzenean erabilgarri dago:

```
ohiartzunfiltroa(sarrera, irteera, atzerapena=0.3, aten=5)
```

from *paketea* import *elementua* erabiltzean, *elementua* paketearen azpimodulua (edo azpipaketea) edo paketeak zehazturiko beste edozein izen, funtzio, klase edo aldagai, izan daiteke. import sententziak lehenengo *elementua* paketearen barruan al dagoen begiratzen du. Hala ez bada, modulu bat dela pentsatuz kargatzen saiatuko da. Lortzen ez badu `ImportError` espezioa jaurtiko du.

import *elementua*.*azpielementua*.*azpiaazpielementua* sintaxia erabiltzean, *elementu* bakoitza, azkena ez ezik, *paketea* izango da. Azken *elementua* *paketea* edo *modulua* izan daiteke baina ez goiko mailan zehazturiko funtzio, klase edo aldagaia.

5.3.1 Pakete batetik * inportatu

Eragiketa hau Windows-en ez da oso zuzen ibiltzen fitxategien sistemak ez baitu fitxategien miuskulataz ideia zehatzik. **OHIARTZUNA.PY** fitxategia ezin dugu jakin **ohiartzuna**, **Ohiartzuna** edo **OHIARTZUNA** bezala inportatuko duen. DOS-en berriz izen luzedun moduluentzat arazoa da 8 hizkitik ezin baita pasatu. Berez, `from Soinua.Efektuak import * landuz` paketearen dauden azpimodulu denak aurkitu eta denak inportatu beharko lituzke.

Hau konpontzeko paketearen egileak paketeari ezaugarri esplizitu bat ezartzea da. Import sententziak honako ohitura du: pakete baten `__init__.py`-ren kodeak `__all__` izeneko zerrenda bat zehazten badu, `from paketea import *` aurkitzean inportatu beharreko moduluen izen zerrenda izango da. Paketearen bertsio berri bat kaleratzean egilearen esku dago zerrenda hori eguneratzea. Adibidez **Soinua/Efektuak/__init__.py** fitxategiak hurrengo kodea izan lezake:

```
__all__ = ["ohiartzuna", "surround", "alderantzizkatu"]
```

`__all__` zehaztu gabe badago `from Soinua.Efektuak import *` sententziak ez ditu `Soinua.Efektuak` azpipaketeko modulu guztiak inportatuko uneko izendatze gunera. `Soinua.Efektuak` paketea (haseratze kodea landuz) eta paketearen zehazturiko izenak inportatzeaz arduratuko da, `__init__.py`-n zehazturiko edozein izen eta lehenagoko inport sententziez inportaturiko paketearen edozein azpimodulu inportatzeaz ere bai. Adibidez:

```
import Soinua.Efektuak.ohiartzuna
import Soinua.Efektuak.surround
from Soinua.Efektuak import *
```

Adibide honetan `ohiartzuna` eta `surround` moduluak uneko izendatze eremura inportatuko dira, `Soinua.Efektuak` paketearen zehazturik daudelako `from ... import sententzia` lantzean (`__all__` zehaztua badago ere balio du).

6 Sarrera irteerak

Programa baten irteerak erakusteko modu ezberdinak daude; gizakiek ulertzeko moduan datuak inprimatu edo geroago erabiltzeko fitxategi batean idatzi daitezke. Kapitulu honetan aukera batzuk ikusiko ditugu.

6.1 Irteera formatu hobetua

Orain arte bi modu ezberdin erabili ditugu balioak idazteko: *expresio sententziak* eta `print` sententzia. Hirugarren modu bat ere badago: fitxategi objetuen `write()` metodoa, `sys.stdout-i` esker da zilegi.

Irteerei formatua emateko bi modu daude: lehenengoan norberak eratu beharko ditu kateak, mozketak eta elkarketak erabiliz. `string` modulu estandarrak baditu kateak zutabe zabalera jakin batzuetara egokitzeko eragiketak. Bigarrenengo modua `%` operadorea ezker argumentu bezala kate batekin erabiltzea litzateke. `%` operadoreak ezker argumentua C-ko `sprintf()` formatu estiloko kate bat bezala azalduko du eta eskuin argumentuari aplikatu beharko zaio formatuaren emaitz katea itzultzeko.

Python-ek balioak kateetara itzultzeko `repr()` funtzioa du, edo komatxo simple alderantzizkatu artean idatziz balioa. Adibide batzuk:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = '"x"-en balioa: ' + 'x' + ' eta "y"-rena ' + 'y' + '...'
>>> print s
"x"-en balioa: 31.4 eta "y"-rena 40000...
>>> # Komatxo alderantzizkatuek zenbakiez gain beste mota batzuei
ere eragiten die:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.4, 40000]'
>>> # Katera itzultzeak kateei komatxo eta alderantzizko barrak
gehitzen dizkie:
```

6 Sarrera irteerak

```
... kaixo = 'kaixo mundua\n'  
>>> katkaixo = 'kaixo'  
>>> print cadhola  
'kaixo mundua\012'  
>>> # Komatxo alderantzizkatuen argumentu tupla bat ere izan daiteke:  
... 'x, y, ('euria', 'elurra')'  
"(31.4, 40000, ('euria', 'elurra'))"
```

Hona hemen karratu eta kuboak idatziko dituzten bi modu:

```
>>> import string  
>>> for x in range(1, 11):  
...     print string.rjust('x', 2), string.rjust('x*x', 3),  
...     # Ohar zaitez aurreko lerroko amaierako komaz  
...     print string.rjust('x*x*x', 4)  
...  
1   1   1  
2   4   8  
3   9  27  
4  16  64  
5  25 125  
6  36 216  
7  49 343  
8  64 512  
9  81 729  
10 100 1000  
>>> for x in range(1,11):  
...     print '%2d %3d %4d' % (x, x*x, x*x*x)  
...  
1   1   1  
2   4   8  
3   9  27  
4  16  64  
5  25 125  
6  36 216  
7  49 343  
8  64 512  
9  81 729  
10 100 1000
```

`print`-ek bere argumentuen artean zuriune bat gehitzen du. Adibide honetan `string.rjust()` erabili da katea eskubiruntz egokitzeko ezker aldean zuriuneak gehituz, `string.ljust()` eta `string.center()` ere erabili genitzazke. `string.zfill()` zenbakizko katea ezker aldetik zeroz betetzeko erabiliko da:

```
>>> import string
>>> string.zfill('12', 5)
'00012'
>>> string.zfill('-3.14', 7)
'-003.14'
>>> string.zfill('3.14159265359', 5)
'3.14159265359'
```

`%` operadorearen erabilera:

```
>>> import math
>>> print 'PI-ren balioa gutxigorabehera: %5.3f.' % math.pi
PI-ren balioa gutxigorabehera: 3.142.
```

Katean formatu bat baino gehiago badago, tupla bat pasa beharko zaio eskuin eragingai bezala:

```
>>> taula = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for izena, telef in taula.items():
...     print '%-10s ==> %10d' % (izena, telef)
...
Jack          ==>         4098
Dcab          ==>         7678
Sjoerd        ==>         4127
```

Formatu gehienek C-n bezala jokatzen dute eta modu zehatza pasa beharko zaie, horrela ez bada exzepzio jaurtiko da. `%s`-ri ordea ez badiogu kate bat bidaltzen `str()` barne-funtzioa erabiliz katera itzuliko du. `*` argumentu (osoa) bezala pasatuz gero zabalera edo zehaztasuna adierazi genezake.

Oso luzea den eta banatzea nahi ez den formatuko katea izatean, aldagaiei erreferentzia bere izenez egitea komeni da, hau lortzeko `%(izena)` formatua erabil daiteke:

```
>>> tabla = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % tabla
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

6.2 Fitxategien idazketa eta irakurketa

`open()`-ek fitxategi objektu bat itzuliko digu. Bi argumenturekin erabili ohi da:

```
open(fitxategiIzena, modua).
```

```
>>> f=open('C:/fitxategiizena', 'w')
>>> print f
<open file 'C:/fitxategiizena', mode 'w' at 80a0960>
```

Lehen argumentuak fitxategiaren izena izango du eta bigarrenak fitxategi hau erabiliko den modua. Modua `'r'` (ikakurtzeko bakarrik), `'w'` (idazteko bakarrik, izen bereko beste bat badago ezabatu egingo du) eta `'a'` (fitxategiari zerbait gehitzeko) izan daiteke. `'a'` moduan idazten den edozer lehendik dauden datuen atzetik idatziko da. Fitxategiaren izena bakarrik pasatuz gero argumentu bezala, `'r'` moduan irekiko du.

Windows eta Macintosh-en moduari `'b'` gehituz fitxategia modu bitarrean irekiko du, ondorioz `'rb'`, `'wb'` edo `'r+b'` moduak existitzen dira. Windows-ek testu-fitxategi eta fitxategi-bitarren artean ezberdintasunak ditu: fitxategien lerro amaierako karaktereak arinki aldatuak izango dira automatikoki, irakurtzean edo datuak idaztean. Ezkutuko aldaketa hauek ASCII testudun fitxategien gain ez dute eraginik, modu bitarreko (JPEG edo .EXE) fitxategiek ez bezala.

6.2.1 Fitxategi objektuen metodoak

read(kopurua)

Fitxategitik datu kopuru bat irakurri eta kate bezala itzuliko ditu. *Kopurua* argumentua aukeakoa da, honen faltan edo negatiboa bada, fitxategi osoa irakurriko du. Argumentua positiboa bada gehienez byte kopuru hori irakurri eta itzuliko du. Iada fitxategi amaierara iritsi bada kate hutsa ("") itzuliko du.

```
>>> f.read()
'Hau da fitxategi osoa.\012'
>>> f.read()
''
```

readline()

Fitxategiko lerro bakarra irakurtzen du. Katearen bukaeran lerro aldaketako karaktere bat (`\n`) uzten da, azken lerroan bakarrik kentzen delarik, beti ere fitxategia lerro-jauzi batean amaitzen ez bada. `readline()`-ek kate hutsa itzultzen badigu fitxategia amaierara iritsi dela esan nahi du eta fitxategiko lerro hutsa, lerro-jauzia bakarrik duena, `'\n'` bitartez adieraziko da.

```
>>> f.readline()
'Fitxategiaren lehen lerroa.\012'
>>> f.readline()
'Fitxategiaren bigarren lerroa.\012'
>>> f.readline()
''
```

readlines()

Fitxategiaren lerro guztiak dituen zerrenda bat itzuliko digu. Nahi izanez gero *sizehint* parametroa pasa geniezaiokegu, byte kopuru hori irakurriko du eta lerroa amaitu arte jarraitu, itzuliko dizkigun lerroak osorik egongo dira.

```
>>> f.readlines()
['Fitxategiaren lehen lerroa.\012', 'Fitxategiaren bigarren lerroa.\012']
```

write(*katea*)

Katearen edukia fitxategian idatzi eta None itzuliko du.

```
>>> f.write('Frogak egiten...\n')
```

tell()

Fitxategian fitxategi objektuak duen posizioa itzultzen du, honen haseratik bytetan nehurtua.

seek(*desplazamendua*, *nondik*)

Fitxategi objektuaren posizioa aldatzeko erabiltzen da, *nondik* argumentua hasera bezala hartuta *desplazamendua* aplikatuz lortzen den posiziora. *nondik* 0 erabiliz fitxategi haseratik hasiko da kontatzen, 1 erabiliz uneko posiziotik eta 2 erabiliz fitxategi amaiera hartuko da erreferentzi bezala.

```
>>> f=open('/tmp/fichTrabajo', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)      # Fitxategiaren 5. bytera joan
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Amaiera aurretik hirugarren bytera joan
>>> f.read(1)
'd'
```

close()

Fitxategiaren erabilera amaitzean hau isteko eta sistemaren errekurtsioak askatzeko erabiltzen da. `close()` egin ostean fitxategi objektua erabiltzeko saiakera guztiek huts egingo dute.

```
>>> f.close()
>>> f.read()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

6.2.2 pickle modulua

Fitxategietan kateak ez diren datu mota ezberdinak erabilera errazteko modulua da. Python-eko ia edozein objektu hartu eta kate modura pasatzen ditu. Prozesu honi *zamatzea* deitzen zaio, eta kontrako prozesuari *arintzea*. Zamatze eta arintzearen artean, objektua irudikatzen duen katea fitxategi batean, memorian gorde edo sare konexio baten bitartez hurruneke makina batetara bidali ahal izango da.

`x` objektu bat eta `f` lehendik idazteko moduan irekitako fitxategi bat baditugu objektua zamatze-ko:

```
pickle.dump(x, f)
```

Eta arintzeko:

```
x = pickle.load(f)
```

Adibidea:

Hiztegien ataleko adibide berbera da, baina telefono zenbakiak fitxategi batetan gorde edo idazteko gaitasunarekin:

```
#
# Zenbakiakfitx.py fitxategiaren hasera
#
import string
egia = 1
gezurra = 0
def erakutsi(zenbakiak):
    print "Telefono zenbakiak:"
```

```

for x in zenbakiak.keys():
    print "Izena: ",x," \tZenbakia: ",zenbakiak[x]
print
def gehitu(zenbakiak,izena,zenbakia):
    zenbakiak[izena] = zenbakia
def bilatu(zenbakiak,izena):
    if zenbakiak.has_key(izena):
        return "Zenbakia: "+zenbakiak[izena]
    else:
        return "Ezin izan dut aurkitu "+izena
def ezabatu(zenbakiak,izena):
    if zenbakiak.has_key(izena):
        del zenbakiak[izena]
    else:
        print "Ezin izan dut aurkitu ", izena
def kargatu(zenbakiak,fitxizena):
    fitx_sarrera = open(fitxizena,"r")
    while egia:
        sarrera = fitx_sarrera.readline()
        if sarrera == "":
            break
        sarrera = sarrera[:-1]
        [izena,zenbakia] = string.split(sarrera,",")
        zenbakiak[izena] = zenbakia
    fitx_sarrera.close()
def gorde(zenbakiak,fitxizena):
    fitx_irteera = open(fitxizena,"w")
    for x in zenbakiak.keys():
        fitx_irteera.write(x+", "+zenbakiak[x)+"\n")
    fitx_irteera.close()
def menua():
    print
    print '1. Zenbakiak ikusi'
    print '2. Zenbaki bat gehitu'
    print '3. Zenbaki bat ezabatu'
    print '4. Zenbaki bat bilatu'
    print '5. Zenbakiak kargatu'

```

6 Sarrera irteerak

```
print '6. Zenbakiak gorde'
print '7. Irten'
print
telzerrenda = {}
aukera= 0
while aukera!= 7:
    menua()
    aukera = input("Zerñahi duzu egitea? ")
    if aukera == 1:
        erakutsi(telzerrenda)
    elif aukera == 2:
        print "Sartu izena eta zenbakia"
        izena = raw_input("Izena:")
        zenbakia = raw_input("Zenbakia:")
        gehitu(telzerrenda,izena,zenbakia)
    elif aukera == 3:
        print "Ezabatu izena eta zenbakia"
        izena = raw_input("Izena:")
        ezabatu(telzerrenda,izena)
    elif aukera == 4:
        print "Bilatu zenbakia"
        izena = raw_input("Izena:")
        print bilatu(telzerrenda,izena)
    elif aukera == 5:
        fitxizena = raw_input("Kargatzeko fitxategia:")
        kargatu(telzerrenda,fitxizena)
    elif aukera == 6:
        fitxizena = raw_input("Gordetzeko fitxategia:")
        gorde(telzerrenda,fitxizena)
    elif aukera == 7:
        pass
    else:
        menua()
print "Agur ta hurrenarte"
# Zenbakiakfitx.py fitxategiaren amaiera
```

7 Erroreak eta eszepzioak

Badira (gutxienez) bi errore mota ezberdin: *sintaxi erroreak* eta *eszepzioak*.

7.1 Sintaxi erroreak

Interprete sintaktikoak akatsa izan duen lerroa errepikatuko du errorea eman deneko lehen puntua gezitxo batez seinalatuko du. Errorearen arrazoia gezitxoaren aurreko sinboloan egongo da:

```
>>> while 1 print 'Iepa mundua!'
      File "<stdin>", line 1
        while 1 print 'Iepa mundua!'
                ^
SyntaxError: invalid syntax
```

Adibide honetan, errorea `print` hitz-gakoan antzeman da, honen aurretik bi puntu (:) falta direlako. Fitxategiaren izena eta lerro zenbakia erakusten ditu non bilatu jakiteko, sarrera fitxategi batetik etorritz gero.

7.2 Eszepzioak

Nahiz eta sententzia edo espresio bat sintaktikoki zuzen egon, hau exekutatzean errore bat sortu liteke. Exekuzioan sortzen diren erroreak eszepzioak deitzen dira eta ez dira derrigorrez itzulezinak. Harrapatu ez diren eszepzioek ondorengoa bezalako errore mezuak sortzen dituzte:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
>>> 4 + fantomas*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

7 Erroreak eta eszepzioak

```
NameError:ñame 'fantomas' isñot defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

Errore mezua azken erroak zer gertatu den azaltzen du. Eszepzioak mdu ezberdinetakoak izan daitezke eta mezua zati bezala azalduko dira, adibideetakoak: `ZeroDivisionError`, `NameError` eta `TypeError`. Eszepzio estandarren izenak barne-identifikadoreak dira (ez hitz-erreserbatuak).

7.3 Eszepzioen kudeaketa

Hurrengo adibidean erabiltzaileak zenbaki bat sartu arte datuak eskatzen jarraituko du:

```
>>> while 1:
...     try:
...         x = int(raw_input("Sartu zenbaki bat: "))
...         break
...     except ValueError:
...         print "Ez da zenbakia. Saiatu berriro..."
... 
```

`try` sententziaren erabilera:

- Lehenengo *try* klausula (`try` eta `except`-en arteko sententziak) exekutatu da.
- Eszepziorik jaurtitzen ez bada *except* klausula ez da landuko eta `try` sententziaren exekuzioa amaitu egingo da.
- `try` klausula lantzen ari den bitartean eszepzio bat jaurtitzen bada klausulatik zuzenaen saltatu egingo du. Ondoren `except` hitz-gakoaren ondoren aipatzen den eszepzio motarekin bat baldin badator bere mota `except` klausula exekutatu da eta exekuzioak `try` sententziaren atzetik jarraituko du.
- `except` klausulan aipaturiko eszepzioarekin bat ez datorren eszepzio bat jaurtitzen bada kanpotik habituriko `try` sententzitarra joko du. Eszepzio kudeatzailerik aurkitzen ez bada *ezusteko eszepzio* bilakatzen da eta exekuzioa lehen ikusi dugun moduko mezu batekin amaituko da.

`try` sententziak `except` klausula bat baino gehiago izan ditzake, eszepzio ezberdinak harrapatzeko. Eszepzio batentzat kudeatzaile bakarria exekutatu da. Kudeatzaileek kasuan kasuko `try` klausulan saltatzen duten eszepzioak bakarrik harrapatzen dituzte, ez `try` sententzia bereko beste kudeatzaileetan. `Try` klausula batek eszepzio bat baino gehiago harrapatu ditzake, zerrenda baten izendatuz:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Azken `except` klausulak ez du zertan eszepziorik izendatu behar, honela edozein eszepzio harrapatu dezakelarik. Errore mezu bat erakusteko eta eszepzioa berriro jaurtitzeko (hotsegiten duenatariko batek kudeatuko duelarik eszepzioa) erabili daiteke.

```
import string, sys
try:
    f = open('fitxategia.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "S/I errorea(%s): %s" % (errno, strerror)
except ValueError:
    print "Ezin izan dira datuk osora itzuli."
except:
    print "Errore ezezaguna:", sys.exc_info()[0]
    raise
```

`try ... except` sententziak aukerazko `else` klausula du, `except` klausularen ondoren azalduko da. `Try` klausulak eszepziorik jaurtitzzen ez duen kasuan erabiltzen da:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'Ezin da ireki', arg
    else:
        print arg, '-ek', len(f.readlines()), 'lerro ditu'
        f.close()
```

Eszepzio bat saltatzean, *eszepzioaren argumentu* bezala ezagutzen den balio elkartu bat izango du. Balio hau izan edo ez eta honen mota eszepzio motaren gain daude. Argumentu hau duten

7 Erroreak eta eszepzioak

eszepzioetan `except` klausulak eszepzio izenaren (edo zerrendaren) ondoren aldagai bat adierazi dezake eta argumentuaren balioa jasoko du:

```
>>> try:
...     fantomas()
... except NameError, x:
...     print x
...
name 'fantomas' is not defined
```

Harrapatu ez den eszepzio batek argumentua badu errore mezuaren amaieran azalduko da.

Eszepzio kudeatzaileek ez dituzte `try` klaulan jaurti direlako bakarrik kudeatuko, `try` klausulan deitzen (zuzenean edo zeharka) zaien funtzioetan saltatzen badute ere kudeatuko dituzte.

Adibidez:

```
>>> def fun():
...     x = 1/0
...
>>> try:
...     fun()
... except ZeroDivisionError, xehetasuna:
...     print 'Errore kudeaketa:', xehetasuna
...
Errore kudeaketa: integer division or modulo by zero
```

7.4 Eszepzioak jaurtiaz

`raise` (saltarazi) sententzia erabiliz programatzaileak eszepzio bat azalduarazi dezake. Adibidez:

```
>>> raise NameError, 'Atsaldeon'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Atsaldeon
```

7.5 Erabiltzaileak zehazturiko eszepzioak

Programek beraien eszepzioak izendatu ditzakete aldagai bat kate bat asignatuz edo eszepzio mota berri bat sortuz. Adibidez:

Lehen argumentuak zein eszepzio jaurti behar duen adierazten du eta bigarrenak (aukerazkoa) eszepzioaren argumentua.

```
>>> class NireErrorea:
...     def __init__(self, balioa):
...         self.valor = balioa
...     def __str__(self):
...         return 'self.balioa'
...
>>> try:
...     raise NireErrorea(2*2)
... except NireErrorea, e:
...     print 'Nire eszepzioak saltatu du, balioa:', e.balioa
...
Nire eszepzioak saltatu du, balioa: 4
>>> raise mi_exc, 1
Traceback (innermost last):
  File "<stdin>", line 1
mi_exc: 1
```

7.6 Garbiketa ekintzak

Edozein momentutan exekutatu beharko diren garbiketa ekintzak zehazteko balio duen aukerazko beste klausula bat du `try` sententziak:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Aio danoi!'
...
Aio danoi!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
KeyboardInterrupt
```

`Finally` klausula beti (eszepzioak jaurti edo ez) exekutatu da. `try` sententzia `break` edo `return` erabiliz uzten bada ere `finally` exekutatu egingo da.

`try` sententzia batek `except` klausula bat edo gehiago edo `finally` klausula eduki beharko du, baina ez biak.

7 Erroreak eta eszepzioak

8 Klaseak

Python-en, C++-eko terminologia erabiliz, klaseko atalak (datuak barne) *publicoak* dira eta atal funtzio guztiak *virtualak*. Ez dago eraikitzaile eta destruktore berezirik. Metodo funtzioa, objektua adierazten duen lehen argumentu esplizituarekin zehazten da eta funtzioari inplizituki deitzen laguntzen du. Klaseak objektuak dira, Python-en datu mota guztiak diren bezala. "Objetu" hitzak ez du derrigorrez klase baten instantzia denik esan nahi. Nahiz eta datu mota guztiak ez diren klaseak (fitxategi, zerrenda edo osoak), denek dute semantika zati bat berdina. Sintaxi berezia duten barne operadore gehienak klaseetan birzehaztu daitezke.

Objetuek banakotasuna dute. Izen bat baino gehiago izan ditzake objektu batek, "alias"ak sortzea deitzen zaio honi beste hizkuntza batzuetan. Kasu batzutan alias-as erakusle bezala jokatzen dute. Adibidez, objektu bat pasatzea erraza da, erakuslea bakarrik pasatzen baita. Funtzioak argumentu bezala pasatako objektua aldatzen badu, funtzioari deitzen dionak ikusiko ditu aldaketok.

8.1 Izen eremu eta esparruak

Klaseen erabilera ulertzeko beharrezkoa da izen taldeek nola jokatzen duten jakitea.

Izen eremu bat izenak eta objektuen arteko egokitzapena da. Gehienak hiztegi bezala inplementatzen dira. Izen eremuen adibideak:

- Barne izen taldeak (`abs()` bezalako funtzioak eta barne eszepzioak).
- Modulu baten izen orokorrak.
- Izen lokalak funtzio bati deitzean.

Objetu baten atributuen (puntu baten atzetik datorren edozein izen. Adibidez, z.errealaren adierazpenean errealaren z objektuaren atributu bat da) taldeek ere izen esparru bat osatzen dute. Izen eremuetan ez dago inungo erlaziorik eremu ezberdinetako izenen artean, bi moduluk izen bereko funtzioak zehaztu ditzakete nahasteko arriskurik gabe (izena moduluen izenaren ondoren baitator).

Atributuak soilik irakurtzeko edo irakurri/idazteko izan daitezke. Atributuei balioak asignatu dakizkiokete. Moduluen atributuak irakurri/idazteko dira eta `del` sententzia erabiliz ezabatu daitezke.

8 Klaseak

Izen eremuak momentu ezberdinetan sortzen dira eta bizi-denbora ezberdina dute. Barne izenak dituen izen eremua Python interpretea abiaraztean sortzen da eta ez da ezabatzen. Modulu baten izen eremu orokorra moduluaren definizioa irakurtzean sortuko da eta interpretetik irten arte iraungo dute. Eta funtzio baten izen eremua funtzioari deitzean sortu eta irtetean ezabatuko da. Deitura errekurtsiboek bakoitzak bere izen eremua sortuko du.

Esparru bat (*ambito*) Python programa baten gune testual bat da, non izen eremuara zuzenean iritsi daiteken. Kontestu honetan kalifikatu gabeko (punturik gabe) izen bat izen eremuan bilatzen saiatuko da.

Nahiz eta esparruak estatikoki finkatzen diren, dinamikoki erabiltzen dira. Exekuzioaren edozein puntutan hiru esparru habitua daude, zuzenean heldu daitezken hiru izen eremu: barne esparrua (lehenengo hemen bilatzen da izena, izen lokalak ditu), erdiko esparrua (hemen jarraituko du bilaketa, moduluaren izen orokorrak izango ditu) eta kanpo esparrua (bilaketan azkena, barne izenak ditu).

Esparru lokalak lantzen ari den funtzioaren izenak edukiko ditu. Funtzioetatik kanpo, esparru lokalak esparru orokorraren izen eremu berari egiten dio erreferentzia: moduluaren izen eremua. Klaseen definizioek esparru lokalean beste izen eremu bat hartzen dute.

Modulu baten zehazturiko funtzio baten esparru orokorra modulu beraren izen eremua da. Bestalde, izenen benetako bilaketa dinamikoki egiten da, exekuzio garaian. Baina hizkuntzaren definizioak izenen ebazpen estatikora garamatza.

Asignazioak beti barnekoeneko esparruan doaz. Asignazioek ez dute daturik kopiatzen, soilik izenak objetuei estekatzen dizkie eta berdin ezabatzean (`del x` sententziak `x`-en esparru lokalak erreferentzia egiten dion izen eremuko esteka kenduko du.

8.2 Klaseen definizioen sintaxia

Klase baten definizioaren itxurarik sinpleena:

```
class klaseIzena:
    <sententzia-1>
    .
    .
    .
    <sententzia-N>
```

Normalean klase baten definizio barneko sententziak funtzioen definizioak izango dira, nahiz eta beste sententzi mota batzuk ere zilegi diren. Klase baten definizioan sartzean, izen eremu berri bat sortzen da eta esparru lokal bezala erabiliko du. Klase baten definizioa arruntki uztean (azken lerroa amaitzean) klaseko objektu bat sortuko da. Klasearen definizioan sartzean dagoen

esparru lokala berriro instantziatua izango da eta klase objektua funtzioaren goiburuan emaniko klase izenarekin estekatuko da.

8.3 Klase objektuak

Klaseko objektuek bieragiketa mota jasan ditzakete: atributuei erreferentzia eta instantziazioa.

Atributuei erreferentziek Python-eko sintaxi estandarra erabiltzen dute: `obj.izena`. Atributuentzat balio duten izenak klasea sortzean klasearen izen eremuan zeuden izen guztiak izango dira. Ondorioz klasearen definizioa:

```
class klaseIzena:
    "Adibiderako klasea"
    i = 37
    def f(x):
        return 'iepa mundue!'
```

`klaseIzena.i` eta `klaseIzena.f`, atributuei erreferentzia posibleak dira, oso bat eta metodo objektu bat itzuliko dituzte. `i` atributuari balioaren bat asignatu geniezaiokegu. Dokumentazio katea ikusteko `__doc__` erabiliko dugu.

Klaseen *instantziazioak* funtzioen notazioa erabiltzen du:

```
x = klaseIzena()
```

Klasearen instantzia berri bat sortu eta `x` aldagai lokalari asignatuko dio. Instantziazioak objektu huts bat sortzen du. Klase askok objektuak haserako egoera jakin batean sortzen dituzte `__init__()` erabiliz, honela klaseen instantziazioak honi deituko dio automatikoki:

```
def __init__(self):
    self.hustu()
```

Metodo honek argumentuak ere izan litzake. Adibidez:

```
>>> class Konplexua:
...     def __init__(self, zatiErreala, zatiIrudikaria):
...         self.r = zatiErreala
...         self.i = zatiIrudikaria
...
>>> x = Konplexua(3.0,-4.5)
>>> x.r, x.i
(3.0, -4.5)
```

8.4 Instantzia objektuak

Instantzia objektuekin atributuei erreferentzia bakarriak egin geniezaiokegu. Bi atributu izen mota bakarrik daude: metodoak eta datuen atributuak.

Datuen atributuak deklaratu beharrik ez dago aldagai lokalak bezala.

Metodoak objektuei dagozkien funtzioak dira.

Objetu instantzia baten metodoen izenak klasearen arabekoak izango dira. Funtzio objektuak diren klase baten atributu guztiak instantziei dagozkien metodoak zehazten dituzte. Adibidez, `klaseIzena.f` funtzioa denez `x.f` metodo bati erreferentzia da, baina ez `x.i`, `klaseIzena.i` ez delako funtzioa. `x.f` ez da `klaseIzena.f`-ren berdina, *objetu metodoa* baita eta ez objektu funtzioa.

8.5 Metodo objektuak

`klaseIzena` klasearen adibidean `x.f()`-ri deituz 'iepa mundue!' itzuliko liguke. Baina ez beharrezkoa bereala deitzea: `x.f` metodo objektua denez, gorde eta geroago berreskuratu bait genezake, adibidez:

```
xf = x.f
while 1:
    print xf()
```

'iepa mundue!' erakutsiko digu geuk geldiarazi arte.

`f` metodoaren definizioak argumentu bat behar zuela ohartuko zinen. Deitzen duen objektua pasako zaio lehen argumentu bezala. Orokorrean, metodo bati argumentu zerrenda batekin deitzea, metodo objektua haserako argumentu zerrendan sartuz lortuko genuken argumentu zerrendarekin deitzearen berdina da.

Funtzio objektua den klaseko atributua erakusten badu izenak, metodo objektu bat sortuko da objektu instantzia eta objektu abstraktu batean aurkituriko objektu funtzioa (metodo objektua) bateratuz.

Adibidea:

```

#
# Ajedreza.py fitxategiaren hasera
#
class alfil:
    "Alfilaren klasea"
    def __init__(self, x, y):
        self.posX=x
        self.posY=y
    def pX(self):
        return self.posX
    def pY(self):
        return self.posY
    def mugitu(self,x,y):
        if ((x<=0)or(x>8)or(y<=0)or(y>8)):
            return False
        import math
        if (abs(self.posX-x)!=(abs(self.posY-y))):
            return False
        if ((self.posX==x)and(self.posY==y)):
            return False
        self.posX=x
        self.posY=y
        return True
print '\nAlfilaren jokuan hastera goaz\n'
auk='b'
pieza=alfil(1,1)
while ((auk=='b')or(auk=='B')):
    print 'Alfilaren posizioa:',pieza.pX(),',',pieza.pY()
    x=int(raw_input("Zein X-en posizioraÑahi duzu mugitu? "))
    y=int(raw_input("Zein Y-en posizioraÑahi duzu mugitu? "))
    if (pieza.mugitu(x,y)):
        print 'Mugitu dut.'
    else:
        print 'Ezin da mugimendu hori burutu'
        auk=raw_input('Beste mugimendu saiakerarikÑahi? (b/e) ')
# Ajedreza.py fitxategiaren amaiera

```

8.6 Herentzia

Erorritako klase baten definizioaren sintaxiak honako itxura du:

```
class klaseErorriIzena (klaseOinarriIzena):  
    <sententzia-1>  
    .  
    .  
    .  
    <sententzia-N>
```

klaseOinarriIzena izena klase eratorria zehaztu den esparruan zahaztu beharko da. Klase oinarriaren izen baten ordezkari bat idatzi liteke, adibidez klase oinarria beste modulu batetan dagoenean zehaztua:

```
class klaseErorriIzena (modIzena.klaseOinarriIzena)
```

Klase eratorriaren definizioaren exekuzioa klase oinarriaren modu berean emango da, eta klase objektua sortzean klase oinarria gogoratuko da. Klasean eskatzen den atributu bat aukitzen ez bada, klase oinarrian bilatuko da, eta honela jarraituko da klase oinarria eratorria bada.

klaseErorriIzena() landuz klasearen instantzia berri bat sortuko da. Metodoei erreferentziak egiteko: dagokion klaseko atributua bilatuko du, behar izanez gero klase oinarrietatik jeitxiz, eta metodoaren erreferentzia ondo egongo da modu honetan funtzio objektu bat lortzen bada.

Klase eratorriek klase oinarrietan zehaztutako metodoak berriro definitu ditzakete. Metodoek objektu bereko beste metodoei deitzean ez dutenez lehentasun berezirik, klase oinarri batean zehazturiko metodo batek klase oinarri bereko metodo bati deitzean, berriro definituko duen klase eratorri batetako metodo bati deitzen amaitu lezake.

Erorritako klase batetan metodo bat berriro definitzean, baliteke ordeztu beharrean gehiago sakontzea klase oinarriko izen bereko metodoa. Klase oinarriko metodoari zuzenean deitzeko: klaseOinarriIzena.metodoIzena(self, argumentuak) erabiliz.

8.6.1 Herentzia anizkoitza

Klase oinarri anitzdunetatik eratorritako klase baten definizioaren sintaxiak honako itxura du:

```
class klaseErorriIzena (Oinarria1, Oinarria2, Oinarria3):  
    <sententzia-1>  
    .  
    .
```

<sententzia-N>

Klasearen atributuen erreferentzietan lehenengo sakoneran biatuko da eta gero ezkerreki eskubira. Honela, `klaseEratorriIzena` atributu bat bilatzean `Oinarriale`n bilatuko da, `Oinarriale`n klase oinarrietan eta aurkitzen ez bada `Oinarria2n`, honen klase oinarrietan eta honela hurrenez hurren. Lehenengo sakoneran bilatzeak `Oinarri`learen atributu zuzen eta eratorrien artean ez ezberdintasunik egingo.

Herentzia anizkoitzaren arazo bat klase oinarrietako bat berdina duten bi klasetatik eratorritako klaseena da. Nahiz eta erraza izan, kasu honeta zer gertatuko den jakitea (instantziak "instantzia aldagaien" edo elkarbanatzen duten oinarriak erabilitako datuen atributuen kopia bakarra izango du), ez da garbi ikusten.

8.7 Aldagai pribatuak

Modu mugatu batetan klaseko identifikadore pribatuak erabili daitezke. `__fantomas` formako edozein identifikadore `_klaseIzena__fantomas`-ekin testualki ordezkatzeko da, non `klaseIzena` uneko klasea den haserako azpi-marrak kenduz. Berrito idazte hau indentifikadorearen posizio sintaktikoa kontutan hartu gabe egingo da, modu pribatuan klase eta instantziako aldagaiak, metodoak eta aldagai orokorrak zehazteko erabili daitezke larrik. Beste klase batzuetako instantzietan klase honetako instantzia pribatuko aldagaiak gordetzeko ere balio du. Baliteke berrito idatzitako izenak 255 karaktere baino gehiago izatean izenak moztea. Klasetatik kanpo edo klasearen izenak azpi marrak bakarrik dituenean, izenak ez dira berrito idatziko.

Izenen berridazpenak klaseei metodoak eta instantzia aldagai "pribatuak" zehazteko modu errazagoa ematen dute, klase eratorrietan zehazturiko instantzia aldagaietaz ahaztuz. Aldagai pribatuak aldatu edo irakurri liteke, nahiz eta zaila izan. Akats txiki bat dago: klase oinarriaren izen berdinarekin eratortzean klasea bat, klase oinarriaren aldagai pribatuak erabiltzea posible da.

Hona hemen `__getattr__` eta `__setattr__` bere metodoak inplementatzen dituen eta atributu denak aldagai pribatuetan gordeko dituen klase bat:

```
class atributuBirtualak:
    __vdic = None
    __vdic_izena = locals().keys()[0]
    def __init__(self):
        self.__dict__[self.__vdic_izena] = {}
    def __getattr__(self, izena):
        return self.__vdic[izena]
```

8 Klaseak

```
def __setattr__(self, izena, balioa):
    self.__vdic[izena] = balioa
```

8.8 Amaitzeko

Askotan Pascal-eko "record" edo C-ko "struct"-en antzeko datu egiturak erabiltzea oso baiogarria da. Hau lortzeko klase huts baten definizioa erabili genezake, adibidez:

```
class Langilea:
    pass
ion = Langilea() # Langile fitxa huts bat sortzen da
# Fitxaren eremuak beteko ditugu
ion.izena = 'Ion Elorza'
ion.departamentua = 'Kalkulu zentrua'
ion.soldata = 2000
```

8.8.1 Eszepzioak ere klaseak izan daitezke

erabiltzaileak zehaztutako eszepzioak iada ez daude testu-kate objektuetara mugatuta; klaseen bitartez ere identifikatu daitezke. Mekanismo hauek erabiliz eszepzioen jerarkia zabalkorra sortu liteke.

raise sententziaren bi modu berri:

```
raise Klasea, instantzia
raise instantzia
```

Lehenengo moduan, instantziak Klasearen edo bere eratorri baten instantzia izan behar du. Bigarren modua berriz `raise instantzia.__class__, instantzia-ren bidelabur` bat izango da.

Except klausula batek kate objektuak bezalaxe klaseak zenbakitu ditzake. Except klausula baten klase batek jaurti den eszepzioaren klase berekoa edo bere klase oinarrikoa bada eszepzioa harrapatuko du (alderantziz ez; klase eratorri batek ez du harrapatuko bere klase oinarrienik). Ondorengo adibidea landuz gero, B, C, D erakutsiko du, ordena horretan:

```
class B:
    pass
class C(B):
    pass
class D(C):
```

```

pass
for c in [B,C,D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"

```

Except klausulak alderantziz jarriko bagenitu, B, B, B, erakutsiko liguke, lehenengo klausulak eszepzio guztiak harrapatuko bailituzke, denen klase oinarria izateagatik.

Adibidea:

Alfilaren jokua egokitzea da, hemen alfilaz gain zaldiarekin ere jolastu genezake biak pieza klasearen eratorriak direlarik. Honez gain mugimenduen erabilera okerra eszepzioak erabiliz landuko da:

```

#
# Ajedreza01.py fitxategiaren hasera
#
class pieza:
    "Piezen klasea"
    def __init__(self, x, y):
        self.posX=x
        self.posY=y
    def pX(self):
        return self.posX
    def pY(self):
        return self.posY
class alfila(pieza):
    "Piezatik eratorritako alfilaren klasea"
    def mugitu(self,x,y):
        if ((x<=0)or(x>8)or(y<=0)or(y>8)):
            raise Taula
        import math
        if ((abs(self.posX-x)) != (abs(self.posY-y))):

```

8 Klaseak

```
        raise Mugimendua
    if ((self.posX==x)and(self.posY==y)):
        raise Batez
    self.posX=x
    self.posY=y
class zaldia(pieza):
    "piezatik eratorritako zaldiaren klasea"
    def mugitu(self,x,y):
        if ((x<=0)or(x>8)or(y<=0)or(y>8)):
            raise Taula
        import math
        if ((self.posX==x)and(self.posY==y)):
            raise Batez
        if((abs(self.posX-x)!=2)and(abs(self.posY-y)!=1)or
            (abs(self.posX-x)!=1)and(abs(self.posY-y)!=2)):
            raise Mugimendua
        self.posX=x
        self.posY=y
class Ajedreza:
    pass
class Taula(Ajedreza):
    pass
class Mugimendua(Ajedreza):
    pass
class Batez(Ajedreza):
    pass
def piezalortu():
    "Zein piezarekin jolastuko dugun zehaztuko da"
    auk=int(raw_input( "'Sartu:\n\t1 alfilarekin jolasteko\n\t2
                        zaldiarekin jolasteko\n'"'))
    if auk==1:
        pieza=alfila(1,1)
        return pieza
    elif auk==2:
        pieza=zaldia(1,1)
        return pieza
jokoan=piezalortu()
```

```

jarraitu='b'
while ((jarraitu=='b')or(jarraitu=='B')):
    print 'Zure pieza',jokoan.pX(),',',jokoan.pY(),'posizioan aurkitzen da.'
    try:
        x=int(raw_input('Zein da mugituñahi duzun X posizioa?\n'))
        y=int(raw_input('Zein da mugituñahi duzun Y posizioa?\n'))
        jokoan.mugitu(x,y)
        jarraitu=raw_input('Beste saiakerarikñahi? (b/e)\n')
    except Taula:
        print 'Pieza tulatik kanpo atera duzu!'
    except Mugimendua:
        print 'Horrela ezin duzu mugitu!'
    except Batez:
        print 'Toki berean utzi duzu!'
    except ValueError:
        print 'Zein herritan da hori zenbakia?Ñirean ez behintzat!'
# Ajedreza01.py fitxategiaren amaiera

```