

საქართველოს ტექნიკური უნივერსიტეტი

გულნარა ჯანელიძე

Python

დაპროგრამების ენა

(საბაზისო კურსი)

თბილისი

2018

უაკ 681.3.06

სახელმძღვანელოში წარმოდგენილია დაპროგრამება, თანამედროვე პერიოდში პოპულარულ და აქტიურად განვითარებად, Python ენაზე. წიგნი გათვალისწინებულია მათთვის, ვინც ფლობს დაპროგრამების რომელიმე ენის ან ალგორითმების საფუძვლებს.

წიგნს ვუძღვებ სხედროს და ანას

ISBN 978-9941-8-0603-2

<http://www.gtu.ge>

ყველა უფლება დაცულია. ამ წიგნის არც ერთი ნაწილი (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) არანაირი ფორმით და საშუალებით (იქნება ეს ელექტრონული თუ მექანიკური) არ შეიძლება გამოყენებულ იქნეს გამომცემლის წერილობითი ნებართვის გარეშე.

საავტორო უფლებების დარღვევა ისჯება კანონით.

CD-5089

შინაარსი

შესავალი	7
Python ენის შესაძლებლობები	8
I თავი. Python ენის საფუძვლები	9
1.1.IDLE- დამუშავების გარემო	9
1.2. ენის სინტაქსი.....	11
1.3. შეტანა, გამოტანა, კომენტარი	12
1.4. რიცხვები	13
1.5. რიცხვების გარდაქმნის ფუნქციები	16
1.6. სტრიქონები.....	20
1.7. Unicode სტრიქონები.....	26
II თავი. მმართველი ოპერატორები	29
2.1. if-elif-else ინსტრუქცია	29
2.2. While ციკლი	33
2.3. for ციკლი	36
2.4. continue ოპერატორი.....	41
2.5. break ოპერატორი	41
2.6. ჩადგმული ციკლები	43
III თავი. ძირითადი მონაცემთა ტიპები	45
3.1. რიცხვები: მთელი, ნამდვილი, კომპლექსური	45
3.2. ათვლის სისტემები	48
3.3. ნამდვილი რიცხვები - float.....	49
3.4. კომპლექსური რიცხვები - complex	52
3.5. სტრიქონები.....	53
3.5.1. საბაზო ოპერაციები სტრიქონებზე	56
3.5.2. ფუნქციები და მეთოდები სტრიქონებთან სამუშაოდ	58
3.5.3. სტრიქონის ფორმატირება	62
3.6. სიები	75

3.6.1. მეთოდები სიებისთვის	79
3.6.2. ინდექსები და ჭრები	89
3.6.3. სიების კონსტრუირების დამატებითი საშუალებები	92
3.7. კორტეჟი	94
3.8. ლექსიკონები	100
3.8.1. სიის გარდაქმნა ლექსიკონად	101
3.8.2. ელემენტის მიღება, შეცვლა	102
3.8.3. ელემენტის წაშლა	104
3.8.4. ლექსიკონის კოპირება და გაერთიანება	106
3.8.5. ლექსიკონის გადარჩევა	107
3.8.6. კომპლექსური ლექსიკონები	108
3.8.7. მეთოდები ლექსიკონებთან სამუშაოდ	111
IV თავი. სიმრავლე	113
4.1. სიმრავლის განსაზღვრა	113
4.2. ელემენტების დამატება	114
4.3. ელემენტების წაშლა	114
4.4. სიმრავლის გადარჩევა	115
4.5. ოპერაციები სიმრავლეებზე	115
4.6. სიმრავლეთა შორის დამოკიდებულება	117
4.7. ფუნქციები და მეთოდები სიმრავლეებთან სამუშაოდ	118
4.8. frozen set ტიპი	119
V თავი. ფუნქციები	126
5.1. ფუნქციის განსაზღვრა	126
5.2. ფუნქციები: zip(), map(), lambda(), filter(), reduce()	131
VI თავი. მასივები	141
6.1. მასივის შექმნა	141
6.2. მასივების დაბეჭდვა	145
6.3. მასივებზე საბაზო ოპერაციები	146

6.4. ინდექსები, ჭრები, იტერაციები.....	149
6.5. მასივის ფორმის შეცვლა	153
6.6. მასივების გაერთიანება	155
6.7. მასივის გახლეჩვა	156
6.8. ასლები და წარმოდგენები	157
VII თავი. გამონაკლისების დამუშავება	160
VIII თავი. ფაილები.....	166
8.1. ფაილის გახსნა და დახურვა	166
8.2. ტექსტური ფაილები	169
8.3. ფაილის წაკითხვა.....	171
8.4. CSV ფაილები	174
8.5. ბინარული ფაილები	179
8.5.1. shelve მოდული.....	181
8.6. OSმოდული და მუშაობა ფაილურ სისტემასთან	186
IX თავი. მოდულებთან მუშაობა	189
9.1. მოდულის შექმნა, ჩართვა import და from ინსტრუქციებით.....	189
9.1.1. მოდულის ჩართვა სტანდარტული ბიბლიოთეკიდან	189
9.1.2. ფსევდონიმების გამოყენება	191
9.1.3. from ინსტრუქცია	191
9.1.4. საკუთარი მოდულის შექმნა	193
9.2. ძირითადი ჩაშენებული მოდულები	194
9.2.1. random მოდული	194
9.2.2. math მოდული.....	197
9.2.3. locale მოდული.....	199
9.2.4. decimal მოდული	203
9.2.5. quantize() მეთოდი	204
X თავი. ობიექტზე ორიენტირებული დაპროგრამება.....	208
10.1. კლასები და ობიექტები	208

10.2. კონსტრუქტორი.....	212
10.3. დესტრუქტორი.....	214
10.4. კლასის განსაზღვრა მოდულში და მოდულის ჩართვა პროგრამაში.....	216
10.5. ინკაფსულაცია	217
10.6. მემკვიდრეობითობა.....	223
10.7. მრავლობითი მემკვიდრეობითობა.....	228
10.8. mro() მეთოდი	230
10.9. isinstance და subclass ფუნქციები.....	231
10.10. პოლიმორფიზმი, აბსტრაქტული მეთოდი	233
10.11. ოპერატორების გადატვირთვა.....	236
XI თავი. თარიღთან და დროსთან მუშაობა	244
11.1. datetime მოდული.....	244
11.2. ოპერაციები თარიღებზე	248
11.2.1. თარიღისა და დროის ფორმატირება	248
11.2.2. თარიღსა და დროზე არითმეტიკული ოპერაციები.....	250
11.2.3. timedelta თვისება.....	252
11.2.4. თარიღების შედარება.....	253
XII თავი. გრაფიკული ინტერფეისის შექმნა	254
12.1. დანართის ფანჯრის შექმნა.....	254
12.2. ღილაკები	256
12.3. ელემენტების თვისებების შეცვლა.....	261
12.4. ელემენტის პოზიციონირება	265
12.5. Label - ტექსტური ჭდე	272
12.6. Entry - შეტანის ველი.....	274
12.7. Checkbutton.....	280
12.8. Radiobutton.....	285
12.9. Listbox	290
12.10. მენიუს შექმნა	296
ლიტერატურა.....	302

შესავალი

Python დაპროგრამების ენა შეიქმნა დაახლოებით 1991 წელს ჰოლანდიელის - გვიდო ვან როსუმის მიერ. დაპროგრამების ენის სახელი შერჩეულია ტელესერიალის დასახელების მიხედვით. მას შემდეგ, რაც როსუმმა დაამუშავა ენა, მან დადო ინტერნეტში, სადაც მის გასაუმჯობესებლად შემოუერთდა პროგრამისტების მთელი საზოგადოება.

Python აქტიურად სრულყოფილი ხდება თანამედროვე პერიოდში. ხშირად გამოდის მისი ახალი ვერსიები. ოფიციალური საიტია <http://python.org>.

Python დაპროგრამების ინტერპრეტირებადი ენაა: საწყისი კოდი შესრულების პროცესში ნაწილებად გარდაიქმნება მანქანურში სპეციალური პროგრამით - ინტერპრეტატორით.

Python ხასიათდება ცხადი სინტაქსით. კოდის წაკითხვა მოცემულ ენაზე საკმაოდ მარტივია, რამდენადაც მასში ნაკლებია დამხმარე ელემენტები. კოდი ოპტიმალურია. სტანდარტული ბიბლიოთეკა მოიცავს სასარგებლო ფუნქციების დიდ მოცულობას.

Python სრულყოფილი, შეიძლება ითქვას დაპროგრამების უნივერსალური ენაა. იგი მხარს უჭერს ობიექტზე ორიენტირებულ პროგრამირებას, უფრო ზუსტად, იგი თავიდანვე დამუშავდა, როგორც ობიექტზე ორიენტირებული ენა.

Python ენის შესაძლებლობები

რა შესაძლებლობები აქვს Python დაპროგრამების ენას:

- მუშაობა xml/html ფაილებთან;
- მუშაობა http მოთხოვნებთან;
- გრაფიკული ინტერფეისის (GUI) შექმნა;
- ვებ-სცენარების შექმნა;
- FTP-თან მუშაობა;
- გამოსახულებებთან, აუდიო და ვიდეო ფაილებთან მუშაობა;
- რობოტოტექნიკის შექმნა;
- მათემატიკური და სამეცნიერო გამოთვლების დაპროგრამება;
- და სხვა მრავალი.

Python მხარს უჭერს დაპროგრამების რამდენიმე პარადიგმას, მათ შორის შეიძლება დასახელდეს: სტრუქტურული, ობიექტზე ორიენტირებული, ფუნქციონალური, ასპექტურ-ორიენტირებული.

Python აქტიურად განვითარებადი დაპროგრამების ენაა, რის გამოც შეიძლება გამოყენებულ იქნას მსხვილ პროექტებში. მაგალითად, იგი ინტენსიურად გამოიყენება IT-გიგანტების მიერ, როგორცაა Google, Yandex. სიმარტივე და უნივერსალობა python-ს წარმოაჩენს დაპროგრამების ერთ-ერთ საუკეთესო ენად.

I თავი

Python ენის საფუძვლები

1.1. IDLE- დამუშავების გარემო

Python სისტემის ინსტალაციის შემდეგ გავხსნათ IDLE (Integrated Development and Learning Environment). თავდაპირველად გაიხნება ინტერაქტიული რეჟიმი, რის შემდეგაც შეიძლება პირველი პროგრამის დაწერა. ტრადიციულად ეს იყოს "hello world".

აღნიშნული პროგრამის დასაწერად საკმარისია ერთი სტრიქონი:

```
print("Hello world!")
```

შეიტანეთ კოდი IDLE ფანჯარაში და დააჭირეთ Enter ღილაკს. შედეგი გამოვა ფანჯარაში:



```
>>> print("Hello world!")  
Hello world!  
>>>
```

Ln: 298 Col: 4

შეიტანეთ შემდეგი სტრიქონები და მიიღეთ შედეგები:

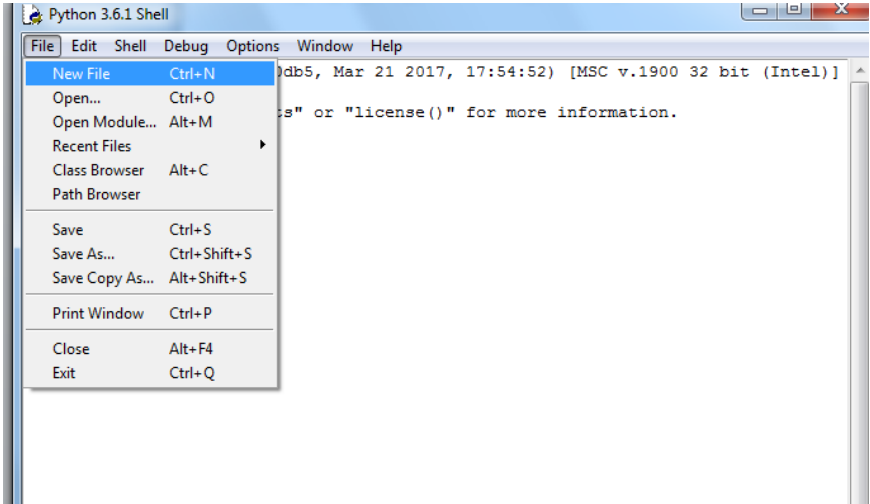
```
print(3 + 4)
```

```
print(3 * 5)
```

```
print(3 ** 2)
```

ინტერაქტიული რეჟიმი არ არის ძირითადი. ძირითადად თქვენ პროგრამულ კოდს შეინახავთ ფაილში

და შემდეგ გაუშვებთ მას. ახალი ფაილის გამოსატანად ინტერაქტულ რეჟიმში აირჩიეთ File → New File ბრძანება და დააჭირეთ Ctrl + N კომბინაციას.

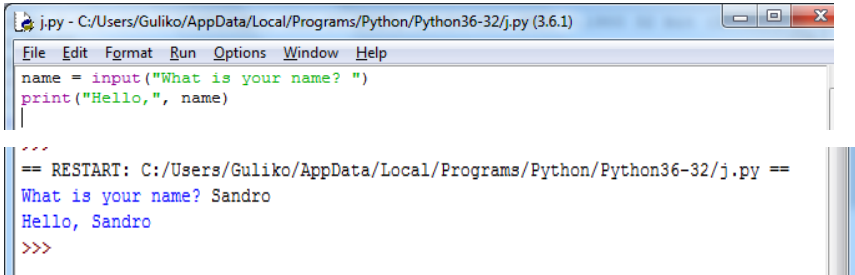


გახსნილ ფანჯარაში შეიტანეთ შემდეგი კოდი:

```
name = input("What is your name? ")  
print("Hello,", name)
```

პირველი სტრიქონი დაბეჭდავს კითხვას: რა გეკვიათ თქვენ? და ელოდება, სანამ თქვენ არ აკრფთ რაიმე ტექსტს და დააჭერთ Enter ღილაკზე. შეტანილ მნიშვნელობას შეინახავს name ცვლადში. მეორე სტრიქონში გამოყენებულია print ფუნქცია ტექსტის ეკრანზე გამოსატანად. დააჭირეთ F5 ღილაკს ან აირჩიეთ Run → Run Module ბრძანება. პროგრამის გაშვებამდე სისტემა შემოგთავაზებთ ფაილის შენახვას. შეინახეთ

ფაილი სასურველ ადგილას და ამის შემდეგ პროგრამა გაეშვება.



```
j.py - C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/j.py (3.6.1)
File Edit Format Run Options Window Help
name = input("What is your name? ")
print("Hello, ", name)

=== RESTART: C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/j.py ===
What is your name? Sandro
Hello, Sandro
>>>
```

1.2. ენის სინტაქსი

- სტრიქონის დასასრული არის ინსტრუქციის დასასრული. წერტილმძიმე არ მოითხოვება;
- ჩაშენებული ინსტრუქციები ერთიანდებიან ბლოკებში შეწევების სიდიდის მიხედვით. შეწევა შეიძლება იყოს ნებისმიერი, მთავარია, რომ ერთი ჩაშენებული ბლოკის ფარგლებში შეწევა იყოს ერთნაირი. კოდის წაკითხვადობას უნდა მიექცეს ყურადღება;
- ძირითადი ინსტრუქციის შემდეგ იწერება ორწერტილი და მის კვალდაკვალ განთავსდება კოდის ჩაშენებული ბლოკი, შეწეული ძირითადი ინსტრუქციის მიმართ;
- ზოგჯერ საჭიროა დაიწეროს რამდენიმე ინსტრუქცია ერთ სტრიქონში, გამოყოფილი წერტილ-მძიმეებით:
a = 1; b = 2; print(a, b)
თუმცა ამის გაკეთება ხშირად არასასურველია.

- დასაშვებია ერთი ინსტრუქციის რამდენიმე სტრიქონად დაწერა. საკმარისია მისი ჩასმა წყვილ მრგვალ, კვადრატულ ან ფიგურულ ფრჩხილებში:

```
if (a == 1 and b == 2 and
    c == 3 and d == 4): # აქ ორიწერტილი უნდა
    print('spam' * 3)
```

- რთული ინსტრუქციის ტანი შეიძლება განთავსდეს ძირითადი ინსტრუქციის სტრიქონშივე, მაგალითად:
if x > y: print(x)

1.3. შეტანა, გამოტანა, კომენტარი

შეტანა და გამოტანა განსხვავდება მიწვევის ('>>>' ან '...') არსებობით ან არარსებობით. მიწვევის გამოჩენის შემდეგ თქვენ უნდა აკრიფოთ ტექსტი. მაგალითში სტრიქონები არ იწყება მიწვევით, გამოიცემა თვით ინტერპრეტატორის მიერ. მიაქციეთ ყურადღება, რომ მაგალითში მხოლოდ მეორადი მიწვევის არსებობა აღნიშნავს, რომ თქვენ უნდა შეიტანოთ ცარიელი სტრიქონი - ამ სახით ინტერაქტიულ რეჟიმში აღინიშნება მრავალსტრიქონიანი რეჟიმის დასასრული.

კომენტარი Python-ში იწყება # სიმბოლოთი და გრძელდება სტრიქონის ბოლომდე. კომენტარი შეიძლება იწყებოდეს სტრიქონის დასაწყისში ან კოდის შემდეგ, მაგრამ არა სტრიქონული გამოსახულების შიგნით.

სტრიქონულ გამოსახულებაში # სიმბოლო წარმოადგენს მხოლოდ # სიმბოლოს.

განვიხილოთ მაგალითები:

#ეს პირველი კომენტარია

abc=3 #ეს მეორე კომენტარია

#... ეს მესამე კომენტარია

STRING="# ეს არ არის კომენტარი"

1.4. რიცხვები

ინტერპრეტატორი მუშაობს, როგორც უბრალო კალკულატორი: თქვენ შეგიძლიათ აკრიფოთ გამოსახულება და იგი გამოიტანს შედეგს. გამოსახულების სინტაქსი მარტივია: ოპერატორები +, -, * და / მუშაობენ, როგორც დაპროგრამების სხვა ენებში. დაჯგუფების მიზნით შეიძლება გამოვიყენოთ ფრჩხილები.

მაგალითები:

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

```
4
```

```
>>> # ეს კომენტარია
```

```
... 2+2
```

```
4
```

```
>>> 2+2 # კომენტარი კოდის სტრიქონშივცა
```

```
4
```

```
>>> (50-5*6)/4
```

5

>>> # მთელი რიცხვა გაყოფისას შედეგი მრგვალდება

... # ნაკლებობით

... 7/3

2

>>> 7/-3

-3

ორი რიცხვის შეკრება:

```
print(6 + 2) # 8
```

ორი რიცხვის გამოკლება:

```
print(6 - 2) # 4
```

ორი რიცხვის გამრავლება:

```
print(6 * 2) # 12
```

ორი რიცხვის გაყოფა:

```
print(6 / 2) # 3.0
```

ორი რიცხვის მთელი რიცხვა გაყოფა:

```
print(7 // 2) # 3
```

ახარისხება:

```
print(6 ** 2) # 36
```

ნაშთის მიღება გაყოფის შედეგად:

```
print(7 % 2) # 1
```

გამოსახულების ამოხსნა:

```
number = 3 + 4 * 5 ** 2 + 7
```

```
print(number) # 110
```

წინა მაგალითში მოქმედებების მიმდევრობის გადასაწყობად გამოყენებულია ფრჩხილები:

```
number = (3 + 4) * (5 ** 2 + 7)
print(number) # 224
```

C ჯგუფის ენების მსგავსად ტოლობის ნიშანი (=) გამოიყენება ცვლადისთვის მნიშვნელობის მისანიჭებლად. მინიჭებული მნიშვნელობა ამ დროს არ გამოიტანება:

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

მნიშვნელობა შეიძლება მინიჭებულ იქნას ერთდროულად რამდენიმე ცვლადზე:

```
>>> x = y = z = 0 # x, y და z ცვლადებს
მიენიჭებათ 0
>>> x
0
>>> y
0
>>> z
0
```

ენაში გამოიყენება მინიჭების სპეციალური ოპერაციები:

```
+= შეკრების შედეგის მინიჭება;
-= გამოკლების შედეგის მინიჭება;
```

*= გამრავლების შედეგის მინიჭება;
/= გაყოფის შედეგის მინიჭება;
//= მთელრიცხვა გაყოფის შედეგის მინიჭება;
**=რიცხვის ხარისხის მინიჭება;
%= გაყოფისგან მიღებული ნაშთის მინიჭება.

მაგალითები:

```
number = 10
number += 5
print(number) # 15
number -= 3
print(number) # 12
number *= 4
print(number) # 48
```

1.5. რიცხვების გარდაქმნის ფუნქციები

Python ენაში გამოიყენება ჩაშენებული ფუნქციები, რომლებიც მუშაობენ რიცხვებთან. კერძოდ, ფუნქციები: int() და float() საშუალებას იძლევიან გადაიყვანონ მნიშვნელობები მთელ ან ნამდვილ ტიპებად, შესაბამისად.

მაგალითად, მოცემულია შემდეგი კოდი:

```
first_number = "2"
second_number = 3
third_number = first_number + second_number
```

კოდის შესრულებისას გენერირდება შეცდომა, რადგან პირველი რიცხვი წარმოადგენს სტრიქონს.

შეცდომის აღმოფხვრისთვის საჭიროა სტრიქონის გარდაქმნა რიცხვად int() ფუნქციის დახმარებით:

```
first_number = "2"  
second_number = 3  
third_number = int(first_number) + second_number  
print(third_number) # 5
```

ანალოგიურად მუშაობს float() ფუნქცია, რომელიც გარდაქმნის მცურავწერტილიან რიცხვად. წილად რიცხვებთან მუშაობის დროს გასათვალისწინებელია, რომ ოპერაციის შედეგი შეიძლება არ იყოს ზუსტი.

მაგალითად:

```
first_number = 2.0001  
second_number = 5  
third_number = first_number / second_number  
print(third_number) # 0.400020000000000004
```

შედეგის დასამრგვალებლად შეიძლება გამოვიყენოთ round() ფუნქცია:

```
first_number = 2.0001  
second_number = 0.1  
third_number = first_number + second_number  
print(round(third_number, 4)) # 2.1001
```

ფუნქციის პირველი პარამეტრი არის დასამრგვალებელი რიცხვი, ხოლო მეორე - რამდენნიშნა უნდა იყოს მიღებული რიცხვის წილადი ნაწილი.

მცურავწერტილიან რიცხვებზე არის სრული მხარდაჭერა. ოპერატორები, რომლებიც შეიცავენ

შერეული ტიპების ოპერანდებს, გარდაქმნიან მთელ ოპერანდს მცურავწერტილიან ტიპად:

```
>>> 4 * 2.5 / 3.3
3.0303030303030303
>>> 7.0 / 2
3.5
```

ასევე, მხარდაჭერა აქვთ კომპლექსურ რიცხვებს. წარმოსახვითი ნაწილი ჩაიწერება 'j' ან 'J' სუფიქსით. კომპლექსური რიცხვები ჩაიწერება, როგორც '(real+imagj)' ან შეიძლება შეიქმნას ფუნქციით: 'complex(real, imag)'.

```
>>> 1j * 1j
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

კომპლექსური რიცხვები წარმოდგინდება მცურავწერტილიანი ორი რიცხვით - ნამდვილი და წარმოსახვითი ნაწილით. z კომპლექსური რიცხვიდან ამ ნაწილების ამოსაღებად გამოიყენეთ z.real და z.imag.

```
>>> a=1.5+0.5j
>>> a.real
```

```
1.5
```

```
>>> a.imag
```

```
0.5
```

მთელი და მცურავწერტილიანი რიცხვების გარდაქმნის ფუნქციები (`int()`, `long()` და `float()`) არ მუშაობენ კომპლექსური რიცხვებისთვის - ასეთი გარდაქმნა არაერთმნიშვნელოვანია. გამოიყენეთ `abs(z)` აბსოლუტური მნიშვნელობის მისაღებად და `z.real` წილადი ნაწილის მისაღებად.

```
>>> a=1.5+0.5j
```

```
>>> float(a)
```

```
Traceback (innermost last):
```

```
File "<stdin>", line 1, in ?
```

```
TypeError: can't convert complex to float; use e.g.
```

```
abs(z)
```

```
>>> a.real
```

```
1.5
```

```
>>> abs(a)
```

```
1.5811388300841898
```

ინტერაქტულ რეჟიმში ბოლო გამოტანილი მნიშვნელობა შეინახება ცვლადში, რაც საშუალებას იძლევა Python გამოყენებულ იქნას სამაგიდო კალკულატორის სახით, მაგალითად:

```
>>> tax = 17.5 / 100
```

```
>>> price = 3.50
```

```
>>> price * tax
```

```
0.61249999999999993
```

```
>>> price + _
```

```
4.1124999999999998
```

```
>>> print (round(_, 2))
```

```
4.11
```

მომხმარებელმა უნდა მიმართოს მას, როგორც ცვლადს, რომელიც მისაწვდომია მხოლოდ წასაკითხად. მას არ მიანიჭოთ ცხადად მნიშვნელობა - თქვენ შექმნით დამოუკიდებელ ლოკალურ ცვლადს იმავე სახელით, ჩაშენებულ ცვლადს გახდით მიუწვდომელს.

1.6. სტრიქონები

რიცხვების გარდა Python მუშაობს სტრიქონებთან ანუ string ტიპებთან, რომლებიც შეიძლება ჩაწერილი იყოს სხვადასხვა გზებით. ისინი შეიძლება ჩასმულ იქნან ერთ ან ორმაგ ბრჭყალებში:

```
>>> 'spam eggs'
```

```
'spam eggs'
```

```
>>> 'doesn\'t'
```

```
"doesn't"
```

```
>>> "doesn't"
```

```
"doesn't"
```

```
>>> "Yes," he said.'
```

```
"Yes," he said.'
```

```
>>> "\"Yes,\" he said."
```

```
"Yes," he said.'
```

```
>>> 'Isn\'t," she said.'
```

```
'Isn\'t," she said.'
```

გრძელი სტრიქონული გამოსახულება შეიძლება დაიყოს სხვადასხვა ხერხებით რამდენიმე სტრიქონად. ახალი სტრიქონის სიმბოლო შეიძლება შეიძლება დამალულ იქნას შებრუნებული სლემის (/) დახმარებით. მაგალითად:

```
hello = "ეს გრძელი სტრიქონული გამოსახულებაა,  
რომელიც შეიცავს \n\  
რამდენიმე სტრიქონს\n\  
print (hello)
```

შედეგი იქნება:

```
ეს გრძელი სტრიქონული გამოსახულებაა, რომელიც  
შეიცავს  
რამდენიმე სტრიქონს
```

სხვანაირად, ტექსტი შეიძლება ჩაისვას სამმაგ ბრჭყალებში: " " " ან ' ' '. სტრიქონების დაბოლოებები ამ შემთხვევაში არ უნდა დაიმალოს, მაგრამ ისინი ჩაერთვებიან ტექსტში:

```
print ("""  
Usage: thingy [OPTIONS]  
-h                Display this usage message  
-H hostname       Hostname to connect to  
""")
```

გამოიტანს შედეგს:

```
Usage: thingy [OPTIONS]
```

-h Display this usage message
-H hostname Hostname to connect to

არსებობს სტრიქონის შეტანის დაუმუშავებელი რეჟიმი, რომელიც შეიძლება განხორციელდეს 'r' ან 'R' სიმბოლოებით ბრჭყალების წინ. ამ შემთხვევაში შებრუნებული სლეში შეიძლება გამოყენებულ იქნას ერთმაგი ან ორმაგი ბრჭყალების დამალვისთვის, თუ მათ წინ უსწრებს შებრუნებული სლეშის კენტი რაოდენობა. თუმცა თვით შებრუნებული სლეში დარჩება სტრიქონის ნაწილად. იქამდე ამ რეჟიმში სტრიქონი არ დამთავრდება შებრუნებული სლეშის კენტი რაოდენობით. დაუმუშავებელი რეჟიმი სასარგებლოა იმ შემთხვევებში, როდესაც საჭიროა შებრუნებული სლეშის მნიშვნელოვანი რაოდენობის შეტანა, მაგალითად რეგულარულ გამოსახულებებში.

სტრიქონები შეიძლება გაერთიანდეს + ოპერატორის გამოყენებით და გამრავლდეს * ოპერატორით:

```
>>> word = 'Help' + 'A'  
>>> word  
'HelpA'  
>>> '<' + word*5 + '>'  
'<HelpAHelpAHelpAHelpAHelpA>'
```

ერთმანეთის გვერდით ჩაწერილი ორი სტრიქონიც გაერთიანდება. მოყვანილ მაგალითში პირველი სტრიქონი შეიძლება ჩაიწეროს შემდეგნაირად:

```
'word = 'Help' 'A'.
```

თუმცა ეს მეთოდი მუშაობს მხოლოდ ერთმანეთის გვერდით დაწერილი სტრიქონებისთვის და არა ნებიქმიერი სტრიქონული გამოსახულებისთვის.

```
>>> 'str' 'ing' # სწორია
'string'
>>> 'str'.strip() + 'ing' # სწორია
'string'
>>> 'str'.strip() 'ing' # შეცდომაა
File "<stdin>", line 1
'str'.strip() 'ing'
^
```

SyntaxError: invalid syntax

სტრიქონის ნებისმიერი სიმბოლოს ამოღება შეიძლება მისი ინდექსის მეშვეობით. C ენის მსგავსად პირველი სიმბოლოს ინდექსია - 0. სიმბოლოსთვის ცალკე ტიპი არ არის. სიმბოლო არის ერთეული სიგრძის სტრიქონი. ქვესტრიქონი შეიძლება განისაზღვროს ჭრის მეშვეობით - ორი ინდექსით, რომლებიც გამოყოფილია ორწერტილით.

```
>>> word[4]
'A'
>>> word[0:2]
'He'
>>> word[2:4]
'lp'
```

Python ენაში სტრიქონის შეცვლა შეუძლებელია. სიმბოლოს შეცვლის მცდელობა განსაზღვრულ პოზიციაში ან ქვესტრიქონში გამოიწვევს შეცდომას:

```
>>> word[0] = 'x'
```

Traceback (innermost last):

File "<stdin>", line 1, in ?

TypeError: object doesn't support item assignment

```
>>> word[: -1] = 'Splat'
```

Traceback (innermost last):

File "<stdin>", line 1, in ?

TypeError: object doesn't support slice assignment

ჭრის ინდექსებს აქვთ სასარგებლო მნიშვნელობა დუმილით: გამოტოვებული პირველი ინდექსი ითვლება 0-ის ტოლად, გამოტოვებული მეორე ინდექსი იძლევა იგივე შედეგს, როგორც თითქოს მითითებული იყოს სტრიქონის ბოლო სიმბოლოს ინდექსი.

```
>>> word[:2] # პირველი-ორი სიმბოლო
```

```
'He'
```

```
>>> word[2: ] # მთელი სტრიქონი გარდა პირველი ორი სიმბოლოსა
```

```
'lpA'
```

ჭრის ოპერაციის სასარგებლო ვარიანტია: $s[:i] + s[i:]$, რომელიც ტოლია s -ის.

```
>>> word[:2] + word[2:]
```

```
'HelpA'
```

```
>>> word[:3] + word[3:]
```


'HelpA'

თუ ინდექსი აჭარბებს სტრიქონის სიგრძეს, იგი დამუშავდება თითქოს სტრიქონის სიგრძის ტოლი იყოს. თუ ზედა საზღვარი ნაკლებია ქვედაზე, დაბრუნდება ცარიელი სტრიქონი.

```
>>> word[1:100]
```

```
'elpA'
```

```
>>> word[10:]
```

```
''
```

```
>>> word[2:1]
```

```
''
```

ინდექსებს შეიძლება ჰქონდეთ უარყოფითი მნიშვნელობები ბოლოდან გადათვლისთვის:

```
>>> word[-1] # ბოლო სიმბოლო
```

```
'A'
```

```
>>> word[-2] # ბოლოსწინა სიმბოლო
```

```
'p'
```

```
>>> word[-2:] # ბოლო ორი სიმბოლო
```

```
'pA'
```

```
>>> word[:-2] # ბოლო ორი სიმბოლოს გარდა
```

```
'Hel'
```

-0 იგივეა, რაც 0, ანუ არ გადაითვლება ბოლოდან.

```
>>> word[-0] # (რამდენადაც -0 იგივეა, რაც 0)
```

```
'H'
```

უარყოფითი ინდექსები ჭრებში, რომლებიც გადიან საზღვრებიდან, მუშავდება თითქოს ისინი ნულის ტოლი

იყოს. მხოლოდ არ შეეცადოთ მათი გამოყენება მარტივი ინდექსებისთვის (ერთელემენტისანი).

```
>>> word[-100:]
```

```
'HelpA'
```

```
>>> word[-10] # შეცდომაა
```

```
Traceback (innermost last):
```

```
File "<stdin>", line 1
```

```
IndexError: string index out of range
```

ჩაშენებული ფუნქცია len() აბრუნებს სტრიქონის სიგრძეს:

```
>>> s = 'supercalifragilisticexpialidocious'
```

```
>>> len(s)
```

```
34
```

1.7. Unicode სტრიქონები

1.6. ვერსიიდან დაწყებული Python ენაში მისაწვდომია მონაცემთა ახალი ტიპი ტექსტის შესანახად - Unicode სტრიქონის. იგი შეიძლება გამოყენებულ იქნას ტექსტთან სამუშაოდ, რომელიც ერთდროულად შეიცავს ასოებს და რამდენიმე ენის სიმბოლოებს. Unicode სტრიქონები სრულად ინტეგრირდება ჩვეულებრივ სტრიქონებთან, ავტომატურად ახდენენ გარდაქმნებს საჭიროების მიხედვით.

Unicode აქვს მნიშვნელოვანი უპირატესობა - წარადგენს ყველა სიმბოლოს გამოყენების შესაძლებლობას, რომლებიც გამოყენებულია

თანამედროვე და ძველ ტექსტებში. ადრე შესაძლებელი იყო 256 სიმბოლოს გამოყენება განსაზღვრული კოდური გვერდიდან, რაც სირთულეს ქმნიდა პროგრამული უზრუნველყოფის ინტერნაციონალიზაციისას. Unicode წყვეტს ამ პრობლემას ყველა სიმბოლოსთვის ერთი კოდური გვერდის განსაზღვრით.

Unicode სტრიქონები მარტივად შეიქმნება:

```
>>> u'Hello World !'
```

```
u'Hello World !'
```

პატარა u ასო ბრჭყალის წინ უჩვენებს, რომ უნდა შეიქმნას Unicode სტრიქონები. თუ გსურთ სტრიქონში ჩართოთ სპეციალური სიმბოლოები, გამოიყენეთ მმართველი მიმდევრობები:

```
>>> u'Hello\u0020World !'
```

```
u'Hello World !'
```

მმართველი მიმდევრობა \u0020 მიუთითებს, რომ აუცილებელია ჩაისვას Unicode სიმბოლო რიგითი ნომრით ათვლის თექვსმეტობით სისტემაში 0x0020 (ხარვეზი).

იმის გამო, რომ პირველი 256 Unicode სიმბოლო იგივეა, რაც სტანდარტულ Latin-1 კოდირებაში, უმეტეს ენებზე ტექსტის შეტანა მარტივდება.

როგორც ჩვეულებრივი სტრიქონებისთვის, ასევე Unicode სტრიქონებისთვის არსებობს დაუმუშავებელი რეჟიმი, რომელიც მოცემულია 'r' ან 'R' სიმბოლოებით ბრჭყალების წინ. მმართველად ითვლება მხოლოდ

მიმდევრობები, რომლებიც გამოიყენება Unicode სიმბოლოების აღსანიშნად და მხოლოდ თუ გამოიყენება შებრუნებული სლემის კენტი რაოდენობა 'u' ასოს წინ:

```
>>> ur'Hello\u0020World !'
```

```
u'Hello World !'
```

```
>>> ur'Hello\\u0020World !'
```

```
u'Hello\\\\u0020World !'
```

ზემოთ აღწერილი მეთოდის გარდა Python გვთავაზობს Unicode სტრიქონის შექმნის შესაძლებლობას ცნობილ კოდირებაში სტრიქონის საფუძველზე. ჩაშენებული ფუნქცია unicode() შეიძლება მუშაობდეს Latin-1, ASCII, UTF-8, UTF-16 და სხვა მრავალთან. დუმილის პრინციპით Python იყენებს ASCII კოდირებას, მაგალითად print ინსტრუქციით ეკრანზე გამოსატანად და ფაილში ჩაწერისას. თუ თქვენ გაქვთ მონაცემები განსაზღვრულ კოდირებაში, Unicode სტრიქონის მისაღებად გამოიყენეთ ჩაშენებული ფუნქცია unicode(), მიუთითეთ კოდირება მეორე არგუმენტის სახით.

თუ Unicode სტრიქონი შეიცავს სიმბოლოებს 127-ზე მეტი კოდით, ASCII -ში გარდაქმნა შეუძლებელია:

```
>>> str(s)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in ?
```

```
UnicodeError: ASCII encoding error: ordinal not in
```

```
range(128)
```

II თავი

მმართველი ოპერატორები

2.1. if-elif-else ინსტრუქცია

განშტოების ოპერატორი Python-ში ამორჩევის ძირითადი ინსტრუმენტია. იგი ამოირჩევს რომელი მოქმედება უნდა შესრულდეს პირობის შემოწმებიდან გამომდინარე.

თავდაპირველად იწერება if ნაწილი პირობის აღმნიშვნელი გამოსახულებით, შემდეგ მას შეიძლება მოყვეს elif ერთი ან რამდენიმე არააუცილებელი ნაწილი და ბოლოს, არააუცილებელი else ნაწილი.

ზოგადი სახე შემდეგნაირად გამოიყურება:

```
if test1:  
    state1  
elif test2:  
    state2  
else:  
    state3
```

მარტივი მაგალითი ბეჭდავს 'true' -ს რადგან 1 არის ჭეშმარიტი:

```
>>> if 1:  
...   print('true')  
... else:  
...   print('false')  
...
```

true

შემდეგ მაგალითში შედეგი დამოკიდებულია იმაზე, თუ რა შეიტანა მომხმარებელმა:

```
a = int(input())
if a < -5:
    print('Low')
elif -5 <= a <= 5:
    print('Mid')
else:
    print('High')
```

კონსტრუქციამ რამდენიმე elif გამოყენებით შეიძლება ჩაანაცვლოს switch – case კონსტრუქცია დაპროგრამების სხვა ენებში.

ლოგიკური ოპერატორები:

X and Y - ჭეშმარიტია, როდესაც ორივე X და Y მნიშვნელობები არის ჭეშმარიტი;

X or Y - ჭეშმარიტია, როდესაც თუნდაც ერთ-ერთი X და Y მნიშვნელობებიდან არის ჭეშმარიტი;

not X - ჭეშმარიტია, თუ X მცდარია.

განვიხილოთ ინსტრუქცია, სადაც X მნიშვნელობიდან გამომდინარე A -ს მიენიჭება Y ან Z. კონკრეტულად, თუ X ჭეშმარიტია, მაშინ A -ს მიენიჭება Y, ხოლო თუ X მცდარია, A -ს მიენიჭება Z.

```
if X:
```

```
    A = Y
```

else:

A = Z

იგივე ინსტრუქცია, შეიძლება დავწეროთ უფრო მოკლედ:

A = Y if X else Z

განვიხილოთ კიდევ ერთი მაგალითი:

```
>>> x = int(input("შეიტანეთ რიცხვი: "))
```

```
>>> if x < 0:
```

```
... x = 0
```

```
... print ('უარყოფითს ვცვლით 0-ით')
```

```
... elif x == 0:
```

```
... print ('ნული')
```

```
... elif x == 1:
```

```
... print ('ერთი')
```

```
... else:
```

```
... print ('მეტი')
```

შედარების ოპერაციის მაგალითები:

```
a = 5
```

```
b = 6
```

```
result = 5 == 6 # ვინახავთ ოპერაციის შედეგს
```

ცვლადში

```
print(result) # False - 5 არ უდრის 6
```

```
print(a != b) # True
```

```
print(a > b) # False - 5 ნაკლებია 6
```

```
print(a < b) # True
```

```
bool1 = True
```

```
bool2 = False
```

```
print(bool1 == bool2) # False - bool1 არ უდრის bool2
```

ლოგიკური ოპერაციის მაგალითები:

and (ლოგიკური გამრავლება)

```
ა) age = 22
```

```
weight = 58
```

```
result = age > 21 and weight == 58
```

```
print(result) # True
```

```
ბ) age = 22
```

```
weight = 58
```

```
isMarried = False
```

```
result = age > 21 and weight == 58 and isMarried
```

```
print(result) # False, რადგან isMarried = False
```

or (ლოგიკური შეკრება)

```
age = 22
```

```
isMarried = False
```

```
result = age > 21 or isMarried
```

```
print(result) # True, რადგან age > 21 არის True
```

not (ლოგიკური უარყოფა):

```
age = 22
```

```
isMarried = False
```

```
print(not age > 21) # False
```

```
print(not isMarried) # True
```

მოქმედებების მიმდევრობის შეცვლისთვის
შეიძლება გამოყენებულ იქნას მრგვალი ფრჩხილები:


```
age = 22
isMarried = False
weight = 58
result = (weight == 58 or isMarried) and not age > 21 #
False
print(result)
```

2.2. While ციკლი

While ციკლი ერთ-ერთი ყველაზე უნივერსალურია Python-ის ციკლებიდან, ამიტომ საკმაოდ ნელია. იგი ციკლის ტანს შეასრულებს მანამ, სანამ ციკლის პირობა ჭეშმარიტია. ციკლის სინტაქსი:

```
while პირობა_გამოსახულება:
    ინსტრუქციები
```

კოდი ბეჭდავს 5-დან 15-მდე კენტ რიცხვებს:

```
i = 5
>>> while i < 15:
...     print(i)
...     i = i + 2
...
5
7
9
11
13
```

განვიხილოთ ფიბონაჩის რიცხვების გამოთვლის პროგრამული კოდი:

```
>>> # ფიბონაჩის მწკრივი:
... # ორი წინა რიცხვის ჯამი განსაზღვრავს
... # მომდევნოს
... a, b = 0, 1
>>> while b < 10:
...     print (b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

განვიხილოთ მაგალითი ფრაგმენტულად:

– პირველი სტრიქონი შეიცავს მრავლობით მინიჭებას: a და b ცვლადებს ერთდროულად ენიჭებათ მნიშვნელობები, შესაბამისად 0 და 1. ბოლო სტრიქონში იგი სრულდება ხელახლა, რაც უჩვენებს, a და b ცვლადებს მიენიჭებათ ახალი მნიშვნელობები.

– `while` ციკლი სრულდება, მანამ პირობა $b < 10$ არის ჭეშმარიტი. როგორც სხვა ენებში, Python-ში ნებისმიერი არანულოვანი მნიშვნელობა ითვლება ჭეშმარიტად, ხოლო 0 -მცდარად. პირობის სახით შეიძლება იყოს სტრიქონი,

სია ან ნებისმიერი მიმდევრობა. მიმდევრობა არანულოვანი სიგრძით არის ჭეშმარიტი, ცარიელი მიმდევრობა - მცდარი. შედარების სტანდარტული ოპერატორები იწერება C ენის ანალოგიურად: <, >, ==, <=, >= და !=.

– ციკლის ტანი დაწერილია შეწევით, რაც გამოიყენება დაჯგუფებული ინსტრუქციის ჩასაწერად.

print ინსტრუქციას გამოაქვს მასზე გადაცემული მნიშვნელობა. ინტერაქტიულ რეჟიმში print ინსტრუქციას სტრიქონი გამოაქვს ბრჭყალების გარეშე და ელემენტებს შორის ჩასვამს ხარვეზს.

```
>>> i = 256*256
```

```
>>> print (' i ცვლადის მნიშვნელობა ტოლია', i)
```

შედეგი:

```
i ცვლადის მნიშვნელობა ტოლია 65536
```

დამასრულებელი მძიმე საშუალებას იძლევა მნიშვნელობის გამოტანის შემდეგ დაუსვას ხარვეზი ახალ სტრიქონზე გადასვლის ნაცვლად:

```
>>> a, b = 0, 1
```

```
>>> while b < 1000:
```

```
... print (b),
```

```
... a, b = b, a+b
```

```
...
```

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

განვიხილოთ პროგრამის კოდი, რომელიც ითვლის რიცხვის ფაქტორიალს:

```

number = int(input("შეიტანეთ რიცხვი: "))
i = 1
factorial = 1
while i <= number:
    factorial *= i
    i += 1
print("რიცხვის ", number, "ფაქტორიალი ტოლია", factorial)

```

ელეგი:

შეიტანეთ რიცხვი: 5
რიცხვის 5 ფაქტორიალი ტოლია 120

2.3. for ციკლი

for ციკლი შედარებით რთულია, ნაკლებად უნივერსალურია, მაგრამ გაცილებით სწრაფად სრულდება, ვიდრე while ციკლი. ეს ციკლი გაივლის ნებისმიერ იტერირებულ ობიექტზე (სტრიქონზე ან სიაზე) და ცალკეული გავლის დროს ასრულებს ციკლის ტანს.

The screenshot shows a Python IDE window titled 'lk.py - C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/lk.py (3.6.1)'. The code in the editor is:

```

for i in 'hello world':
    print(i * 2, end='')

```

შედეგი:

The screenshot shows the output of the Python IDE. The first line is a restart message: '== RESTART: C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/lk.py =='. The second line shows the output of the for loop: 'hheellllloo wwwoorrllldd'. The prompt '>>>' is visible at the bottom.

for ციკლი გამოიძახება ცალკეული რიცხვისთვის რიცხვთა კოლექციაში, რომელიც შეიძლება შეიქმნას range() ფუნქციის გამოყენებით. ციკლის ფორმალური განსაზღვრა:

```
for int_var in ფუნქცია_range:  
    ინსტრუქციები
```

ციკლის შესრულების პირველი გავლისას ციკლი მიიღებს პირველ რიცხვს კოლექციიდან, მეორე გავლისას - მეორეს და ასე შემდეგ, სანამ არ გადაარჩევს ყველა რიცხვს, რის შემდეგაც ციკლი დაასრულებს მუშაობას.

განვიხილოთ ფაქტორიალის გამოთვლის მაგალითი:

```
number = int(input("შეიტანეთ რიცხვი: "))
```

```
factorial = 1
```

```
for i in range(1, number+1):
```

```
    factorial *= i
```

```
print("ფაქტორიალი", number, "რიცხვის ტოლია",  
factorial)
```

თავდაპირველად შეგვაქვს კონსოლიდან რიცხვი. ციკლში განვსაზღვრავთ i ცვლადს, რომელშიც ინახება რიცხვი კოლექციიდან, შექმნილი range ფუნქციის მიერ. აქ ფუნქცია range იღებს ორ არგუმენტს - კოლექციის საწყისი რიცხვს, აქ არის რიცხვი 1 და რიცხვს, რომელამდეც უნდა დაემატოს რიცხვები - number +1.

დავუშვათ კონსოლიდან შეტანილ იქნა 6, მაშინ range ფუნქციის გამოძახება მიიღებს შემდეგ ფორმას:

```
range(1, 6+1):
```

ეს ფუნქცია შექმნის კოლექციას, რომელიც დაიწყება 1-ით და მიმდევრობით შეივსება მთელი რიცხვებით 7-მდე. ანუ ეს იქნება კოლექცია [1, 2, 3, 4, 5, 6].

ციკლის შესრულებისას აღნიშნული კოლექციიდან მიმდევრობით გადაეცემა რიცხვები i ცვლადში, ხოლო თვით ციკლში მოხდება i ცვლადის გამრავლება factorial ცვლადზე. შედეგად მივიღებთ რიცხვების ფაქტორიალს.

შედეგი:

შეიტანეთ რიცხვი: 6

ფაქტორიალი 6 რიცხვის ტოლია 720

range ფუნქციას აქვს შემდეგი ფორმები:

- range(stop) - აბრუნებს ყველა მთელ რიცხვს 0-დან stop-მდე;
- range(start, stop) - აბრუნებს ყველა მთელ რიცხვს შუალედში: start-დან (ჩათვლით) stop-მდე (არ ჩათვლით). ფაქტორიალის გამოთვლის მაგალითში სწორედ ეს ფორმა იყო გამოყენებული;
- range(start, stop, step) - აბრუნებს მთელ რიცხვებს შუალედში: start-დან (ჩათვლით) stop-მდე (არ ჩათვლით), რომლებიც იზრდებიან step მნიშვნელობით.

განვიხილოთ range ფუნქციის გამოძახების მაგალითები:

```
range(5)      # 0, 1, 2, 3, 4
range(1, 5)   # 1, 2, 3, 4
range(2, 10, 2) # 2, 4, 6, 8
```

```
range(5, 0, -1) # 5, 4, 3, 2, 1
```

მაგალითად, გამოვიტანოთ მიმდევრობით ყველა რიცხვი 0-დან 4-მდე:

```
for i in range(5):  
    print(i, end=" ")
```

თუ მოცემულია სია, რომელიც შედგება მთელი რიგი ელემენტებისგან. თავდაპირველად ციკლი მიმართავს პირველ ელემენტს, შემდეგ მეორეს, მესამეს და ასე შემდეგ. ცალკეულ ელემენტზე სრულდება ერთიდაიგივე მოქმედება ციკლის ტანში. არ არის საჭირო ელემენტების ამოღება ინდექსების მიხედვით და არც იმის განსაზღვრა თუ რომელი ელემენტით მთავრდება სია. ციკლი თვითონ გადაარჩევს და განსაზღვრავს დასასრულს.

განვიხილოთ მაგალითი, სადაც ციკლი ამოარჩევს სიიდან ელემენტებს მიმდევრობით:

```
#რამდენიმე სტრიქონის სიგრძის განსაზღვრა:  
a = ['კატა', 'სპილო', 'კურდღელი']  
for x in a:  
    print(x, len(x))
```

შედეგი:

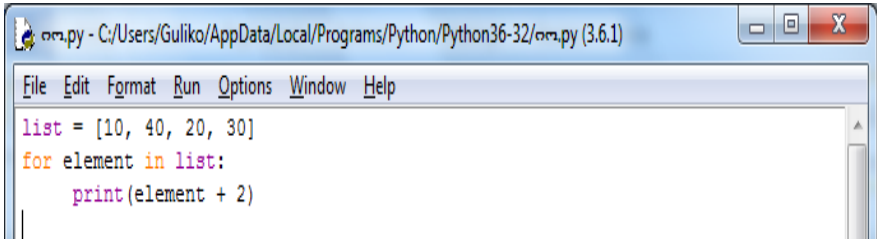
კატა 4

სპილო 5

კურდღელი 8

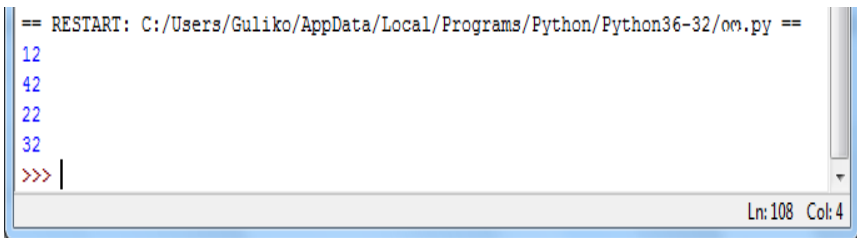
განვიხილოთ მაგალითი, სადაც სია შედგება რიცხვებისგან, შედეგად დაიბეჭდება 2-ით გაზრდილი

რიცხვები. მოცემულ მაგალითში სიაში რიცხვები არ იცვლება:



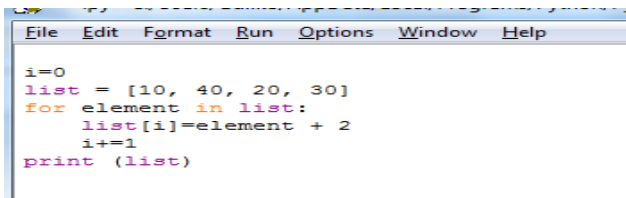
```
o.py - C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/o.py (3.6.1)
File Edit Format Run Options Window Help
list = [10, 40, 20, 30]
for element in list:
    print(element + 2)
```

შედეგი:



```
== RESTART: C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/o.py ==
12
42
22
32
>>> |
Ln: 108 Col: 4
```

ზოგჯერ საჭიროა თვით სიის შეცვლა, მაგალითად შეიცვალოს ცალკეული ელემენტის მნიშვნელობა ან იმ ელემენტების მნიშვნელობები, რომლებიც აკმაყოფილებენ განსაზღვრულ პირობას. ამ შემთხვევაში უმეტეს შემთხვევაში საჭიროა ცვლადის შემოტანა, რომელიც აღნიშნავს ელემენტის ინდექსს:



```
File Edit Format Run Options Window Help
i=0
list = [10, 40, 20, 30]
for element in list:
    list[i]=element + 2
    i+=1
print (list)
```


შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/Lc
>>>
[12, 42, 22, 32]
>>>
```

2.4. continue ოპერატორი

continue ოპერატორი აკეთებს გადასვლას ციკლის შემდეგ იტერაციაზე.

განვიხილოთ მაგალითი, სადაც 'O' ასოს შემთხვევაში ციკლის ტანი არ შესრულდება:

```
File Edit Format Run Options Window
for i in 'hello world':
    if i == 'o':
        continue
    print(i * 2, end='')
```

შედეგი:

```
hheellll  wwrrlldd
>>> .
```

2.5. break ოპერატორი

break ოპერატორი იწვევს ციკლის უპირობოდ შეწყვეტას.

მოცემულ მაგალითში ციკლი შეწყვეტს, მუშაობას 'O' სიმბოლოსთან:

```
for i in 'hello world':
...   if i == 'o':
...     break
...   print(i * 2, end="")
```

შედეგი:
hheellll

მოცემულ მაგალითში ციკლში გამოყენებულია else. ინსტრუქციის ბლოკი else-ის შიგნით შესრულდება მხოლოდ იმ შემთხვევაში, თუ ციკლიდან გასვლა მოხდა break-ის გარეშე:

```
>>> for i in 'hello world':
...   if i == 'a':
...     break
... else:
...   print('a ასო არ არის სტრიქონში')
...
...
```

a ასო არ არის სტრიქონში

განვიხილოთ მაგალითი, ვალუტის გამცვლელი პუნქტი:

```
print("დასრულებისთვის დააჭირეთ Y")
while True:
    data = input("შეიტანეთ თანხა გაცვლისთვის: ")
    if data.lower() == "y":
        break # ციკლიდან გასვლა
    money = int(data)
```

```

cache = round(money / 2.56)
print("დაიცემა", cache, "დოლარი")
print("გაცვლითი პუნქტის მუშაობა დასრულებულია")

```

შედეგი:

```

== RESTART: C:/Users/Guliko/AppData/Local/Programs/Python/Python38-64/Python.exe
დასრულებისთვის დააჭირეთ Y
შეიტანეთ თანხა გაცვლისთვის: 100
დაიცემა 39.06 დოლარი
შეიტანეთ თანხა გაცვლისთვის: 200
დაიცემა 78.12 დოლარი
შეიტანეთ თანხა გაცვლისთვის: 450
დაიცემა 175.78 დოლარი
შეიტანეთ თანხა გაცვლისთვის: y
გაცვლითი პუნქტის მუშაობა დასრულებულია
>>>

```

2.6. ჩადგმული ციკლები

ერთი ციკლი შეიძლება შეიცავდეს სხვა ციკლებს. განვიხილოთ გამრავლების ცხრილის მაგალითი:

```

for i in range(1, 10):
    for j in range(1, 10):
        print(i * j, end="\t")
    print("\n")

```

გარე ციკლი for i in range(1, 10) იმუშავებს 9-ჯერ, რადგან range ფუნქციის მიერ დაბრუნებულ კოლექციაში 9

რიცხვია. შიდა ციკლი `for j in range(1, 10)` იმუშავებს 9-ჯერ გარე ციკლის ერთი იტერაციისთვის და შესაბამისად, 81-ჯერ გარე ციკლის ყველა იტერაციისთვის.

შიდა ციკლის ცალკეულ იტერაციაში კონსოლზე გამოიტანება `i` და `j` რიცხვების ნამრავლი. მივიღებთ შედეგს:

```
...
== RESTART: C:/Users/Guliko/AppData/Local/Programs/Python/Python36-32/bv.py :
1      2      3      4      5      6      7      8      9
2      4      6      8      10     12     14     16     18
3      6      9      12     15     18     21     24     27
4      8      12     16     20     24     28     32     36
5     10     15     20     25     30     35     40     45
6     12     18     24     30     36     42     48     54
7     14     21     28     35     42     49     56     63
8     16     24     32     40     48     56     64     72
9     18     27     36     45     54     63     72     81
```

III თავი ძირითადი მონაცემთა ტიპები

3.1. რიცხვები: მთელი, ნამდვილი, კომპლექსური

Python ენაში რიცხვები არაფრით განსხვავდება ჩვეულებრივი რიცხვებისგან. მათზე ვრცელდება ჩვეულებრივი მათემატიკური ოპერაციები:

$x + y$	შეკრება
$x - y$	გამოკლება
$x * y$	გამრავლება
x / y	გაყოფა
$x // y$	მთელირიცხვა გაყოფა
$x \% y$	გაყოფის შედეგად მიღებული ნაშთი
$-x$	რიცხვის ნიშნის შეცვლა
$abs(x)$	რიცხვის მოდული
$divmod(x, y)$	წყვილი ($x // y, x \% y$)
$x ** y$	ახარისხება
$pow(x, y[, z])$	x^y მოდულის მიხედვით (თუ მოდული მოცემულია)

უნდა აღინიშნოს, რომ python-ში მთელი რიცხვები, სხვა მრავალი ენებისგან განსხვავებით, მხარს უჭერენ გრძელ (მრავალ ოპერაციიან) არითმეტიკას, თუმცა ისინი მოითხოვენ დიდ მეხსიერებას.

```
>>> 255 + 34
```

```
289
```

```
>>> 5 * 2
```

```
10
```

```

>>> 20 / 3
6.666666666666667
>>> 20 // 3
6
>>> 20 % 3
2
>>> 3 ** 4
81
>>> pow(3, 4)
81
>>> pow(3, 4, 27)
0
>>> 3 ** 150
36998848503512697292470078245169664418647310038972297
3815184405301748249

```

ბიტური ოპერაციები

მთელ რიცხვებზე შეიძლება შესრულდეს ბიტური ოპერაციები.

$x y$	ან ოპერაცია
$x \wedge y$	გამომრიცხავი ან ოპერაცია
$x \& y$	და ოპერაცია
$x \ll n$	დაძვრა მარცხნივ
$x \gg y$	დაძვრა მარჯვნივ
$\sim x$	ბიტების ინვერსია

დამატებითი მეთოდები

მეთოდი `int.bit_length()` - იძლევა ბიტების რაოდენობას, რომელიც საჭიროა რიცხვის წარმოსადგენად ორობითი სახით, ნიშნის და საწყისი ნულის გათვალისწინების გარეშე:

```
>>> n=-37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

მეთოდი `int.to_bytes(length, byteorder, *, signed=False)` - დააბრუნებს ბაიტების სტრიქონს, რომელიც წარმოადგენს ამ რიცხვს.

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() // 8) + 1, byteorder='little')
b'\xe8\x03'
```

მეთოდი `int.from_bytes(bytes, byteorder, *, signed=False)` - დააბრუნებს რიცხვს ბაიტების მოცემული სტრიქონიდან.

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
```

16

```
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>>int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

3.2. ათვლის სისტემები

ათვლის ერთი სისტემიდან მეორეში რიცხვის გადასაყვანად Python ენაში არის რამდენიმე ფუნქცია:

- `int([object], [ათვლის სისტემის ფუძე])` - ათვლის ათობით სისტემაში გარდაქმნა. გაჩუმებით ათვლის სისტემა ათობითია, მაგრამ შეიძლება მიცემულ იქნას ნებისმიერი ფუძე 2-დან 36-ის ჩათვლით.
 - `bin(x)` - მთელი რიცხვის გარდაქმნა ორობით სტრიქონად.
 - `hex(x)` - მთელი რიცხვის გარდაქმნა 16-ობით სტრიქონად.
 - `oct(x)` - მთელი რიცხვის გარდაქმნა რვაობით სტრიქონად.
- განვიხილოთ მაგალითები:

```
>>> a = int('19') #სტრიქონის გადაყვანა რიცხვში
>>> b = int('19.5') # სტრიქონი არ არის მთელი რიცხვი
```

Traceback (most recent call last):

File "", line 1, in

ValueError: invalid literal for int() with base 10: '19.5'


```
>>> c = int(19.5) # გამოყენებულია მცურავწერტილიანი
რიცხვისთვის, აჭრის წილად ნაწილს.
```

```
>>> print(a, c)
```

```
19 19
```

```
>>> bin(19)
```

```
'0b10011'
```

```
>>> oct(19)
```

```
'0o23'
```

```
>>> hex(19)
```

```
'0x13'
```

```
>>> 0b10011 # ასე შეიძლება ჩაიწეროს რიცხვითი
მუდმივებიც
```

```
19
```

```
>>> int('10011', 2)
```

```
19
```

```
>>> int('0b10011', 2)
```

```
19
```

3.3. ნამდვილი რიცხვები - float

ნამდვილი რიცხვებიც მხარს უჭერენ იმავე ოპერაციებს, რასაც მთელი რიცხვები. მხოლოდ, კომპიუტერში რიცხვის წარმოდგენიდან გამომდინარე ნამდვილი რიცხვები არაზუსტია, რამაც შეიძლება გამოიწვიოს ცდომილება:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
```

```
0.9999999999999999
```

მაღალი სიზუსტისთვის გამოიყენება სხვა ობიექტები (მაგალითად Decimal და Fraction). ნამდვილი რიცხვები არ უჭერენ მხარს გრძელ (მრავალპერაციიან) არითმეტიკას:

```
>>> a = 3 ** 1000
```

```
>>> a + 0.1
```

Traceback (most recent call last):

File "", line 1, in

OverflowError: int too large to convert to float

რიცხვებთან მუშაობის მაგალითები:

```
>>> c = 150
```

```
>>> d = 12.9
```

```
>>> c + d
```

```
162.9
```

```
>>> p = abs(d - c) # Модуль числа
```

```
>>> print(p)
```

```
137.1
```

```
>>> round(p) # Округление
```

```
137
```

დამატებითი მეთოდები

`loat.as_integer_ratio()` - მთელი რიცხვების წყვილი, რომელთა დამოკიდებულება ტოლია ამავე რიცხვის.

`float.is_integer()` - არის თუ არა მნიშვნელობა მთელი რიცხვი.

`float.hex()` - გადაიყვანს float-ს hex (ათვლის თექვსმეტობით სისტემაში).

`float.fromhex(s)` - თექვსმეტობითიდან float-ში გადაყვანა.

```
>>> (10.5).hex()
'0x1.5000000000000p+3'
>>> float.fromhex('0x1.5000000000000p+3')
10.5
```

სტანდარტული გამოსახულებების გარდა რიცხვებთან სამუშაოდ Python ენაში არის რამდენიმე სასარგებლო მოდული.

`math` მოდული - წარმოადგენს შედარებით რთულ მათემატიკურ ფუნქციებს.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.sqrt(85)
9.219544457292887
```

`random` მოდული აგენერირებს შემთხვევით რიცხვებს და შემთხვევითი არჩევის ფუნქციებს.

```
>>> import random
>>> random.random()
0.15651968855132303
```

3.4. კომპლექსური რიცხვები - complex

Python ენაში ჩაშენებულია ასევე, კომპლექსური რიცხვები:

```
>>> x = complex(1, 2)
>>> print(x)
(1+2j)
>>> y = complex(3, 4)
>>> print(y)
(3+4j)
>>> z = x + y
>>> print(x)
(1+2j)
>>> print(z)
(4+6j)
>>> z = x * y
>>> print(z)
(-5+10j)
>>> z = x / y
>>> print(z)
(0.44+0.08j)
>>> print(x.conjugate()) # კონიუგატის რიცხვი
(1-2j)
>>> print(x.imag) # წარმოსახვითი ნაწილი
2.0
>>> print(x.real) # ნამდვილი ნაწილი
1.0
```

```

>>> print(x > y) # კომპლექსური რიცხვების შედარება
არ შეიძლება
Traceback (most recent call last):
  File "", line 1, in
TypeError: unorderable types: complex() > complex()
>>> print(x == y) # შეიძლება შემოწმდეს ტოლობაზე
False
>>> abs(3 + 4j) # კომპლექსური რიცხვის მოდული
5.0
>>> pow(3 + 4j, 2) # ახარისხება
(-7+24j)

```

3.5. სტრიქონები

სტრიქონები Python-ში - სიმბოლოების მოწესრიგებული მიმდევრობაა, რომელიც გამოიყენება ტექსტური ინფორმაციის შესანახად და წარმოსადგენად. ამიტომ სტრიქონების დახმარებით შეიძლება მუშაობა ყველა ინფორმაციასთან, რომელიც წარმოდგენილია ტექსტური ფორმით.

სტრიქონების ლიტერალები

Python-ში სტრიქონებთან მუშაობა ძალიან მოხერხებულია. არსებობს სტრიქონების რამდენიმე ლიტერალი, რომელსაც განვიხილავთ.

სტრიქონები აპოსტროფებში და ბრჭყალებში

S = 'spam"s'

S = "spam's"

სტრიქონები აპოსტროფებში და ბრჭყალებში ერთიდაიგივეა. ორი ვარიანტის არსებობა იმაშია, რომ იყოს ტექსტში აპოსტროფის ან ბრჭყალების ჩასმის შესაძლებლობა.

ეკრანირებული მიმდევრობები - მომსახურე სიმბოლოები

ეკრანირებული მიმდევრობები საშუალებას იძლევა ჩაისვას სიმბოლოები, რომელთა შეტანაც რთულია კლავიატურიდან.

ეკრანირებული მიმდევრობა	დასახელება
\n	სტრიქონის გადატანა
\a	ხმა
\b	ერთი პოზიციით დაბრუნება
\f	გვერდის გადატანა
\r	კურსორის გადატანა სტრიქონის დასაწყისში
\t	ჰორიზონტალური ტაბულაცია
\v	ვერტიკალური ტაბულაცია
\N{id}	უნიკოდის მონაცემთა ბაზის ID იდენტიფიკატორი
\uhhhh	უნიკოდის 16-ბიტის სიმბოლო 16-ობით წარმოდგენაში
\Uhhhh...	უნიკოდის 32-ბიტის სიმბოლო 32-ობით წარმოდგენაში
\xhh	სიმბოლოს 16-ობითი მნიშვნელობა

\ooo	სიმბოლოს 8-ობითი მნიშვნელობა
\0	სიმბოლო Null (არ არის სტრიქონის დასრულების ნიშანი)

თუ გახსნილი ბრჭყალის წინ დგას სიმბოლო 'r' (ნებისმიერ რეგისტრში), ეკრანირების მექანიზმი გაუქმდება.

S = r'C:\newt.txt'

სტრიქონი არ შეიძლება დამთავრდეს შებრუნებული სლემის სიმბოლოთი. გადაწყვეტის გზები:

S = r'\n\n\\[:-1]

S = r'\n\n' + '\\'

S = '\\n\n'

სტრიქონები სამმაგ აპოსტროფებში ან ბრჭყალებში

სამმაგ ბრჭყალებში ჩასმული სტრიქონის მთავარი ღირსება არის, რომ მათი გამოყენება შეიძლება ტექსტის მრავალსტრიქონიანი ბლოკების ჩასაწერად. ასეთი სტრიქონის შიგნით შეიძლება ბრჭყალების და აპოსტროფების არსებობა, მთავარია, რომ არ იყოს სამი ბრჭყალი მიმდევრობით.

>>> c = '''ეს ძალიან დიდი

...სტრიქონია, ტექსტის

... მრავალსტრიქონიანი ბლოკი'''

>>> c

'ეს ძალიან დიდი\nსტრიქონია, ტექსტის\nმრავალსტრიქონიანი ბლოკი'

```
>>> print(c)
ეს ძალიან დიდი
სტრიქონია, ტექსის
მრავალსტრიქონიანი ბლოკი
```

3.5.1. საბაზო ოპერაციები სტრიქონებზე

კონკატენაცია (შეკრება):

```
>>> S1 = 'spam'
>>> S2 = 'eggs'
>>> print(S1 + S2)
'spameggs'
```

სტრიქონის დუბლირება:

```
>>> print('spam' * 3)
spamspamspam
```

სტრიქონის სიგრძე - len ფუნქცია

```
>>> len('spam')
4
```

ინდექსზე წვდომა:

```
>>> S = 'spam'
>>> S[0]
's'
>>> S[2]
'a'
>>> S[-2]
```


'a'

როგორც ხედავთ Python-ში შესაძლებელია უარყოფით ინდექსზე წვდომა. ამ შემთხვევაში ათვლა ხდება სტრიქონის ბოლოდან.

სტრიქონიდან ფრაგმენტის ამოჭრა.

ზოგადი ფორმა: [X:Y]. X - ჭრის დასაწყისია, ხოლო Y - ჭრის დასასრული. სიმბოლო Y ნომრით ჭრაში არ შედის. დუმილით პირველი ინდექსია 0, ხოლო მეორე - სტრიქონის სიგრძე.

```
>>> s = 'spameggs'
```

```
>>> s[3:5]
```

```
'me'
```

```
>>> s[2:-2]
```

```
'ameg'
```

```
>>> s[:6]
```

```
'spameg'
```

```
>>> s[1:]
```

```
'pameggs'
```

```
>>> s[:]
```

```
'spameggs'
```

შეიძლება მიეთითოს ბიჯი, რომელთანაც საჭიროა ჭრის განხორციელება

```
>>> s[::-1]
```

```
'sggemaps'
```

```
>>> s[3:5:-1]
```

```
"
>>> s[2::2]
'aeg'
```

3.5.2. ფუნქციები და მეთოდები სტრიქონებთან სამუშაოდ

მეთოდის გამოძახებისას საჭიროა გახსოვდეთ, რომ Python-ში სტრიქონები მიეკუთვნება კატეგორიას, რომელთა მიმდევრობა არ იცვლება ანუ ყველა ფუნქციას და მეთოდს შეუძლია მხოლოდ შექმნას ახალი სტრიქონი.

```
>>> s = 'spam'
>>> s[1] = 'b'
```

Traceback (most recent call last):

File "", line 1, in

```
s[1] = 'b'
```

TypeError: 'str' object does not support item assignment

```
>>> s = s[0] + 'b' + s[2:]
```

```
>>> s
```

```
'sbam'
```

ამიტომ, ყველა სტრიქონული მეთოდი აბრუნებს ახალ სტრიქონს, რომელიც შეიძლება მიენიჭოს ცვლადს.

მეთოდები და ფუნქციები სტრიქონებისთვის

ფუნქცია/მეთოდი	დანიშნულება
S = 'str'; S = "str"; S = '''str'''; S = ""str""	სტრიქონების ლიტერალები

<code>S = "\n\r\t\nbbb"</code>	ეკრანირებული მიმდევრობები
<code>S = r"C:\temp\new"</code>	დაუფორმატებელი სტრიქონები
<code>S = b"byte"</code>	ბაიტების სტრიქონი
<code>S1 + S2</code>	კონკატენაცია (სტრიქონების გაერთიანება)
<code>S1 * 3</code>	სტრიქონის განმეორება
<code>S[i]</code>	ინდექსით მიმართვა
<code>S[i:j:step]</code>	ჭრა სტრიქონში
<code>len(S)</code>	სტრიქონის სიგრძე
<code>S.find(str, [start],[end])</code>	სტრიქონში ქვესტრიქონის ძებნა. აბრუნებს პირველი შესვლის ნომერს ან -1-ს.
<code>S.rfind(str, [start],[end])</code>	სტრიქონში ქვესტრიქონის ძებნა. აბრუნებს ბოლო შესვლის ნომერს ან -1-ს.
<code>S.index(str, [start],[end])</code>	სტრიქონში ქვესტრიქონის ძებნა. აბრუნებს პირველი შესვლის ნომერს ან იძახებს ValueError
<code>S.rindex(str, [start],[end])</code>	სტრიქონში ქვესტრიქონის ძებნა. აბრუნებს ბოლო შესვლის ნომერს ან იძახებს ValueError
<code>S.replace(шаблон, замена)</code>	შაბლონის შეცვლა
<code>S.split(символ)</code>	სტრიქონის გახლეჩა
<code>S.isdigit()</code>	შედგება თუ არა სტრიქონი ციფრისგან.
<code>S.isalpha()</code>	შედგება თუ არა სტრიქონი ასოსგან
<code>S.isalnum()</code>	შედგება თუ არა სტრიქონი ასოსგან ან ციფრისგან
<code>S.islower()</code>	შედგება თუ არა სტრიქონი ქვედა რეგისტრის სიმბოლოებისგან
<code>S.isupper()</code>	შედგება თუ არა სტრიქონი ზედა რეგისტრის სიმბოლოებისგან

S.isspace()	შედგება თუ არა სტრიქონი უხილავი სიმბოლოსგან (ხარვეზი, '\f', '\n', '\r', '\t', '\v')
S.istitle()	იწყება თუ არა სტრიქონში სიტყვა დიდი ასოთი.
S.upper()	სტრიქონის გადაყვანა ზედა რეგისტრში.
S.lower()	სტრიქონის გადაყვანა ქვედა რეგისტრში.
S.startswith(str)	იწყება თუ არა S სტრიქონი str შაბლონით
S.endswith(str)	მთავრდება თუ არა S სტრიქონი str შაბლონით
S.join(სია)	სიდიდან სტრიქონის აწყობა
ord(სიმბოლო)	სიმბოლოს ASCII კოდში წარმოდგენა
chr(რიცხვი)	ASCII კოდის სიმბოლოდ წარმოდგენა
S.capitalize()	გადაჰყავს სტრიქონის პირველი სიმბოლო ზედა რეგისტრში, დანარჩენები ქვედაში.
S.center(width, [fill])	აბრუნებს ცენტრირებულ სტრიქონს, რომლის კიდეებში არის შვესების სიმბოლო, დუმილით ხარვეზი.
S.count(str, [start],[end])	ითვლის სტრიქონში ქვესტრიქონის შესვლათა რაოდენობას.
S.expandtabs([tabsize])	დააბრუნებს სტრიქონის ასლს, სადაც ყველა ტაბულაციის სიმბოლო ერთი ან რამდენიმე ხარვეზით, მიმდინარე სვეტიდან გამომდინარე. თუ TabSize არ არის მითითებული, ტაბულაციის ზომა განისაზღვრება 8 ხარვეზის ტოლი.
S.lstrip([chars])	ხარვეზი სიმბოლოების წაშლა

	სტრიქონის დასაწყისში.
S.rstrip([chars])	ხარვეზი სიმბოლოების წაშლა სტრიქონის ბოლოში.
S.strip([chars])	ხარვეზი სიმბოლოების წაშლა სტრიქონის დასაწყისში და ბოლოში.
S.partition(template)	აბრუნებს კორტეჟს, რომელიც შეიცავს ნაწილს პირველი შაბლონის წინ, თვით შაბლონს და ნაწილს შაბლონის შემდეგ. თუ შაბლონი არ არის ნაპოვნი, დაბრუნდება კორტეჟი, რომელიც შეიცავს თვით სტრიქონს და შემდეგ ორ ცარიელ სტრიქონს.
S.rpartition(sep)	აბრუნებს კორტეჟს, რომელიც შეიცავს ნაწილს ბოლო შაბლონის წინ, თვით შაბლონს და ნაწილს შაბლონის შემდეგ. თუ შაბლონი არ არის ნაპოვნი, დაბრუნდება კორტეჟი, რომელიც შეიცავს ორ ცარიელ სტრიქონს და შემდეგ თვით სტრიქონს.
S.swapcase()	გადაჰყავს სიმბოლოები ზედა რეგისტრიდან ქვედაში და ქვედადან ზედაში.
S.title()	ცალკეული სიტყვის პირველი სიმბოლო გადაჰყავს ზედა რეგისტრში, ყველა დანარჩენი ქვედაში.
S.zfill(width)	გადააკეთებს სტრიქონს არანაკლებ width სიგრძისა, საჭიროებისამებრ შეავსებს პირველ სიმბოლოებს ნულებით.
S.ljust(width, fillchar=" ")	გადააკეთებს სტრიქონს არანაკლებ width სიგრძისა, საჭიროებისამებრ შეავსებს ბოლო სიმბოლოებს fillchar-ით.

<code>S.rjust(width, fillchar="")</code>	გადააკეთებს სტრიქონს არანაკლებ <code>width</code> სიგრძისა, საჭიროებისამებრ შეავსებს პირველ სიმბოლოებს <code>fillchar</code> სიმბოლოთი.
<code>S.format(*args, **kwargs)</code>	სტრიქონის ფორმატირება

3.5.3. სტრიქონის ფორმატირება

ხშირად წარმოიშვება სიტუაციები, როდესაც უნდა შეიქმნას სტრიქონი, რომელშიც ჩაისვას პროგრამის შესრულების პროცესში მიღებული მონაცემები. ასეთი შეიძლება იყოს სამომხმარებლო შეტანა, მონაცემები ფაილიდან. მსგავს შემთხვევებში საჭიროა სტრიქონების ფორმატირება, რომელიც შეიძლება განხორციელდეს % ოპერატორით ან `format` მეთოდის დახმარებით.

თუ ჩანაცვლებისთვის მოითხოვება მხოლოდ ერთი არგუმენტი, მაშინ მნიშვნელობა არის თვით არგუმენტი.

```
>>> 'Hello, {}'.format('ANNA')
'Hello, ANNA!'
```

თუ ჩანაცვლებისთვის მოითხოვენ რამდენიმე არგუმენტი, მაშინ მნიშვნელობები იქნება ყველა არგუმენტი:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
```

```
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
>>>Coordinates:{latitude},
```

```
{longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

განვიხილოთ format მეთოდის სინტაქსი:

შეცვლის ველი ::= "{" [ველის სახელი] ["!" გარდაქმნა] [":" სპეციფიკაცია] "}"

ველის სახელი ::= arg_name ("." ატრიბუტის სახელი | "[" ინდექსი "]") *

გარდაქმნა ::= "r" (შიდა წარმოდგენა) | "s" (ადამიანური წარმოდგენა)

სპეციფიკაცია ::= იხ. ქვევით

განვიხილოთ მაგალითი:

```
>>> "Units destroyed: {players[0]}".format(players = [1, 2, 3])
'Units destroyed: 1'
>>> "Units destroyed: {players[0]:!r}".format(players = ['1', '2', '3'])
'Units destroyed: '1''
```

ფორმატის სპეციფიკაცია:

სპეციფიკაცია ::=

[[fill]align][sign][#][0][width][,][.precision][type]

დაგროვება ::= სიმბოლო გარდა '{' ან '}'

გათანაბრება ::= "<" | ">" | "=" | "^"

ნიშანი ::= "+" | "-" | " "

სიგანე ::= integer

სიზუსტე ::= integer

ტიპი ::= "b" | "c" | "d" | "e" | "E" | "f" | "F" | "g" | "G" |
"n" | "o" | "s" | "x" | "X" | "%"

გათანაბრება ხდება სიმბოლო-გათანაბრებლის მეშვეობით. დასაშვებია გათანაბრების შემდეგი ვარიანტები:

ალამი	მნიშვნელობა
'<'	ობიექტის გათანაბრება მარცხენა კიდის მიხედვით (დუმლით).
'>'	ობიექტის გათანაბრება მარჯვენა კიდის მიხედვით
'='	შემავსებელი იქნება ნიშნის შემდეგ, მაგრამ ციფრების წინ. მუშაობს მხოლოდ რიცხვით ტიპებთან.
'^'	ცენტრის მიმართ გასწორება.

ოპცია „ნიშანი“ გამოიყენება მხოლოდ რიცხვებისთვის და შეიძლება მიიღოს შემდეგი მნიშვნელობები:

ალამი	მნიშვნელობა
'+'	შეიძლება გამოყენებულ იქნას ყველა რიცხვისთვის
'_'	‘-’ უარყოფითისთვის, არაფერი - დადებითისთვის
ხარვეზი	‘-’ უარყოფითისთვის, ხარვეზი - დადებითისთვის

„ტიპი“ ველმა შეიძლება მიიღოს შემდეგი მნიშვნელობები:

ტიპი	მნიშვნელობა
d', 'i', 'u'	ათობითი რიცხვი
'o'	რიცხვი ათვლის რვაობით სისტემაში
'x'	რიცხვი ათვლის 16-ობით სისტემაში(ასოები ქვედა რეგისტრში)
'X'	რიცხვი ათვლის 16-ობით სისტემაში(ასოები ზედა რეგისტრში)
'e'	რიცხვი მცურავი წერტილით ექსპონენტი. ექსპონენტა ქვედა რეგისტრში.
'E'	რიცხვი მცურავი წერტილით ექსპონენტი. ექსპონენტა ზედა რეგისტრში.
'f', 'F'	რიცხვი მცურავი წერტილით (ჩვეულებრივი ფორმატი).
'g'	რიცხვი მცურავი წერტილით ექსპონენტი, (ექსპონენტა ქვედა რეგისტრში) თუ იგი ნაკლებია 4-ზე ან სიზუსტით, წინააღმდეგ შემთხვევაში ჩვეულებრივი ფორმატი.
'G'	რიცხვი მცურავი წერტილით ექსპონენტი, (ექსპონენტა ზედა რეგისტრში) თუ იგი

	ნაკლებია4-ზე ან სიზუსტით, წინააღმდეგ შემთხვევაში ჩვეულებრივი ფორმატი.
'c'	სიმბოლო (ერთსიმბოლოიანი სტრიქონი ან რიცხვი - სიმბოლოს კოდი)
's'	სტრიქონი.
'%'	რიცხვი მრავლდება 100-ზე, აისახება მცურავწერტილიანი რიცხვი, მის შემდეგ ნიშანი %.

განვიხილოთ მაგალითები:

```
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
>>> "repr() shows quotes: {!r}; str() doesn't: {!s}".format('test1',
'test2')
"repr() shows quotes: 'test1'; str() doesn't: test2"
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
>>> '{:*^30}'.format('centered') # use '*' as a fill char
!*****centered*****!
>>> '{:+f}; {+f}'.format(3.14, -3.14) # show it always
'+3.140000; -3.140000'
>>> '{: f}; { f}'.format(3.14, -3.14) # show a space for positive
numbers
```

```
' 3.140000; -3.140000'
>>> '{:-f}; {-f}'.format(3.14, -3.14) # show only the minus --
same as '{:f}; {f}'
'3.140000; -3.140000'
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
>>> points = 19.5
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 88.64%'
```

სტრიქონებთან მუშაობის მაგალითები

1) პროგრამა ბეჭდავს ორ სტრიქონს:

```
name = "Tom"
surname = 'Smith'
print(name, surname) # Tom Smith
```

2) ორი სტრიქონი გაერთიანდა და შემდეგ იბეჭდება გაერთიანებული სტრიქონი:

```
name = "Tom"
surname = 'Smith'
fullname = name + " " + surname
print(fullname) # Tom Smith
```

3) მოცემულ პროგრამაში საჭიროა შეიკრიბოს სტრიქონი და რიცხვი. ამ შემთხვევაში საჭიროა რიცხვის გადაყვანა სტრიქონად str() ფუნქციის გამოყენებით:

```
name = "Tom"
age = 32
info = "Name: " + name + " Age: " + str(age)
print(info) # Name: Tom Age: 32
```

4) გამოიტანს ტექსტს ორ სტრიქონად:

```
print ("ვსწავლობთ python ენაზე დაპროგრამებას\n-  
სტრიქონებთან მუშაობა საინტერესოა")  
კონსოლზე გამოვა ორი სტრიქონი:
```

ვსწავლობთ python ენაზე დაპროგრამებას
სტრიქონებთან მუშაობა საინტერესოა

5) ტექსტში ბრჭყალების ასახვისთვის, მის წინ იწერება შებრუნებული სლემი:

```
print("Cafe \"Central Perk\"")
```

შედეგი: Cafe „Central Perk“

6) სტრიქონების შედარებისას მხედველობაში მიიღება სიმბოლოები და მათი რეგისტრები. ციფრული სიმბოლო პირობითად ნაკლებია, ვიდრე ნებისმიერი ანბანის სიმბოლო. ანბანური სიმბოლო ზედა რეგისტრში პირობითად ნაკლებია, ვიდრე ანბანური სიმბოლო ქვედა რეგისტრში. მაგალითად:

```
str1 = "1a"
```

```
str2 = "aa"  
str3 = "Aa"  
print(str1 > str2) # False, რადგან str1 - ში პირველი  
სიმბოლო ციფრია
```

```
print(str2 > str3) # True, რადგან str2 -ში პირველი  
სიმბოლო ქვედა რეგისტრშია.
```

თუ საწყისი სიმბოლოები ორივე სტრიქონში ციფრია, უმცირესად ითვლება უმცირესი ციფრი. მაგალითად: 1a<2a.

თუ საწყისი ანბანური სიმბოლოები ერთიდაიმავე რეგისტრშია, განიხილება ანბანის მიხედვით. მაგალითად: aa<ba, ba<ca.

უმჯობესია ორივე სტრიქონი გადაყვანილ იქნას ერთნაირ რეგისტრში.

7) პროგრამაში სტრიქონები გადადის ერთნაირ რეგისტრებში და ხდება მათი შედარება:

```
tr1 = "Tom"  
str2 = "tom"  
print(str1 == str2) # False - არ არის ტოლი  
print(str1.lower() == str2.lower()) # True
```

8) სტრიქონის განსაზღვრულ პოზიციაში სიმბოლოს შეცვლის მცდელობა იწვევს შეცდომას:

```
>>> word = 'strength'  
>>> word[2] = 'y'
```

TypeError: 'str' object does not support item assignment

9) სტრიქონში სიმბოლოს შეცვლა შეიძლება შემდეგი სახით:

```
>>> word = word[:3] + '!' + word[4:]  
      'str!ngth'
```

10) სტრიქონში სიმბოლოს შეცვლა შეიძლება შემდეგი სახითაც:

```
>>> word = word.replace('!', 'e')  
      'strenght'
```

11) ათვლა ხდება ბოლოდან:

```
>>> word[-1]  
h
```

12) ერთი ტიპის ბრჭყალები ნებისმიერად ჩაშენდება სხვა ტიპის ბრჭყალებში:

```
>>> '123'  
'123'  
>>> "7'8'9"  
"7'8'9"
```

13) გრძელი სტრიქონები გაყოფილია რამდენიმე ნაწილად მებრუნებული სლემით:

```
>>> s = 'this is first word\  
and this is second word'
```

14) სტრიქონების დიდი ნაკრები და მთელი ტექსტი შეიძლება ჩაისვას სამმაგ ბრჭყალებში:

```
>>> print ""  
One
```

```
Two
Three
""
```

15) მაგალითში გამოყენებულია მმართველი მიმდევრობები:

```
>>> s = 'a\nb\tc'
>>> print s
a
b c
```

16) მაგალითში სტრიქონი შედგება სამი რიცხვის ბინარული მიმდევრობისგან - ორი რვაობითი და ერთი თექვსმეტობითი:

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
>>> len(s)
3
```

17) სტრიქონის ჭრები:

```
>>> word = 'strength'
>>> word[4]
n
>>> word[0:2]
st
>>> word[2:4]
re
```

18) ჭრაში გამოტოვებულია ჯერ პირველი სიმბოლო, შემდეგ - მეორე სიმბოლო:

```
>>> word[:3]
```

```
str
```

```
>>> word[5:]
```

```
gth
```

19) სტრიქონიდან სიმბოლოების მიმდევრობის არჩევა ციკლურობით:

```
>>> s = '1234567890'
```

```
>>> s[::2]
```

```
'13579'
```

```
>>> s[1:10:2]
```

```
'24680'
```

```
>>> s[::-1]
```

```
'0987654321'
```

20) სტრიქონების გამრავლება:

```
>>> '123' * 3
```

```
'123123123'
```

20) სტრიქონების ფორმატირება. პროცენტის მარცხნივ მიეთითება სტრიქონი, მარჯვნივ - მნიშვნელობა ან მნიშვნელობათა ჩამონათვალი:

```
>>> s = 'Hello %s' % 'world'
```

```
>>> s
```

```
'Hello world'
```

```
>>> s = 'one %s %s' % ('two','three')
```

```
>>> s
```



```
'one two three'
```

21) რიცხვის სტრიქონში გადასაყვანად გამოყენებულია რიცხვითი სპეციფიკატორი - %d ან %f:

```
>>> s = 'one %d %f' % (2, 3.5)
```

```
>>> s
```

```
'one 2 3.500000'
```

22) მაგალითში შედეგის სტრიქონს ექნება სიგრძე 10 სიმბოლო, წილად ნაწილში გამოყოფილია 5 სიმბოლო:

```
>>> x = 4/3
```

```
>>> '%10.5f' % x
```

```
' 1.33333'
```

23) ხარვეზები მარცხნიდან შეიძლება დაფორმატდეს ნულებით:

```
>>> from math import pi
```

```
>>> '%015.10f' % pi
```

```
'0003.1415926536'
```

24) მაგალითში ფორმატირებისთვის გამოყენებულია Template - სტრიქონების შაბლონები:

```
>>> from string import Template
```

```
>>> s = Template('1 $two 3 4 $five')
```

```
>>> d={}
```

```
>>> d['two']=2
```

```
>>> d['five']=5
```

```
>>> s.substitute(d)
```

```
'1 2 3 4 5'
```

25) ქვესტრიქონის ძებნა სტრიქონში:

```
>>> s = 'The find method finds a substring'  
>>> s.find('find')  
4  
>>> s.find('finds')  
16  
>>> s.find('findsa')  
-1
```

26) join - აერთიანებს გამომყოფი ნიშნით სტრიქონების ნაკრებს:

```
>>> seq = ['one','two','three']  
>>> sep = ','  
>>> sep.join(seq)  
'one,two,three'
```

27) split - დაყოფს სტრიქონს მიმდევრობად:

```
>>> s = '/usr/local/bin'  
>>> s.split('/')  
['', 'usr', 'local', 'bin']
```

28) replace - სტრიქონში ერთ ქვესტრიქონს ჩაანაცვლებს მეორეთი:

```
>>> s = 'replace method returns a string'  
>>> s.replace('returns','return')  
'replace method return a string'
```

29) strip - შლის ხარვეზს მარცხნიდან და მარჯვნიდან:

```
>>> ' this is whitespace string '.strip()  
'this is whitespace string'
```

30) translate - აკეთებს მრავლობით შეცვლას. მაგალითში ყველა სიმბოლო '1' შეიცვლება სიმბოლო '3'-ით, ხოლო სიმბოლო '2', შეიცვლება სიმბოლოთი - '4':

```
>>> from string import maketrans
>>> table = maketrans('12', '34')
>>> '1212 5656'.translate(table)
'3434 5656'
```

3.6. სიები

Python-ში არის რამდენიმე მონაცემთა ტიპი, რომლებიც გამოიყენება რამდენიმე მნიშვნელობის დასაჯგუფებლად. ყველაზე მოქნილი არის სია (list), რომელიც შეიძლება ჩაიწეროს მძიმეებით გამოყოფილი და კვადრატულ ფრჩხილებში ჩასმული ელემენტების სიის სახით. არ არის აუცილებელი, რომ ელემენტები იყოს ერთი ტიპის.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

ინდექსების ნუმერაცია იწყება 0-დან. სიებისთვის შეიძლება განხორციელდეს ჭრა, გაერთიანება და სხვა:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
```

```

100
>>> a[1:-1]
['eggs', 100]
>>> a[2:] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam',
'eggs', 100, 'Boe!']

```

სტრიქონებისგან განსხვავებით, რომლებიც შეუცვლელია, არსებობს სიის ცალკეული ელემენტის შეცვლის შესაძლებლობა:

```

>>> a
['spam', 'eggs', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['spam', 'eggs', 123, 1234]

```

ჭრაზე მინიჭება ასევე შესაძლებელია, რამაც შეიძლება გამოიწვიოს სიის ზომის ცვლილება:

```

>>> # რამდენიმე ელემენტის შეცვლა:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # წაშლა“
... a[0:2] = []
>>> a
[123, 1234]
>>> # ჩასმა:

```

```
... a[1:1] = ['bletch', 'xyzyz']
>>> a
[123, 'bletch', 'xyzyz', 1234]
>>> # თავისივე ასლის ჩასმა დასაწყისში:
>>> a[:0] = a
>>> a
[123, 'bletch', 'xyzyz', 1234, 123, 'bletch', 'xyzyz',
1234]
```

სიებისთვის გამოიყენება ჩაშენებული ფუნქცია len():

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

```
8
```

სიები შეიძლება იყოს ჩაშენებული, როდესაც ერთი სია შეიცავს სხვა სიას, მაგალითად:

```
>>> q = [2, 3]
```

```
>>> p = [1, q, 4]
```

```
>>> len(p)
```

```
3
```

```
>>> p[1]
```

```
[2, 3]
```

```
>>> p[1][0]
```

```
2
```

```
>>> p[1].append('xtra')
```

```
>>> p
```

```
[1, [2, 3, 'xtra'], 4]
```

```
>>> q
```

```
[2, 3, 'xtra']
```

append() მეთოდი ელემენტს დაამატებს სიის ბოლოში. მიაქციეთ ყურადღება, რომ p[1] და q სინამდვილეში იგზავნება ერთიდაიმავე ობიექტზე.

სია შეიძლება შეიქმნას ლიტერალის მეშვეობითაც:

```
>>> s = [] # ცარიელი სია
>>> l = ['s', 'p', ['isok'], 2]
>>> s
[]
>>> l
['s', 'p', ['isok'], 2]
```

როგორც მაგალითიდან ჩანს, სია შეიძლება შეიცავდეს ნებისმიერი ობიექტის ნებისმიერ რაოდენობას, მათ შორის ჩაშენებულ სიებს ან არ შეიცავდეს არაფერს.

სია შეიძლება შეიქმნას, ასევე სიების გენერატორის მეშვეობით. სიების გენერატორი არის ახალი სიის შექმნის ხერხი, რომელიც იყენებს გამოსახულებას მიმდევრობის ცალკეულ ელემენტთან. სიების გენერატორი ძალიან ჰგავს for ციკლს.

```
>>> c = [c * 3 for c in 'list']
print(c)
შედეგი:
['lll', 'iii', 'sss', 'ttt']
```

შეიძლება სიების გენერატორის უფრო რთული კონსტრუქცია:

```
>>> c = [c * 3 for c in 'list' if c != 'i']
>>> print (c)
```

```
['lll', 'sss', 'ttt']
```

```
>>> c = [c + d for c in 'list' if c != 'i' for d in 'spam' if d != 'a']
```

```
>>> print (c)
```

```
['ls', 'lp', 'lm', 'ss', 'sp', 'sm', 'ts', 'tp', 'tm']
```

ჩაშენებულ შემთხვევებში უმჯობესია გამოიყენოთ ჩვეულებრივი for ციკლი სიების გენერაციისთვის.

3.6.1. მეთოდები სიებისთვის

მეთოდი	მოქმედება
<code>list.append(x)</code>	დაამატებს ელემენტს სიის ბოლოში
<code>list.extend(L)</code>	გააფართოვებს სიას ბოლოში L სიის ყველა ელემენტის დამატებით.
<code>list.insert(i, x)</code>	i-ურ ელემენტზე ჩასვამს x მნიშვნელობას
<code>list.remove(x)</code>	სიიდან წაშლის პირველ ელემენტს, რომელსაც აქვს x მნიშვნელობა. <code>ValueError</code> , თუ არ არის ასეთი მნიშვნელობა.
<code>list.pop([i])</code>	წაშლის i-ურ ელემენტს და დააბრუნებს მას. თუ ინდექსი არ არის მითითებული წაშლის ბოლო ელემენტს.
<code>list.index(x, [start [, end]])</code>	დააბრუნებს პირველივე x მნიშვნელობის მქონე ელემენტის მდებარეობას. ძებნა

	მიმდინარეობს თავიდან ბოლომდე.
<code>list.count(x)</code>	დააბრუნებს x მნიშვნელობის მქონე ელემენტების რაოდენობას.
<code>list.sort([key=function])</code>	დაალაგებს სიას ფუნქციის საფუძველზე.
<code>list.reverse()</code>	შეაბრუნებს სიას
<code>list.copy()</code>	სიის ზედაპირული ასლი
<code>list.clear()</code>	ასუფთავებს სიას

სიების მეთოდები ცვლიან თვით სიებს, ამდენად შესრულების შედეგი არ არის საჭირო ჩაიწეროს ამ ცვლადში.

```
>>> l = [1, 2, 3, 5, 7]
>>> l.sort()
>>> l
[1, 2, 3, 5, 7]
>>> l = l.sort()
>>> print(l)
None
```

მაგალითები:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
```



```
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

მაგალითები: სიების შექმნა, შეცვლა, წაშლა და მუშაობა სიის ელემენტებთან

სია შეიძლება შეიქმნას ჩამოთვლილთაგან ერთ-ერთი ხერხით:

```
>>> a = []
>>> type(a)
<class 'list'>
>>> b = list()
>>> type(b)
<class 'list'>
```

ასევე, შეიძლება სიის შექმნა წინასწარ მოცემული მონაცემთა ნაკრებით:

```
>>> a = [1, 2, 3]
>>> type(a)
```

```
<class 'list'>
```

სია თუ უკვე არსებობს და თქვენ გსურთ მისი ასლის შექმნა, შეგიძლიათ გამოიყენოთ შემდეგი ხერხი:

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = a[:]
```

```
>>> print(a)
```

```
[1, 3, 5, 7]
```

```
>>> print(b)
```

```
[1, 3, 5, 7]
```

ან შეიძლება ასეც:

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = list(a)
```

```
>>> print(a)
```

```
[1, 3, 5, 7]
```

```
>>> print(b)
```

```
[1, 3, 5, 7]
```

თუ სიებისთვის შეასრულებთ უბრალო მინიჭებას, მაშინ `b` ცვლადს მიენიჭება მიმართვა იმავე ელემენტზე მესხიერებაში, რომელზეც იგზავნება `a` და არა `a` სიის ასლი. ანუ თუ თქვენ შეცვლით `a` სიას, მაშინ `b` სიაც აგრეთვე შეიცვლება.

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = a
```

```
>>> print(a)
```

```
[1, 3, 5, 7]
```

```
>>> print(b)
```

```
[1, 3, 5, 7]
>>> a[1] = 10
>>> print(a)
[1, 10, 5, 7]
>>> print(b)
[1, 10, 5, 7]
```

მოცემულ მაგალითში სიას ემატება ელემენტი:

```
>>> a = []
>>> a.append(3)
>>> a.append("hello")
>>> print(a)
[3, 'hello']
```

მაგალითში წარმოდგენილია სიიდან ელემენტის ამოგდება `remove(x)` მეთოდის გამოყენებით:

```
>>> b = [2, 3, 5]
>>> print(b)
[2, 3, 5]
>>> b.remove(3)
>>> print(b)
[2, 5]
```

თუ საჭიროა ელემენტის ამოგდება მისი ინდექსის მიხედვით გამოიყენეთ ბრძანება:

```
del სიის_სახელი[ინდექსი]
>>> c = [3, 5, 1, 9, 6]
>>> print(c)
[3, 5, 1, 9, 6]
```

```
>>> del c[2]
>>> print(c)
[3, 5, 9, 6]
```

ელემენტების წაშლა del ბრძანების გამოყენებით:

```
>>> 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]
```

სიის ელემენტის მნიშვნელობის შეცვლა მისი ინდექსის მიხედვით, შეიძლება პირდაპირ მასზე მიმართვით:

```
>>> d = [2, 4, 9]
>>> print(d)
[2, 4, 9]
>>> d[1] = 17
>>> print(d)
[2, 17, 9]
```

სიის გასუფთავება შეიძლება მისი ხელახალი ინიციალიზებით, თითქოს მას ახლიდან ქმნიდეთ. სიის ელემენტზე წვდომის მისაღებად ამ ელემენტის ინდექსი მიუთითეთ კვადრატულ ფრჩხილებში:

```
>>> a = [3, 5, 7, 10, 3, 2, 6, 0]
```

```
>>> a[2]
```

```
7
```

შეიძლება უარყოფითი ინდექსების გამოყენება. ასეთ შემთხვევაში ათვლა დაიწყება ბოლოდან. მაგალითად სიის ბოლო ელემენტზე წვდომისთვის შეიძლება გამოყენებულ იქნას ბრძანება:

```
>>> a[-1]
```

```
0
```

სიიდან ზოგიერთი ქვესიის მისაღებად ინდექსების განსაზღვრულ დიაპაზონში, მიუთითეთ საწყისი და ბოლო ინდექსები კვადრატულ ფრჩხილებში და გამოყავით ისინი ორწერტილით:

```
>>> a[1:4]
```

```
[5, 7, 10]
```

მაგალითში ხდება სიის ბოლოში ელემენტის დამატება. იგივე ოპერაცია შეიძლება შესრულდეს ასეც: $a[\text{len}(a):] = [x]$.

```
>>> a = [1, 2]
```

```
>>> a.append(3)
```

```
>>> print(a)
```

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

აფართოვებს არსებულ სიას b სიიდან ყველა ელემენტის დამატების გზით. ექვივალენტურია ბრძანება: $a[\text{len}(a):] = b$.

```
>>> a = [1, 2]
```

```
>>> b = [3, 4]
```

```
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

x ელემენტს ჩასვამს i პოზიციაში. პირველი არგუმენტი არის ელემენტის ინდექსი, რომლის შემდეგაც ჩაისმება x ელემენტი:

```
>>> a = [1, 2]
>>> a.insert(0, 5)
>>> print(a)
[5, 1, 2]
>>> a.insert(len(a), 9)
>>> print(a)
[5, 1, 2, 9]
```

წაშლის x ელემენტის პირველ შესვლას სიიდან:

```
>>> a = [1, 2, 3]
>>> a.remove(1)
>>> print(a)
[2, 3]
```

წაშლის ელემენტს მითითებული პოზიციიდან და გამოიტანს მას. თუ არგუმენტი არ არის მითითებული წაიშლება სიის ბოლო ელემენტი:

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> print(a.pop(2))
```

```
3
```

```
>>> print(a.pop())
```

```
5
```

```
>>> print(a)
```

```
[1, 2, 4]
```

წაშლის სიიდან ყველა ელემენტს. ექვივალენტურია შემდეგის `del a[:]`:

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> print(a)
```

```
[1, 2, 3, 4, 5]
```

```
>>> a.clear()
```

```
>>> print(a)
```

```
[]
```

აბრუნებს ელემენტის ინდექსს:

```
>>> a = [1, 2, 3, 4, 5]
```

```
>>> a.index(4)
```

```
3
```

აბრუნებს მითითებული ელემენტის შესვლათა რაოდენობას სიაში:

```
>>> a=[1, 2, 2, 3, 3]
```

```
>>> print(a.count(2))
```

```
2
```

ალაგებს სიის ელემენტებს ზრდადობით. უკუმიმდევრობით სორტირებისთვის გამოიყენება ალამი `reverse=True`.

```
>>> a = [1, 4, 2, 8, 1]
```

```
>>> a.sort()
```

```
>>> print(a)
```

```
[1, 1, 2, 4, 8]
```

სიას გადაალაგებს უკუმიმდევრობით:

```
>>> a = [1, 3, 5, 7]
```

```
>>> a.reverse()
```

```
>>> print(a)
```

```
[7, 5, 3, 1]
```

აბრუნებს სიის ასლს. ექვივალენტურია შემდეგის:
`a[:]`.

```
>>> a = [1, 7, 9]
```

```
>>> b = a.copy()
```

```
>>> print(a)
```

```
[1, 7, 9]
```

```
>>> print(b)
```

```
[1, 7, 9]
```

```
>>> b[0] = 8
```

```
>>> print(a)
```

```
[1, 7, 9]
```



```
>>> print(b)
```

```
[8, 7, 9]
```

ქმნის სიას მთელი რიცხვებისგან 0-დან n-მდე, სადაც n უნდა შეიტანოთ წინასწარ:

```
n = int(input())
```

```
a = []
```

```
for i in range(n):
```

```
    a.append(i)
```

```
print(a)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

წინა მოქმედება შეიძლება ჩაიწეროს უფრო მარტივად:

```
n = int(input())
```

```
a = [i for i in range(n)]
```

```
print(a)
```

ან კიდევ უფრო მარტივად, იმ შემთხვევაში თუ თქვენ მეტი აღარ გინდათ n-ის გამოყენება:

```
a = [i for i in range(int(input()))]
```

```
print(a)
```

3.6.2. ინდექსები და ჭრები

მაგალითში მოცემულია სიის ელემენტზე მიმართვა ინდექსის მიხედვით. რადგან 4 ინდექსის მქონე ანუ მე-5 ელემენტი არ არის სიაში, გამოაქვს შეტყობინება ინდექსის შეცდომაზე:

```
>>> a = [1, 3, 8, 7]
```

```
>>> a[0]
1
>>> a[3]
7
>>> a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

მაგალითში სიის ელემენტებზე მიმართვისთვის გამოყენებულია უარყოფითი ინდექსები. ამ შემთხვევაში მიმართვა იწყება ბოლოდან. რადგან სიაში არ არის მე-5 ელემენტი, გამოდის შეტყობინება შეცდომაზე:

```
>>> a = [1, 3, 8, 7]
>>> a[-1]
7
>>> a[-4]
1
>>> a[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Python-ში სიებისთვის გამოიყენება ჭრები. `item[START:STOP:STEP]` - იღებს ჭრას START ნომრიდან STOP-მდე, ბიჯით STEP. დუმილით `START = 0`, `STOP =` ობიექტის სიგრძეს, `STEP = 1`. შესაბამისად, ზოგიერთი

(შესაძლებელია, ყველა) პარამეტრი შეიძლება იყოს გამოტოვებული.

```
>>> a = [1, 3, 8, 7]
>>> a[:]
[1, 3, 8, 7]
>>> a[1:]
[3, 8, 7]
>>> a[:3]
[1, 3, 8]
>>> a[::2]
[1, 8]
```

ასევე, ყველა ეს პარამეტრი შეიძლება იყოს უარყოფითი:

```
>>> a = [1, 3, 8, 7]
>>> a[::-1]
[7, 8, 3, 1]
>>> a[:-2]
[1, 3]
>>> a[-2::-1]
[8, 3, 1]
>>> a[1:4:-1]
[]
```

ბოლო მაგალითში მიღებულია ცარიელი სია, რამდენადაც $START < STOP$, ხოლო $STEP$ უარყოფითია. იგივე მოხდება, თუ მნიშვნელობის დიაპაზონი აღმოჩნდება ობიექტის საზღვრებს გარეთ:

```
>>> a = [1, 3, 8, 7]
```

```
>>> a[10:20]
```

```
[]
```

ჭრების საშუალებით შეიძლება ელემენტის არამართო ამოღება, არამედ დამატება და წაშლა (რასაკვირველია, მხოლოდ ცვლადი მიმდევრობებისთვის).

```
>>> a = [1, 3, 8, 7]
```

```
>>> a[1:3] = [0, 0, 0]
```

```
>>> a
```

```
[1, 0, 0, 0, 7]
```

```
>>> del a[:-3]
```

```
>>> a
```

```
[0, 0, 7]
```

3.6.3. სიების კონსტრუირების დამატებითი საშუალებები

Python ენაში არსებობს სიების კონსტრუირების დამატებითი შესაძლებლობები. სიების ასეთი განსაზღვრა ჩაიწერება კვადრატულ ფრჩხილებში ჩასმული გამოსახულების გამოყენებით და მათ შემდეგ for ბლოკებით:

```
>>> freshfruit = ['banana',  
... 'loganberry',  
... 'passion fruit']
```

```

>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]

```

ელემენტები შეიძლება გაიფილტროს პირობის მითითებით, რომელიც ჩაიწერება if გასაღებური სიტყვის გამოყენებით:

```

>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]

```

გამოსახულება, რომელიც იძლევა კორტეჟს, აუცილებლად უნდა ჩაისვას ფრჩხილებში:

```

>>> [x, x**2 for x in vec]
File "<stdin>", line 1
[x, x**2 for x in vec]
^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]

```

თუ კონსტრუქტორში მითითებულია რამდენიმე for ბლოკი, მეორე მიმდევრობის ელემენტები გადაირჩევიან პირველის ცალკეული ელემენტისთვის და ა.შ. ანუ გადაირჩევა ყველა კომბინაცია:

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

3.7. კორტეჟი

კორტეჟი (tuple) არის შეუცვლელი მონაცემთა სტრუქტურა, რომელიც ძალიან ჰგავს სიას. მისი უპირატესობა არის ის რომ, იგი დაცულია ცვლილებებისგან. ამავდროულად, მცირე ზომისაა.

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> b = [1, 2, 3, 4, 5, 6]
>>> a.__sizeof__()
36
>>> b.__sizeof__()
44
```

კორტეჟთან მუშაობა დაახლოებით იგივეა, რაც სიებთან.

ცარიელი კორტეჟის შექმნა:

```
>>> a = tuple() # tuple() ჩაშენებული ფუნქციის  
დახმარებით
```

```
>>> a
```

```
()
```

```
>>> a = () # კორტეჟის ლიტერალის დახმარებით
```

```
>>> a
```

```
()
```

```
>>>
```

კორტეჟის შექმნა ერთი ელემენტისგან:

```
>>> a = ('s')
```

```
>>> a
```

```
's'
```

მაგრამ აქ მივიღეთ სტრიქონი. ჩვენ გვინდოდა კორტეჟის მიღება, რისთვისაც კოდი შემდეგნაირად უნდა დაიწეროს:

```
>>> a = ('s',)
```

```
>>> a
```

```
('s',)
```

მიღებულ იქნა კორტეჟი. საქმე იყო ‘,’ სიმბოლოში. მხოლოდ ფრჩხილები არაფერს ნიშნავს, უფრო ზუსტად ნიშნავს, რომ მის შიგნით არის ერთი ინსტრუქცია. კორტეჟი შეიძლება შეიქმნას ასეც:

```
>>> a = 's',
```

```
>>> a
```

```
('s',)
```

თუმცა უმჯობესია ფრჩხილების გამოყენება, რადგანაც არის შემთხვევები, როდესაც ფრჩხილები არის აუცილებელი.

კორტეჟის შექმნა იტერირებული ობიექტიდან შეიძლება tuple() ფუნქციის გამოყენებით:

```
>>> a = tuple('hello, world!')
>>> a
('h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!')
```

სიებზე გამოყენებული ყველა ოპერაცია, რომელიც არ ცვლის სიას (შეკრება, რიცხვზე გამრავლება, index() და count() მეთოდები და ზოგიერთი სხვა ოპერაციები) გამოიყენება კორტეჟისთვის. შეიძლება ელემენტების ადგილების შეცვლა.

მაგალითები:

კორტეჟის ელემენტების შეცვლა არ შეიძლება, რაც ჩანს მაგალითში:

```
>>> b = (1, 2, 3)
>>> print(b)
(1, 2, 3)
>>> b[1] = 15
```

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

```
b[1] = 15
```

TypeError: 'tuple' object does not support item assignment

არსებობს რამდენიმე მიზეზი, რომლიდან გამომდინარე ღირს კორტეჟის გამოყენება სიებთან ერთად. ერთ-ერთი მიზეზია მონაცემების დაცვა შემთხვევითი ცვლილებებისგან. ამავედროულად, კორტეჟი მეხსიერებაში იკავებს მცირე მოცულობას, სიებთან შედარებით.

```
>>> lst = [10, 20, 30]
>>> tpl = (10, 20, 30)
>>> print(lst.__sizeof__())
32
>>> print(tpl.__sizeof__())
24
```

კორტეჟი მუშაობს უფრო სწრაფად, ვიდრე სიები. ანუ ელემენტების გადარჩევაზე და სხვა ოპერაციებზე იხარჯება ნაკლები დრო. ასევე აღსანიშნავია, რომ კორტეჟი შეიძლება გამოყენებულ იქნას ლექსიკონთან გასაღების სახით.

ცარიელი კორტეჟის შექმნა შეიძლება ერთ-ერთი ბრძანებით:

```
>>> a = ()
>>> print(type(a))
<class 'tuple'>
>>> b = tuple()
>>> print(type(b))
<class 'tuple'>
```

კორტეჟი მოცემული შემადგენლობით იქმნება ისევე, როგორც სია, მხოლოდ კვადრატული ფრჩხილების ნაცვლად გამოიყენება მრგვალი:

```
>>> a = (1, 2, 3, 4, 5)
```

```
>>> print(type(a))
```

```
<class 'tuple'>
```

```
>>> print(a)
```

```
(1, 2, 3, 4, 5)
```

კორტეჟის შექმნა tuple() ფუნქციით:

```
>>> a = tuple((1, 2, 3, 4))
```

```
>>> print(a)
```

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

კორტეჟის ელემენტებზე წვდომა ხორციელდება ინდექსის მითითებით, სიების მსგავსად. მისი ელემენტის ცვლილების მცდელობისას მიიღება შეცდომა:

```
>>> a = (1, 2, 3, 4, 5)
```

```
>>> print(a[0])
```

```
1
```

```
>>> print(a[1:3])
```

```
(2, 3)
```

```
>>> a[1] = 3
```

```
Traceback (most recent call last):
```

```
File "<pyshell#24>", line 1, in <module>
```

```
a[1] = 3
```

TypeError: 'tuple' object does not support item assignment

ცალკეული ელემენტის წაშლა კორტეჟიდან შეუძლებელია. რაც იძლევა შეცდომას:

```
>>> a = (1, 2, 3, 4, 5)
```

```
>>> del a[0]
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    del a[0]
TypeError: 'tuple' object doesn't support item deletion
```

კორტეჟი მთლიანად შეიძლება წაიშალოს: del a

სიის კორტეჟში გადასაყვანად საკმარისია მისი გადაცემა tuple() ფუნქციის არგუმენტის სახით:

```
>>> lst = [1, 2, 3, 4, 5]
>>> print(type(lst))
<class 'list'>
>>> print(lst)
[1, 2, 3, 4, 5]
>>> tpl = tuple(lst)
>>> print(type(tpl))
<class 'tuple'>
>>> print(tpl)
(1, 2, 3, 4, 5)
```

უკუობერაცია ასევე კორექტულია:

```
>>> tpl = (2, 4, 6, 8, 10)
>>> print(type(tpl))
<class 'tuple'>
>>> print(tpl)
(2, 4, 6, 8, 10)
>>> lst = list(tpl)
>>> print(type(lst))
<class 'list'>
>>> print(lst) #[2, 4, 6, 8, 10]
```

3.8. ლექსიკონები (dictionary)

სიებისა და კორტეჟების გარდა Python-ში არის კიდევ ერთი ჩაშენებული მონაცემთა სტრუქტურა, რომელსაც ჰქვია ლექსიკონი (dictionary).

სიების მსგავსად ლექსიკონი ინახავს ელემენტების კოლექციას. ლექსიკონში ცალკეულ ელემენტს აქვს უნიკალური გასაღები, რომელთან ასოცირებულია ზოგიერთი მნიშვნელობა.

ლექსიკონის განსაზღვრას აქვს შემდეგი სინტაქსი:

```
dictionary={გასაღები1:მნიშვნელობა1,
```

```
გასაღები2:მნიშვნელობა2, ....}
```

განვსაზღვროთ ლექსიკონი:

```
users = {1: "Tom", 2: "Bob", 3: "Bill"}
```

```
elements = {"Au": "ოქრო", "Fe": "რკინა", "H":  
"წყალბადი", "O": "ჟანგბადი"}
```

users ლექსიკონში გასაღების სახით გამოიყენება რიცხვები, ხოლო მნიშვნელობების სახით - სტრიქონები. elements ლექსიკონში გასაღების სახით გამოიყენება სტრიქონები.

არ არის აუცილებელი, რომ გასაღები და სტრიქონი იყოს ერთი ტიპის. ისინი შეიძლება წარმოადგენდნენ სხვადასხვა ტიპებს:

```
objects = {1: "Tom", "2": True, 3: 100.6}
```

შეიძლება განისაზღვროს ცარიელი ლექსიკონი ელემენტების გარეშე:

```
objects = {}
```

ან ასე:

```
objects = dict()
```

3.8.1. სიის გარდაქმნა ლექსიკონად

მიუხედავად იმისა, რომ ლექსიკონი და სია არ არის მსგავსი ტიპების სტრუქტურის მიხედვით, ცალკეული სახის სიისთვის არის ლექსიკონად გარდაქმნის შესაძლებლობა `dict()` ჩაშენებული ფუნქციის დახმარებით. ამისათვის სია უნდა ინახავდეს ჩაშენებული სიების ნაკრებს. ცალკეული ჩაშენებული სია უნდა შედგებოდეს ორი ელემენტისგან - ლექსიკონად კონვერტაციისთვის პირველი ელემენტი გახდება გასაღები, ხოლო მეორე - მნიშვნელობა:

```
users_list = [  
    "+111123455", "Tom"],  
    "+384767557", "Bob"],  
    "+958758767", "Alice"]  
]  
users_dict = dict(users_list)  
print(users_dict) # {"+111123455": "Tom", "+384767557":  
"Bob", "+958758767": "Alice"}  
შედეგი:  
{'+111123455': 'Tom', '+384767557': 'Bob', '+958758767':  
'Alice'}
```

მსგავსად, შეიძლება ლექსიკონად გარდაიქმნას ორგანზომილებიანი კორტეჟი ანუ კორტეჟი, რომელიც თავის მხრივ შეიცავს ორ ელემენტთან კორტეჟებს:

```
users_tuple = (  
    "+111123455", "Tom"),  
    ("+384767557", "Bob"),  
    ("+958758767", "Alice")  
)  
users_dict = dict(users_tuple)  
print(users_dict)
```

3.8.2. ელემენტის მიღება, შეცვლა

ლექსიკონის ელემენტებზე წვდომისთვის საჭიროა გასაღების გამოყენება:

```
dictionary[key]
```

მაგალითად, მივიღოთ და შევცვალოთ ელემენტი ლექსიკონში:

```
users = {  
    "+11111111": "Tom",  
    "+33333333": "Bob",  
    "+55555555": "Alice"}  
# ვიღებთ ელემენტს გასაღებით "+11111111"  
print(users["+11111111"]) # Tom  
# ელემენტის მნიშვნელობის დაყენება გასაღებით  
"+33333333"
```

```
users["+33333333"] = "Bob Smith"  
print(users["+33333333"]) # Bob Smith
```

თუ ასეთი გასაღებით ელემენტის მნიშვნელობა არ აღმოჩნდება, მოხდება მისი დამატება:

```
users["+44444444"] = "Sam"
```

მაგრამ თუ ჩვენ ვეცდებით მნიშვნელობის მიღებას გასაღებით, რომელიც არ არის ლექსიკონში, Python გამოიტანს შეცდომას:

```
user = users["+44444444"] # KeyError
```

მსგავსი სიტუაციის წინასწარ განსაზღვრის მიზნით, ელემენტზე მიმართვამდე შეიძლება შემოწმდეს ლექსიკონში გასაღების არსებობა გამოსახულებით **key in dictionary**. თუ გასაღები არის ლექსიკონში, მაშინ მოცემული გამოსახულება აბრუნებს True მნიშვნელობას:

```
key = "+44444444"
```

```
if key in users:
```

```
    user = users[key]
```

```
    print(user)
```

```
else:
```

```
    print("ელემენტი არ არის ნაპოვნი")
```

ელემენტის მისაღებად შეიძლება get მეთოდის გამოყენება, რომელსაც აქვს ორი ფორმა:

- **get(key)** - აბრუნებს ლექსიკონიდან ელემენტს key გასაღებით. თუ ელემენტი ასეთი გასაღებით არ არის, მაშინ აბრუნებს None მნიშვნელობას.

- **get(key, default)** - აბრუნებს ლექსიკონიდან ელემენტს key გასაღებით. თუ ელემენტი ასეთი გასაღებით არ არის, მაშინ დააბრუნებს მნიშვნელობას დუმილით (default)
key = "+55555555"
user = users.get(key)
user = users.get(key, "Unknown user")

3.8.3. ელემენტის წაშლა

გასაღების მიხედვით ელემენტის წასაშლელად, გამოიყენება del ოპერატორი:

```
users = {  
    "+11111111": "Tom",  
    "+33333333": "Bob",  
    "+55555555": "Alice"  
}  
del users["+55555555"]  
print(users)
```

უნდა გაითვალისწინოთ, რომ თუ მსგავსი გასაღები არ აღმოჩნდება ლექსიკონში, გამოვა შეტყობინება შეცდომაზე. ამიტომ წაშლის წინ სასურველია შემოწმდეს მოცემული გასაღებით ელემენტის არსებობა.

```
key = "+55555555"  
if key in users:  
    user = users[key]  
    del users[key]  
    print(user, "წაშლილია")
```


else:

```
print("ელემენტი არ არის ნაპოვნი")
```

წაშლის სხვა ხერხია pop() მეთოდი. მას აქვს ორი ფორმა:

- **pop(key)** - წაშლის ელემენტს key გასაღების მიხედვით და დააბრუნებს წაშლილ ელემენტს. თუ ელემენტი ასეთი გასაღებით არ არის, გენერირდება შეცდომა;
- **pop(key, default)** - წაშლის ელემენტს key გასაღებით და აბრუნებს წაშლილ ელემენტს. თუ ელემენტი ასეთი გასაღებით არ არის, მაშინ დააბრუნებს default მნიშვნელობას.

```
users = {  
    "+11111111": "Tom",  
    "+33333333": "Bob",  
    "+55555555": "Alice"  
}  
key = "+55555555"  
user = users.pop(key)  
print(user)  
user = users.pop("+44444444", "Unknown user")  
print(user)
```

თუ საჭიროა ყველა ელემენტის წაშლა, მაშინ შეიძლება **clear()** მეთოდის გამოყენება:

```
users.clear()
```

3.8.4. ლექსიკონის კოპირება და გაერთიანება

`copy()` მეთოდი აკოპირებს ლექსიკონის შემადგენლობას და აბრუნებს ახალ ლექსიკონს:

```
users={"+1111111": "Tom", "+3333333": "Bob", "+5555555":  
"Alice"}
```

```
users2 = users.copy()
```

`update()` მეთოდი აერთიანებს ორ ლექსიკონს:

```
users = {"+1111111": "Tom", "+3333333": "Bob", "+5555555":  
"Alice"}
```

```
users2 = {"+2222222": "Sam", "+6666666": "Kate"}
```

```
users.update(users2)
```

```
print(users) # {"+1111111": "Tom", "+3333333": "Bob",  
"+5555555": "Alice", "+2222222": "Sam", "+6666666": "Kate"}
```

```
print(users2) # {"+2222222": "Sam", "+6666666": "Kate"}
```

ამ დროს ლექსიკონი `users2` რჩება უცვლელად. იცვლება `users` ლექსიკონი, რომელშიც ემატება მეორე ლექსიკონიდან. თუ საჭიროა, რომ ორივე საწყისი ლექსიკონი იყოს უცვლელად, ხოლო გაერთიანების შედეგი იყოს მესამე ლექსიკონი, მაშინ შეიძლება წინასწარ დაკოპირდეს ერთი ლექსიკონი მეორეში:

```
users3 = users.copy()
```

```
users3.update(users2)
```

3.8.5. ლექსიკონის გადარჩევა

ლექსიკონის გადარჩევისთვის შეიძლება for ციკლის გამოყენება:

```
users = {  
    "+11111111": "Tom",  
    "+33333333": "Bob",  
    "+55555555": "Alice"  
}
```

```
for key in users:
```

```
    print(key, " - ", users[key])
```

ელემენტების გადარჩევისას ვიღებთ მიმდინარე ელემენტის გასაღებს, რომლის მიხედვით შეიძლება თვით ელემენტის მიღება.

ელემენტების გადარჩევის სხვა ხერხია items() მეთოდის გამოყენება:

```
for key, value in users.items():
```

```
    print(key, " - ", value)
```

items() მეთოდი აბრუნებს კორტეჟების ნაკრებს, ცალკეული კორტეჟი შეიცავს გასაღებს და ელემენტის მნიშვნელობას, რომელიც გადარჩევისას იქვე შეიძლება მივიღოთ key და value ცვლადებში.

არსებობს ცალკე გასაღების და ცალკე მნიშვნელობის გადარჩევის შესაძლებლობა keys() მეთოდის გამოყენებით:

```
for key in users.keys():
```

```
    print(key)
```

თუმცა გადარჩევის ამ მეთოდს აზრი არა აქვს, რადგან keys() მეთოდის გამოძახების გარეშეც შეიძლება გასაღების გადარჩევა, როგორც ზემოთ იყო ნაჩვენები.

მხოლოდ მნიშვნელობის გადასარჩევად შეიძლება გამოყენებულ იქნას values() მეთოდი:

```
for value in users.values():  
    print(value)
```

3.8.6. კომპლექსური ლექსიკონები

რიცხვითი და სტრიქონის ტიპის მარტივი ობიექტების გარდა ლექსიკონი შეიძლება ინახავდეს უფრო რთულ ობიექტებს - იგივე სიებს, კორტეჟებს და სხვა ლექსიკონებს:

```
users = {  
    "Tom": {  
        "phone": "+971478745",  
        "email": "tom12@gmail.com"  
    },  
    "Bob": {  
        "phone": "+876390444",  
        "email": "bob@gmail.com",  
        "skype": "bob123"  
    }  
}
```

მოცემულ მაგალითში ლექსიკონის ცალკეული ელემენტის მნიშვნელობა თავის მხრივ წარმოადგენს

ცალკეულ ლექსიკონს. ჩაშენებული ლექსიკონის ელემენტზე მიმართვისთვის შესაბამისად საჭიროა ორი გასაღების გამოყენება:

```
old_email = users["Tom"]["email"]
users["Tom"]["email"] = tom12@gmail.com
```

თუ შეეცდებით მნიშვნელობის მიღებას გასაღების მიხედვით, რომელიც არ არის ლექსიკონში Python აგენერირებს KeyError-ს:

```
tom_skype = users["Tom"]["skype"] # KeyError
```

შეცდომის თავიდან ასაცილებლად, უნდა შემოწმდეს გასაღების არსებობა ლექსიკონში:

```
key = "skype"
if key in users["Tom"]:
    print(users["Tom"]["skype"])
else:
    print("skype is not found")
```

მაგალითები:

ლექსიკონის შექმნა ლიტერალის დახმარებით:

```
>>> d = {}
>>> d
{}
>>> d = {'dict': 1, 'dictionary': 2}
>>> d
```

```
{'dict': 1, 'dictionary': 2}
```

ლექსიკონის შექმნა dict ფუნქციის დახმარებით:

```
>>> d = dict(short='dict', long='dictionary')
```

```
>>> d
```

```
{'short': 'dict', 'long': 'dictionary'}
```

```
>>> d = dict([(1, 1), (2, 4)])
```

```
>>> d
```

```
{1: 1, 2: 4}
```

ლექსიკონის შექმნა `fromkeys` მეთოდის დახმარებით:

```
>>> d = dict.fromkeys(['a', 'b'])
```

```
>>> d
```

```
{'a': None, 'b': None}
```

```
>>> d = dict.fromkeys(['a', 'b'], 100)
```

```
>>> d
```

```
{'a': 100, 'b': 100}
```

ლექსიკონის შექმნა ლექსიკონის გენერატორების დახმარებით, რომლებიც ძალიან ჰგვანან სიების გენერატორებს:

```
>>> d = {a: a ** 2 for a in range(7)}
```

```
>>> d
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

ლექსიკონში ჩანაწერის დამატება და გასაღების მნიშვნელობის ამოღება:

```
>>> d = {1: 2, 2: 4, 3: 9}
```

```
>>> d[1]
```

```
2
```

```
>>> d[4] = 4 ** 2
```

```
>>> d
```

```
{1: 2, 2: 4, 3: 9, 4: 16}
```

```
>>> d['1']
```

```
Traceback (most recent call last):
```

```
File "", line 1, in
```

```
d['1']
```

```
KeyError: '1'
```

როგორც მაგალითიდან ჩანს, ახალ გასაღებზე მინიჭება აფართოვებს ლექსიკონს, არსებულ გასაღებზე მინიჭება მას გადააწერს მნიშვნელობას, ხოლო არარსებული გასაღების ამოღება იძლევა შეცდომას.

3.8.7. მეთოდები ლექსიკონებთან სამუშაოდ

dict.clear() - ასუფთავებს ლექსიკონს;

dict.copy() - აბრუნებს ლექსიკონის ასლს;

classmethod dict.fromkeys(seq[, value]) - ქმნის ლექსიკონს გასაღებით seq და მნიშვნელობით value (დუმილით None).

dict.get(key[, default]) - აბრუნებს გასაღების მნიშვნელობას, მაგრამ იგი თუ არ არის, აბრუნებს default მნიშვნელობას (დუმილით None);

dict.items() - აბრუნებს წყვილს - გასაღები, მნიშვნელობა;

dict.keys() - აბრუნებს გასაღებებს ლექსიკონიდან;

dict.pop(key[, default]) - წაშლის გასაღებს და აბრუნებს მნიშვნელობას. თუ გასაღები არ არის აბრუნებს default (დუმილით აგენერირებს გამონაკლისს KeyError);

dict.popitem() - წაშლის და აბრუნებს წყვილს: გასაღები, მნიშვნელობა. თუ ლექსიკონი ცარიელია აგენერირებს გამონაკლისს KeyError. ლექსიკონი არ არის დალაგებული;

dict.setdefault(key[, default]) - აბრუნებს გასაღების მნიშვნელობას. თუ იგი არ არის, არ აგენერირებს გამონაკლისს, არამედ ქმნის გასაღებს default მნიშვნელობით. (დუმილით None);

dict.update([other]) - განახლებს ლექსიკონს: გასაღები, მნიშვნელობა წყვილის დამატებით other-დან. არსებული გასაღებები გადაიწერება. აბრუნებს None (არა ახალი ლექსიკონი!);

dict.values() - აბრუნებს მნიშვნელობას ლექსიკონიდან.

IV თავი. სიმრავლე

4.1. სიმრავლის განსაზღვრა

სიმრავლე (set) წარმოადგენს ელემენტების ნაკრების კიდევ ერთ სახეობას. სიმრავლის განსაზღვრისთვის გამოიყენება ფიგურული ფრჩხილები, რომელშიც ჩამოითვლება ელემენტები:

```
users = {"Tom","Bob","Alice", "Tom"}  
print(users) # {"Tom","Bob","Alice"}
```

მიაქციეთ ყურადღება, რომ print ფუნქციამ გამოიტანა "Tom" ელემენტი ერთხელ, თუნდაც სიმრავლის განსაზღვრაში ეს ელემენტი ორჯერ მონაწილეობს, რადგანაც სიმრავლე შეიცავს მხოლოდ უნიკალურ მნიშვნელობებს.

სიმრავლის განსაზღვრისთვის შეიძლება გამოყენებულ იქნას set() ფუნქცია, რომელსაც გადაეცემა ელემენტების სია ან კორტეჟი:

```
users3 = set(["Mike", "Bill", "Ted"])
```

set ფუნქციის გამოყენება მოხერხებულია ცარიელი სიმრავლის შესაქმნელად:

```
users = set()
```

სიმრავლის სიგრძის განსაზღვრისთვის გამოიყენება len() ჩაშენებული ფუნქცია:

```
users = {"Tom","Bob","Alice"}  
print(len(users)) # 3
```

4.2. ელემენტების დამატება

თითო ელემენტის დასამატებლად გამოიყენება `add()` მეთოდი:

```
users = set()
users.add("Sam")
print(users)
```

4.3. ელემენტების წაშლა

ერთი ელემენტის წასაშლელად გამოიყენება `remove()` მეთოდი, რომელშიც გადაეცემა წასაშლელი ელემენტი. მაგრამ უნდა იქნას გათვალისწინებული, რომ თუ ასეთი ელემენტი არ არის სიმრავლეში, გენერირდება შეცდომა. ამიტომ წაშლამდე `in` ოპერატორით უნდა შემოწმდეს არის თუ არა ელემენტი სიმრავლეში:

```
users = {"Tom", "Bob", "Alice"}
user = "Tom"
if user in users:
    users.remove(user)
```

```
print(users)
```

შედეგი:

```
{'Bob', 'Alice'}
```

ასევე, წასაშლელად შეიძლება გამოყენებულ იქნას `discard()` მეთოდი, რომელიც არ აგენერირებს შეცდომას ელემენტის არ არსებობის დროს:

```
user = "Tim"
users.discard(user)
```

ყველა ელემენტის წასაშლელად გამოიყენება `clear()` მეთოდი:

```
users.clear()
```

4.4. სიმრავლის გადარჩევა

ელემენტების გადასარჩევად შეიძლება `for` ციკლის გამოყენება:

```
users = {"Tom", "Bob", "Alice"}
for user in users:
    print(user)
```

გადარჩევისას ცალკეული ელემენტი განთავსდება `user` ცვლადში.

4.5. ოპერაციები სიმრავლეებზე

`copy()` მეთოდის დახმარებით შეიძლება დაკოპირდეს ერთი სიმრავლის შემადგენლობა სხვა ცვლადში:

```
users = {"Tom", "Bob", "Alice"}
users3 = users.copy()
```

გამოვიტანოთ ორივე სიმრავლე. დავრწმუნდებით, რომ `users` და `users3` შეიცავს ერთიდაიგივე ელემენტებს:

```
print (users)
print (users3)
```

შედეგი:

```
{'Alice', 'Tom', 'Bob'}
{'Alice', 'Tom', 'Bob'}
```

union() მეთოდი აერთიანებს ორ სიმრავლეს და აბრუნებს ახალ სიმრავლეს:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.union(users2)
print(users3)
```

შედეგი:

```
{'Kate', 'Sam', 'Tom', 'Bob', 'Alice'}
```

სიმრავლეთა თანაკვეთა საშუალებას იძლევა მიღებულ იქნას მხოლოდ ის ელემენტები, რომლებიც ერთდროულად არიან ორივე სიმრავლეში. **intersection()** მეთოდი ახდენს სიმრავლეთა თანაკვეთის ოპერაციას და აბრუნებს ახალ სიმრავლეს:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.intersection(users2)
print(users3)
```

შედეგი:

```
{'Bob'}
```

intersection მეთოდის ნაცვლად შეიძლება გამოყენებულ იქნას ლოგიკური გამრავლების ოპერაცია:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
print(users & users2)
```

მიიღება იგივე შედეგი:

```
{'Bob'}
```

სიმრავლის სხვაობა აბრუნებს ელემენტებს, რომელიც არის პირველ სიმრავლეში და არ არის მეორეში. სიმრავლეთა სხვაობის მისაღებად შეიძლება გამოყენებულ იქნას difference მეთოდი ან გამოკლების ოპერაცია:

```
users = {"Tom", "Bob", "Alice"}
users2 = {"Sam", "Kate", "Bob"}
users3 = users.difference(users2)
print(users3)      # {"Tom", "Alice"}
print(users - users2) # {"Tom", "Alice"}
```

4.6. სიმრავლეთა შორის დამოკიდებულება

issubset მეთოდი დაადგენს არის თუ არა მიმდინარე სიმრავლე სხვა სიმრავლის ქვესიმრავლე:

```
users = {"Tom", "Bob", "Alice"}
superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}
print(users.issubset(superusers)) # True
print(superusers.issubset(users)) # False
```

issuperset მეთოდი, პირიქით, აბრუნებს True, თუ მიმდინარე სიმრავლე შეიცავს ქვესიმრავლეს:

```
users = {"Tom", "Bob", "Alice"}
superusers = {"Sam", "Tom", "Bob", "Alice", "Greg"}
print(users.issuperset(superusers)) # False
print(superusers.issuperset(users)) # True
```

4.7. ფუნქციები და მეთოდები სიმრავლებთან სამუშაოდ

`len(s)`- სიმრავლეში ელემენტების რაოდენობა;

`x in s` - მიკუთვნება სიმრავლეზე;

`set.isdisjoint(other)` - ჭეშმარიტია, თუ `set` და `other` არა აქვთ საერთო ელემენტი;

`set == other` – `set` ყველა ელემენტი ეკუთვნის `other` და პირიქით;

`set.issubset(other)` ანუ `set <= other` - `set` ყველა ელემენტი ეკუთვნის `other`;

`set.issuperset(other)` ან `set >= other` – `other` ყველა ელემენტი ეკუთვნის `set`;

`set.union(other, ...)` ან `set | other |` - ყველა სიმრავლის გაერთიანება;

`set.intersection(other, ...)` ან `set & other & ...`-სიმრავლეთა თანაკვეთა;

`set.difference(other, ...)` ან `set - other` - აბრუნებს სიმრავლეს ელემენტებისგან, რომლებიც არის `set`-ში და არ არის `other`-ში;

`set.symmetric_difference(other)`; `set ^ other` - აბრუნებს სიმრავლეს ელემენტებისგან, რომლებიც გვხვდება ერთ სიმრავლეში და არ გვხვდება ორივე სიმრავლეში.

`set.copy()` - სიმრავლის ასლი.

`set.update(other, ...)`; `set |= other |` - სიმრავლეთა გაერთიანება;

`set.intersection_update(other, ...)`; `set &= other & ...`-
 სიმრავლეთა თანაკვეთა;
`set.difference_update(other, ...)`; `set -= other | ..`- სიმრავლეთა
 გამოკლება;
`set.symmetric_difference_update(other)`; `set ^= other` -
 მიიღება სიმრავლე ელემენტებისგან, რომელიც გვხვდება
 ერთ-ერთ სიმრავლეში, მაგრამ არ გვხვდება ორივეში;
`set.add(elem)` - დაამატებს ელემენტს სიმრავლეში;
`set.remove(elem)` - წაშლის ელემენტს სიმრავლედან;
`set.discard(elem)` - წაშლის ელემენტს. თუ იგი მდებარეობს
 სიმრავლეში;
`set.pop()` - წაშლის სიმრავლედან პირველ ელემენტს,
 რამდენადაც სიმრავლე არ არის მოწესრიგებული, არ
 შეიძლება ითქვას თუ რომელი ელემენტი იქნება პირველი;
`set.clear()` - სიმრავლის გასუფთავება.

4.8. frozen set ტიპი

frozen set ტიპი არის სიმრავლის ტიპი, რომელიც არ
 შეიძლება შეიცვალოს. მის შესაქმნელად გამოიყენება
`frozenset` ფუნქცია:

```
users = frozenset({"Tom", "Bob", "Alice"})
```

`frozenset` ფუნქციას გადაეცემა ელემენტების ნაკრები
 - სია, კორტეჟი, სხვა სიმრავლე.

ასეთ სიმრავლეში ვერ დავამატებთ ახალ
 ელემენტებს და ვერ წავშლით უკვე არსებულს. ამდენად
`frozen set` მხარს უჭერს ოპერაციათა შეზღუდულ ნაკრებს:

- len(s) - აბრუნებს სიმრავლის სიგრძეს;
- x in s - აბრუნებს True, თუ x ელემენტი არის s სიმრავლეში;
- x not in s - აბრუნებს True, თუ x ელემენტი არ არის s სიმრავლეში;
- s.issubset(t) - აბრუნებს True, თუ t შეიცავს s სიმრავლეს;
- s.issuperset(t) - აბრუნებს True, თუ t შედის s სიმრავლეში;
- s.union(t) - აბრუნებს s და t სიმრავლეთა გაერთიანებას;
- s.intersection(t) - აბრუნებს s და t სიმრავლეთა გადაკვეთას;
- s.difference(t) - აბრუნებს s და t სიმრავლეთა სხვაობას;
- s.copy() - აბრუნებს s სიმრავლის ასლს.

მაგალითები:

სიმრავლის შექმნა:

```
>>> a = set([1,2,3,4,5])
```

```
>>> a
```

```
set([1, 2, 3, 4, 5]) # {1, 2, 3, 4, 5}
```

შეიძლება სიმრავლის ჩაწერა გამარტივებული ფორმით:

```
>>> b = {1,3,5}
```

```
>>> b
```

```
set([1, 3, 5]) # {1, 3, 5}
```

როგორც მაგალითიდან ჩანს, სიმრავლეს აქვს იგივე ლიტერალი, როგორც ლექსიკონს, მაგრამ ცარიელი

სიმრავლე ლიტერალის დახმარებით არ შეიძლება შეიქმნას:

```
>>> a = set()
>>> a
set()
>>> a = set('hello')
>>> a
{'h', 'o', 'l', 'e'}
>>> a = {'a', 'b', 'c', 'd'}
>>> a
{'b', 'c', 'a', 'd'}
>>> a = {i ** 2 for i in range(10)} #სიმრავლეთა
გენერატორი
>>> a
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
>>> a = {} # ეს არ შეიძლება!
>>> type(a)
<class 'dict'>
```

სიმრავლე ვერ შეინახავს ორ ერთნაირ ელემენტს:

```
>>> set([1,1,1,2,2,2,3,3,3,4])
set([1, 2, 3, 4])
>>> set("hello")
set(['h', 'e', 'l', 'o'])
```

მაგალითში მოცემულია სიმრავლის გადაცემა for ციკლში:

```
a = set([1,2,3,4,5])
```

```
for i in a:
```

```
    print (i)
```

შედეგი:

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

სიმრავლე გამოყენებულია გამოორებადი ელემენტების წასაშლელად:

```
>>> words = ['hello', 'daddy', 'hello', 'mum']
```

```
>>> set(words)
```

```
{'hello', 'daddy', 'mum'}
```

frozenset გამოყენების მაგალითი:

```
>>> a = set('qwerty')
```

```
>>> b = frozenset('qwerty')
```

```
>>> a == b
```

```
True
```

```
>>> True
```

```
True
```

```
>>> type(a - b)
```

```
<class 'set'>
```

```
>>> type(a | b)
```

```
<class 'set'>
```

```
>>> a.add(1)
```

```
>>> b.add(1)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'frozenset' object has no attribute 'add'

მოცემულ მაგალითში სია გარდავექმნით სიმრავლედ, ხოლო შემდგომ სიმრავლე გარდავექმნით სიად. რის შედეგადაც სიმრავლე გათავისუფლდება დუბლირებული ელემენტებისგან:

```
>>> lst = ["a","c","b","c","c","d","g","d","e","a","g","f","b"]
>>> lst
['a', 'c', 'b', 'c', 'c', 'd', 'g', 'd', 'e', 'a', 'g', 'f', 'b']
>>> newlst = list(set(lst))
>>> newlst
['a', 'c', 'b', 'e', 'd', 'g', 'f']
```

a და b სიმრავლეები გაერთიანდება:

```
>>> a = [1,2,4,5,7]
>>> b = [2,3,4,5,6,7,8,9]
>>> a
[1, 2, 4, 5, 7]
>>> b
[2, 3, 4, 5, 6, 7, 8, 9]
>>> set(a)|set(b)
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

სიმრავლეთა გამოკლება:

```
>>> a
[1, 2, 4, 5, 7]
>>> b
[2, 3, 4, 5, 6, 7, 8, 9]
>>> set(a)-set(b)
```

```
set([1])
```

```
>>> set(b)-set(a)
```

```
set([8, 9, 3, 6])
```

სიმრავლეთა თანაკვეთა:

```
>>> set(a)&set(b)
```

```
set([2, 4, 5, 7])
```

სიმრავლეთა სიმეტრიული სხვაობა. აბრუნებს ელემენტებს, რომლებიც არ იკვეთებიან გადაცემულ სიმრავლეებში:

```
>>> set(a)^set(b)
```

```
set([1, 3, 6, 8, 9])
```

მაგალითში მოწმდება ერთი სიმრავლის ელემენტი შედის თუ არა მეორეში, ფაქტობრივად ერთი სიმრავლე არის თუ არა მეორეს ქვესიმრავლე:

```
>>>a=set([1,2,3,4,5,6,7,8,9])#1-დან 9-მდე ნატურალური რიცხვების სიმრავლე
```

```
>>> b = set([3,4,5,6,7]) # 3-დან 7-მდე ნატურალური რიცხვების სიმრავლე
```

```
>>> c = set([3,5,7]) # 3, 5 და 7 რიცხვების სიმრავლე
```

```
>>> a
```

```
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
>>> b
```

```
set([3, 4, 5, 6, 7])
```

```
>>> c
```

```
set([3, 5, 7])
```

```
>>> e = set([3,5,7,9,0])
```

```
>>> d=set([9, 3, 5, 7])
```

```

>>> e
set([0, 9, 3, 5, 7])
>>> a
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> b
set([3, 4, 5, 6, 7])
>>> a>b
True
>>> b>a # "a" არ არის "b" -ს ქვესიმრავლე
False
>>> b
set([3, 4, 5, 6, 7])
>>> c
set([3, 5, 7])
>>> c>b
False
>>> b>c # "c" არის "b"-ს ქვესიმრავლე
True
>>> a>c # "c" არის "a"-ს ქვესიმრავლე
True
>>> a>b>c # "c" არის "b" და "a"-ს ქვესიმრავლე, "b" არის
"a"-ს ქვესიმრავლე
True
>>> a
set([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> c
set([3, 5, 7])
>>> d
set([9, 3, 5, 7])
>>> a>d>c #True

```

V თავი. ფუნქციები

5.1. ფუნქციის განსაზღვრა

ფუნქცია არის კოდის ბლოკი, რომელიც ასრულებს განსაზღვრულ ამოცანას და რომელიც შეიძლება განმეორებით იყოს გამოყენებული პროგრამის სხვა ნაწილებში. ფუნქცია მიიღებს არგუმენტებს და აბრუნებს მნიშვნელობას. ფუნქცია განისაზღვრება def ინსტრუქციის დახმარებით. ფუნქციის განსაზღვრის სინტაქსი:

```
def ფუნქციის_სახელი ([პარამეტრები]):  
    ინსტრუქციები
```

ფუნქციის განსაზღვრა იწყება def გამოსახულებით, რომელიც შედგება ფუნქციის სახელისგან, ფრჩხილების ნაკრებისგან პარამეტრებით და ორწერტილისგან. პარამეტრები ფრჩხილებში არააუცილებელია. შემდეგი სტრიქონიდან იწერება ინსტრუქციების ბლოკი, რომელსაც ასრულებს ფუნქცია. ფუნქციის ყველა ინსტრუქციას აქვს შეწევა სტრიქონის დასაწყისიდან.

მარტივი ფუნქციის განსაზღვრა:

```
def say_hello():  
    print("Hello")
```

ფუნქცია გამოიძახება say_hello-თი. მას არა აქვს პარამეტრები და შეიცავს ერთადერთ ინსტრუქციას, რომელსაც გამოაქვს კონსოლზე სტრიქონი Hello.

ფუნქციის გამოსამახებლად უნდა მიეთითოს ფუნქციის სახელი, რომლის შემდეგაც ფრჩხილებში მიეთითება გადასაცემი მნიშვნელობები. მაგალითად:

```
def say_hello():  
    print("Hello")  
say_hello()  
say_hello()  
say_hello()
```

სამჯერ ხდება ფუნქციის გამოძახება. კონსოლზე მიიღება შედეგი:

```
Hello  
Hello  
Hello
```

განვსაზღვროთ პარამეტრებიანი ფუნქცია:

```
def say_hello(name):  
    print("Hello,",name)  
say_hello("Tom")  
say_hello("Bob")  
say_hello("Alice")
```

ფუნქცია მიიღებს name პარამეტრს და ფუნქციის გამოძახებისას პარამეტრის ნაცვლად შეგვიძლია გადავცეთ რაიმე მნიშვნელობა:

```
Hello, Tom  
Hello, Bob  
Hello, Alice
```

ფუნქციის ზოგიერთი პარამეტრი შეიძლება გავხადოთ არააუცილებელი, რისთვისაც ფუნქციის

განსაზღვრისას მათთვის უნდა მიეთითოს მნიშვნელობები დუმილით. მაგალითად:

```
def say_hello(name="Tom"):
    print("Hello,", name)
say_hello()
say_hello("Bob")
```

აქ name პარამეტრი არააუცილებელია. თუ ფუნქციის გამოძახებისას არ გადავცემთ პარამეტრს, მაშინ მიიღებს მნიშვნელობას დუმილით, ანუ სტრიქონს "Tom".

მნიშვნელობათა გადაცემის დროს ფუნქცია შეუსაბამებს მათ პარამეტრებს იმავე მიმდევრობით, რომლითაც ისინი გადაეცემიან. მაგალითად:

```
def display_info(name, age):
    print("Name:", name, "\t", "Age:", age)
display_info("Tom", 22)
```

ფუნქციის გამოძახებისას პირველი მნიშვნელობა Tom გადაეცემა პირველ პარამეტრს - name, მეორე მნიშვნელობა -22 გადაეცემა მეორე პარამეტრს - age და ასე შემდეგ რიგის მიხედვით. სახელობითი პარამეტრის გამოყენება საშუალებას იძლევა შეიცვალოს გადაცემის მიმდევრობა:

```
def display_info(name, age):
    print("Name:", name, "\t", "Age:", age)
display_info(age=22, name="Tom")
```

* სიმბოლოს გამოყენებით შეიძლება მიეთითოს პარამეტრების განუსაზღვრელი რაოდენობა:

```
def sum(*params):
    result = 0
```



```

for n in params:
    result += n
return result

sumOfNumbers1 = sum(1, 2, 3, 4, 5)    # 15
sumOfNumbers2 = sum(3, 4, 5, 6)     # 18
print(sumOfNumbers1)
print(sumOfNumbers2)

```

როგორც ვხედავთ sum ფუნქციას ერთ შემთხვევაში გადაეცემა 5 პარამეტრი, მეორეში - 4 პარამეტრი.

ფუნქციამ შეიძლება დააბრუნოს მნიშვნელობა, რისთვისაც გამოიყენება return ოპერატორი, რომლის შემდეგაც მიეთითება დასაბრუნებელი მნიშვნელობა, მაგალითად, ფუნქცია აბრუნებს x და y რიცხვების ჯამს:

```

def add(x, y):
    return x + y

```

ეს ფუნქცია შეიძლება გამოძახებულ იქნას შემდეგი სახით:

```

>>> add(1, 10)
11
>>> add('abc', 'def')
'abcdef'

```

მაგალითი:

```

def exchange(usd_rate, money):
    result = round(money/usd_rate, 2)
    return result

result1 = exchange(60, 30000)

```

```
print(result1)
result2 = exchange(56, 30000)
print(result2)
result3 = exchange(65, 30000)
print(result3)
```

რამდენადაც ფუნქცია აბრუნებს მნიშვნელობას, შეგვიძლია ეს მნიშვნელობა მივანიჭოთ რაიმე ცვლადს და შემდეგ გამოვიყენოთ იგი: `result2 = exchange(56, 30000)`.

Python-ში ფუნქციას შეუძლია დააბრუნოს ერთდროულად რამდენიმე მნიშვნელობა:

```
def create_default_user():
    name = "Tom"
    age = 33
    return name, age

user_name, user_age = create_default_user()
print("Name:", user_name, "\t Age:", user_age)
```

მოცემულ მაგალითში `create_default_user` ფუნქცია აბრუნებს ორ მნიშვნელობას: `name` და `age`. ფუნქციის გამოძახებისას ეს მნიშვნელობები რიგის მიხედვით ენიჭება ცვლადებს: `user_name` და `user_age`, რის შემდეგაც შეიძლება მათი გამოყენება.

ფუნქცია შეიძლება იყოს ნებისმიერი სირთულის და დააბრუნოს ნებისმიერი ობიექტი: სიები, კორტეჟი და თვით ფუნქციები:

```
>>> def newfunc(n):
...     def myfunc(x):
```

```

...     return x + n
...     return myfunc
...
>>> new = newfunc(100) # new - ეს ფუნქციაა
>>> new(200)
300

```

ფუნქცია შეიძლება არ მთავრდებოდეს return ინსტრუქციით, ამ დროს ფუნქცია დააბრუნებს None მნიშვნელობას:

```

>>> def func():
...     pass
...
>>> print(func())
None

```

5.2. ფუნქციები: zip(), map(), lambda(), filter(), reduce()

map(), zip() და lambda() ფუნქციები საშუალებას იძლევიან საკმაოდ მარტივად შესრულდეს სხვადასხვა მანიპულაციები მონაცემებთან.

ვთქვათ, მოცემულია ერთნაირი სიგრძის ორი სია: a = [1, 2] და b = [3, 4] და საჭიროა მათი შერწყმა წყვილებად. ამ შემთხვევაში ყველაზე მარტივია zip ფუნქციის გამოყენება:

```

a = [1,2]
b = [3,4]

```

```
print (zip(a,b))  
[(1, 3), (2, 4)]
```

ანალოგიურად გამოიყენება zip ფუნქცია სამი წყვილის მისაღებად:

```
a = [1,2]  
b = [3,4]  
c = [5,6]  
print (zip(a,b,c))  
[(1, 3, 5), (2, 4, 6)]
```

ზოგადი ფორმით:

```
list = [a, b, c]  
print (zip(*list))  
[(1, 3, 5), (2, 4, 6)]
```

ნიშანი list წინ მიუთითებს, რომ გადაეცემა არგუმენტების სია ანუ იგივეა თითქოს გადაცემული იყოს a, b, c ანუ შეიძლება ასეც: print zip([a, b, c]), შედეგი არ შეიცვლება.

არის სიტუაციები, როდესაც რომელიმე ფუნქციის გამოყენება საჭიროა სიის ცალკეული ელემენტისთვის. მოცემულ შემთხვევაში კოდი შეიძლება ასეც დაიწეროს:

```
def f(x):  
    return x*x  
nums = [1, 2, 3]  
for num in nums:  
    print (f(num))
```

ან შეიძლება ასეც:

```

def f(x):
    return x*x
nums = [1, 2, 3]
print ([f(num) for num in nums])

```

მაგრამ **map()** გამოყენებით კოდს გაცილებით მარტივი სახე აქვს:

```

def f(x):
    return x*x
nums = [1, 2, 3]
print (map(f, nums))

```

მაგალითიდან ჩანს, რომ **map** მიიღებს რაიმე ფუნქციას და სიას, ხოლო აბრუნებს შედეგს ასევე, სიის სახით. მას შეიძლება გადაეცეს რამდენიმე სია, მაშინ ფუნქციამ უნდა მიიღოს რამდენიმე არგუმენტი, **map**-ში გადაცემული სიის რაოდენობის მიხედვით:

```

def f(x, y):
    return x*y
a = [1,3,4]
b = [3,4,5]
print (map(f, a, b))
[3, 12, 20]

```

თუ სიები სხვადასხვა სიგრძისაა, მაშინ ისინი შეივსება **None** მნიშვნელობებით საჭირო სიგრძემდე. თუ წავშლით **b** სიიდან ბოლო მნიშვნელობას - მაგალითი არ იმუშავებს, რადგან **f** ფუნქციაში მოხდება რიცხვის **None**-ზე გამრავლების მცდელობა, რაც არ არის დაშვებული

python-ში. ამიტომ თუ f ფუნქცია საკმაოდ მოცულობითია, მიზანშეწონილია შემოწმდეს გადასაცემი მნიშვნელობები. მაგალითად:

```
def f(x, y):  
    if (y == None):  
        y = 1  
    return x*y
```

თუ map კონსტრუქციაში ფუნქციის ადგილზე გვაქვს None, მაშინ map მოქმედებს zip მსგავსად, მაგრამ თუ გადასაცემი სიები სხვადასხვა სიგრძისაა შედეგში გამოვა None, რაც ზოგიერთ მომენტში მოსახერხებელია.

lambda() ფუნქცია გამოიყენება მაშინ, როდესაც საჭიროა განისაზღვროს ფუნქცია def func_name(): გარეშე.

ანონიმური ფუნქცია შეიძლება შეიცავდეს მხოლოდ ერთ გამოსახულებას, მაგრამ იგი სრულდება უფრო სწრაფად. ანონიმური ფუნქციები იქმნება lambda ინსტრუქციის დახმარებით. გარდა ამისა, არ არის აუცილებელი ისინი მიენიჭოს ცვლადს, როგორც ვაკეთებდით def func() ინსტრუქციით. ასეთი ფუნქციის ტანი არ შეიძლება შეიცავდეს ერთზე მეტ ინსტრუქციას ან გამოსახულებას. ასეთი ფუნქცია შეიძლება გამოყენებულ იქნას რაიმე კონვეიერული გამოთვლების ჩარჩოებში (მაგალითად filter(), map() და reduce()) ან დამოუკიდებლად, იქ სადაც მოითხოვება გამოთვლების წარმოება, რომელიც მოხერხებულია შევკრათ ფუნქციად:

```
>>> func = lambda x, y: x + y
```

```

>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y>('a', 'b'))
'ab'

```

lambda ფუნქციას, ჩვეულებრივისგან განსხვავებით, არ მოეთხოვება return ინსტრუქცია:

```

>>> func = lambda *args: args
>>> func(1, 2, 3, 4)
(1, 2, 3, 4)

```

lambda ფუნქციის გამოყენებით შესაძლებელია რამდენიმე მოქმედებიანი კოდის ერთ სტრიქონად დაწერა. მაგალითად:

```

def f(x, y):
    if (y == None):
        y = 1
    return x*y

```

შეიძლება წარმოვადგინოთ ასეთი სახით:

```

lambda x, y: x * (y if y is not None else 1)

```

filter ფუნქცია კმნის ელემენტების სიას, რომლისთვისაც ფუნქცია აბრუნებს ჭეშმარიტ მნიშვნელობას.

განვიხილოთ მაგალითი, სადაც გაფილტვრა ხდება filter ფუნქციაში მოცემული პირობის მიხედვით:

```
number_list = range(-5, 5)
less_than_zero = list(filter(lambda x: x < 0, number_list))
print(less_than_zero)
```

შედეგი:

```
[-5, -4, -3, -2, -1]
```

reduce() სასარგებლო ფუნქციაა სიებში გამოთვლებისა და შედეგის დაბრუნებისთვის. მას შეუძლია შეასრულოს მცოცავი გამოთვლები ერთმანეთის მომდევნო წყვილებისთვის სიებში. მაგალითად გვინდა გამოვითვალოთ მთელი რიცხვებისგან შემდგარი სიის ნამრავლი. ეს ამოცანა შეიძლება გადაწყდეს ციკლის გამოყენებითაც:

```
product = 1
list = [1, 2, 3, 4]
for num in list:
    product = product * num
print (product)
```

შედეგი:

```
24
```

გაცილებად მოკლედ დაიწერება კოდი **reduce()** ფუნქციის გამოყენებით:

```
>>> from functools import reduce
>>> product = reduce((lambda x, y: x * y), [1, 2, 3, 4])
>>> print (product)
```


მაგალითები:

1) ფუნქციის გამოძახება სხვადასხვა არგუმენტებით:

```
>>> def func(a, b, c=2): # c - არააუცილებელი  
არგუმენტი
```

```
...     return a + b + c
```

```
>>> func(1, 2) # a = 1, b = 2, c = 2 (დუმილით)
```

```
5
```

```
>>> func(1, 2, 3) # a = 1, b = 2, c = 3
```

```
6
```

```
>>> func(a=1, b=3) # a = 1, b = 3, c = 2
```

```
6
```

```
>>>func(a=3, c=6) #a = 3, c = 6, b არ არის  
განსაზღვრული
```

```
Traceback (most recent call last):
```

```
File "", line 1, in
```

```
func(a=3, c=6)
```

```
TypeError: func() takes at least 2 arguments (2 given)
```

2) ფუნქცია იღებს ცვლადი რაოდენობის არგუმენტებს. args არის კორტეჟი :

```
>>> def func(*args):
```

```
...     return args
```

```
>>> func(1, 2, 3, 'abc')
```

```
(1, 2, 3, 'abc')
```

```
>>> func()
```

```
()
```

```
>>> func(1)
```

(1.)

3) ფუნქცია იღებს ნებისმიერი რაოდენობის სახელობით არგუმენტს. kwargs ცვლადში ინახება ლექსიკონი:

```
>>> def func(**kwargs):
...     return kwargs
...
>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
>>> func()
{}
>>> func(a='python')
{'a': 'python'}
```

4) ფიბონაჩის რიცხვების გამოთვლა:

```
>>> def fibb(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fibb(n-1) + fibb(n-2)
>>> print(fibb(10))
55
```

5) ფაქტორიალის გამოთვლა:

```
>>> def factorial(n):
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
>>> print(factorial(5))
120
```

6) lambda ფუნქცია:

```
>>> (lambda x: x**2)(5)
25
```

7) Lambda - ფუნქცია შეიძლება მივანიჭოთ რაიმე ცვლადს და შემდგომ გამოვიყენოთ იგი ფუნქციის სახელის სახით:

```
>>> sqrt = lambda x: x**0.5
>>> sqrt(25)
5.0
```

8) Lambda - ფუნქციით შეიძლება დამუშავდეს სიები. განვიხილოთ map() მაგალითი. map() ფუნქცია იღებს ორ არგუმენტს, პირველი არის ფუნქცია, რომელიც გამოყენებული იქნება სიის ცალკეული ელემენტისთვის, მეორე - სიაა, რომელიც უნდა დამუშავდეს:

```
>>> l = [1, 2, 3, 4, 5, 6, 7]
>>> list(map(lambda x: x**3, l))
[1, 8, 27, 64, 125, 216, 343]
```

9) მაგალითში გამოყენებულია map() და lambda() ფუნქციები:

```

def square(x):
    return (x**2)
def cube(x):
    return (x**3)
funcs = [square, cube]
for r in range(5):
    value = map(lambda x: x(r), funcs)
    print (value)

```

შედეგი: [0, 0] [1, 1] [4, 8] [9, 27] [16, 64]

10) ქვემოთ მოცემულ მაგალითში შეიძლება გამოგვეყენებინა ციკლი, მაგრამ map() ფუნქცია გაცილებით სწრაფად მუშაობს. ფუნქცია pow იღებს ორ არგუმენტს ცალკეული გამოძახებისას:

```

>>> pow(3,5)
243
>>> pow(2,10)
1024
>>> pow(3,11)
177147
>>> pow(4,12)
16777216
>>> list(map(pow, [2, 3, 4], [10, 11, 12]))
[1024, 177147, 16777216]

```

11) reduce გამოყენების მაგალითი:

```

>>> reduce( (lambda x, y: x / y), [1, 2, 3, 4] )
0.041666666666666664

```

12) reduce აერთიანებს რამდენიმე სიას:

```

>>> reduce(list.__add__, [[1, 2, 3], [4, 5], [6, 7, 8]], [])

```

შედეგი: [1, 2, 3, 4, 5, 6, 7, 8]

VI თავი. მასივები

NumPy არის Python ენის ბიბლიოთეკა, რომელსაც აქვს დიდი, მრავალგანზომილებიანი მასივების და მატრიცების მხარდაჭერა, მაღალი დონის და სწრაფი მათემატიკური ფუნქციების დიდ ბიბლიოთეკასთან ერთად, რომელიც გამოიყენება მასივებზე ოპერაციების განსახორციელებლად.

NumPy ძირითადი ობიექტია ერთგვაროვანი მრავალგანზომილებიანი მასივი (numpy-ში ეწოდება `numpy.ndarray`). ეს არის ერთი ტიპის, ჩვეულებრივ რიცხვითი, ელემენტებისგან შედგენილი მრავალგანზომილებიანი მასივი.

6.1. მასივის შექმნა

NumPy-ში არსებობს მასივის შექმნის მრავალი ხერხი. ერთ-ერთი მარტივი ხერხია მასივის შექმნა ჩვეულებრივი სიების ან კორტეჟისგან `numpy.array()` ფუნქციის გამოყენებით. `array` არის ფუნქცია, რომელიც ქმნის `ndarray` ტიპის ობიექტს:

```
>>> import numpy as np
>>> a = np.array([1, 2, 3])
>>> a
array([1, 2, 3])
>>> type(a)
<class 'numpy.ndarray'>
```

`array()` ფუნქცია ქმნის ჩადგმულ მიმდევრობებს მრავალგანზომილებიან მასივებში. მასივის ელემენტების ტიპი დამოკიდებულია საწყისი მიმდევრობის ელემენტების ტიპზე. თუმცა შეიძლება იგი ხელახლა განისაზღვროს შექმნის მომენტში.

```
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

ასევე შეიძლება ტიპი ხელახლა განისაზღვროს შექმნის მომენტში:

```
>>> b = np.array([[1.5, 2, 3], [4, 5, 6]],
dtype=np.complex)
>>> b
array([[ 1.5+0.j,  2.0+0.j,  3.0+0.j],
       [ 4.0+0.j,  5.0+0.j,  6.0+0.j]])
```

`array()` ფუნქცია არ არის ერთადერთი მასივის შესაქმნელად. ჩვეულებრივ, მასივის ელემენტები დასაწყისში უცნობია, ხოლო მასივი, რომელშიც ისინი უნდა იქნან შენახული უნდა არსებობდეს. ამიტომ არსებობს რამდენიმე ფუნქცია იმისათვის, რომ შეიქმნას მასივი რაიმე საწყისი შემადგენლობით (დუმილით მასივის ტიპია - `float64`).

`zeros()` ფუნქცია ქმნის მასივს ნულებისგან, ხოლო `ones()` ფუნქცია - ერთებისგან. ორივე ფუნქცია მიიღებს კორტეჟს ზომით და არგუმენტს `dtype`:

```
>>> np.zeros((3, 5))
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.]])
```

```
>>> np.ones((2, 2, 2))
array([[[ 1.,  1.],
        [ 1.,  1.]],
       [[ 1.,  1.],
        [ 1.,  1.]])
```

`eye()` ფუნქცია ქმნის მატრიცას, რომლის დიაგონალზე არის 1-ები.

```
>>> np.eye(5)
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

`empty()` ფუნქცია ქმნის მასივს შევსების გარეშე. საწყისი შემადგენლობა შემთხვევითია და დამოკიდებულია მეხსიერების მდგომარეობაზე მასივის შექმნის მომენტში.

```
>>> np.empty((3, 3))
array([[ 6.93920488e-310, 6.93920488e-310, 6.93920149e-310],
       [ 6.93920058e-310, 6.93920058e-310, 6.93920058e-310],
       [6.93920359e-310, 0.00000000e+000, 6.93920501e-310]])
```

```
>>> np.empty((3, 3))
```

```
array([[ 6.93920488e-310, 6.93920488e-310, 6.93920147e-310],
       [ 6.93920149e-310, 6.93920146e-310, 6.93920359e-310],
       [ 6.93920359e-310, 0.00000000e+000, 3.95252517e-322]])
```

რიცხვთა მიმდევრობის შესაქმნელად NumPy-ში არის `arange()` ფუნქცია, რომელიც ანალოგიურია Python-ში ჩაშენებული `range()` ფუნქციისა, მაგრამ სივრცის ნაცვლად იგი აბრუნებს მასივებს და მიიღებს არამხოლოდ მთელ მნიშვნელობებს:

```
>>> np.arange(10, 30, 5)
```

```
array([10, 15, 20, 25])
```

```
>>> np.arange(0, 1, 0.1)
```

```
array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])
```

`arange()` გამოყენებისას `float` ტიპის არგუმენტებით, ვერ განვსაზღვრავთ თუ რამდენი ელემენტი მიიღება მცურავმიმდინარე რიცხვების სიზუსტის შეზღუდულობის გამო). ამიტომ ასეთ შემთხვევაში უმჯობესია `linspace()`, ფუნქციის გამოყენება, რომელიც ბიჯის ნაცვლად ერთ-ერთი არგუმენტის სახით მიიღებს რიცხვს, რომელიც საჭირო ელემენტების რაოდენობის ტოლია:

```
>>> np.linspace(0, 2, 9) # 9 რიცხვი 0-დან 2-ის ჩათვლით
```

```
array([ 0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2. ])
```

`fromfunction()` იყენებს ფუნქციას ინდექსების ყველა კომბინაციისთვის:

```
>>> def f1(i, j):
```



```

...     return 3 * i + j
...
>>> np.fromfunction(f1, (3, 4))
array([[ 0.,  1.,  2.,  3.],
       [ 3.,  4.,  5.,  6.],
       [ 6.,  7.,  8.,  9.]])
>>> np.fromfunction(f1, (3, 3))
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.],
       [ 6.,  7.,  8.]])

```

6.2. მასივების დაბეჭდვა

თუ მასივი ძალიან დიდია დასაბეჭდად, NumPy ავტომატურად მალავს მასივის ცენტრალურ ნაწილს და გამოაქვს მხოლოდ მისი კუთხეები:

```

>>> print(np.arange(0, 3000, 1))
[ 0  1  2 ..., 2997 2998 2999]

```

თუ საჭიროა მთელი მასივის დანახვა, გამოიყენეთ `numpy.set_printoptions` ფუნქცია:

```

np.set_printoptions(threshold=np.nan)

```

6.3. მასივებზე საბაზო ოპერაციები

მასივებზე მათემატიკური ოპერაციები სრულდება ცალკეული ელემენტის მიხედვით. შეიქმნება ახალი მასივი, რომელიც შეივსება ოპერატორის მოქმედების შედეგებით:

```
>>> import numpy as np
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> a + b
array([20, 31, 42, 53])
>>> a - b
array([20, 29, 38, 47])
>>> a * b
array([ 0, 30, 80, 150])
>>> a / b #0-ზე გაყოფისას დაბრუნდება inf (უსასრულობა)
array([  inf, 30.  , 20.  , 16.66666667])
<string>:1: RuntimeWarning: divide by zero encountered in
true_divide
>>> a ** b
array([  1,  30, 1600, 125000])
>>> a % b # 0-ზე გაყოფისას ნაშთის აღების დროს
დაბრუნდება 0
<string>:1: RuntimeWarning: divide by zero encountered in
remainder
array([0, 0, 0, 2])
```

მაგალითში, ბუნებრივია, მასივები უნდა იყოს ერთნაირი ზომის:

```
>>> c = np.array([[1, 2, 3], [4, 5, 6]])
>>> d = np.array([[1, 2], [3, 4], [5, 6]])
>>> c + d
```

Traceback (most recent call last):

File "<input>", line 1, in <module>

ValueError: operands could not be broadcast together with shapes (2,3) (3,2)

შეიძლება შესრულდეს მათემატიკური ოპერაციები მასივსა და რიცხვს შორის:

```
>>> a + 1
array([21, 31, 41, 51])
>>> a ** 3
array([ 8000, 27000, 64000, 125000])
>>> a < 35 # გაფილტვრაც შეიძლება მოვახდინოთ
array([ True,  True, False, False], dtype=bool)
```

NumPy წარმოგვიდგენს მრავალ მათემატიკურ ოპერაციას მასივების დასამუშავებლად:

```
>>> np.cos(a)
array([ 0.40808206,  0.15425145, -0.66693806,  0.96496603])
>>> np.arctan(a)
array([ 1.52083793,  1.53747533,  1.54580153,  1.55079899])
>>> np.sinh(a)
array([ 2.42582598e+08,  5.34323729e+12,  1.17692633e+17,
```

```
2.59235276e+21])
```

მრავალი უნარული ოპერაცია, როგორცაა მასივის ყველა ელემენტის ჯამის გამოთვლა, წარმოდგენილია ასევე, ndarray კლასის მეთოდების სახით:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.sum(a)
21
>>> a.sum()
21
>>> a.min()
1
>>> a.max()
6
```

დუმილით, ეს ოპერაციები გამოიყენება მასივებზე, თითქოს იგი იყოს რიცხვების სია, მისი ფორმისგან დამოუკიდებლად. axis პარამეტრის მითითებით შეიძლება გამოიყენოთ ოპერაცია მასივის მითითებული ღერძებისთვის:

```
>>> a.min(axis=0) # უმცირესი რიცხვი ცალკეულ სვეტში
array([1, 2, 3])
>>> a.min(axis=1) # უმცირესი რიცხვი ცალკეულ სტრიქონში
array([1, 4])
```

6.4. ინდექსები, ჭრები, იტერაციები

ერთგანზომილებიან მასივებში ინდექსირება, ჭრები და იტერაციები მსგავსია სიების და Python-ის სხვა მიმდევრობების.

```
>>> a = np.arange(10) ** 3
>>> a
array([ 0,  1,  8, 27, 64, 125, 216, 343, 512, 729])
>>> a[1]
1
>>> a[3:7]
array([ 27, 64, 125, 216])
>>> a[3:7] = 8
>>> a
array([ 0,  1,  8,  8,  8,  8,  8, 343, 512, 729])
>>> a[::-1]
array([729, 512, 343,  8,  8,  8,  8,  8,  1,  0])
>>> del a[4:6]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: cannot delete array elements
>>> for i in a:
...     print(i ** (1/3))
შედეგი:
0.0
1.0
2.0
```

2.0
2.0
2.0
2.0
7.0
8.0
9.0

მრავალგანზომილებიან მასივში ცალკეულ ღერძზე მოდის ერთი ინდექსი. ინდექსები გადაეცემა მძიმეებით გამოყოფილი რიცხვთა მიმდევრობის სახით:

```
>>> b = np.array([[ 0, 1, 2, 3],
...               [10, 11, 12, 13],
...               [20, 21, 22, 23],
...               [30, 31, 32, 33],
...               [40, 41, 42, 43]])
>>> b[2,3] # მეორე სტრიქონი, მესამე სვეტი
23
>>> b[(2,3)]
23
>>> b[2][3] # შეიძლება ასეც
23
>>> b[:,2] # მესამე სვეტი
array([ 2, 12, 22, 32, 42])
>>> b[:2] # პირველი-მეორე სტრიქონი
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13]])
```

```
>>> b[1:3, :, :] # მეორე და მესამე სტრიქონები
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

როდესაც ინდექსები ნაკლებია, ვიდრე ღერძები, გამოტოვებული ინდექსები გათვალისწინებულ იქნება ჭრების დახმარებით:

```
>>> b[-1] # ბოლო სტრიქონი. ექვივალენტურია b[-1,: ]
array([40, 41, 42, 43])
```

`b[i]` - ასე წაიკითხება `b[i, <იმდენი სიმბოლო ':', რამდენიც არის საჭირო>]`. NumPy -ში ეს შეიძლება ჩაიწეროს წერტილების დახმარებით: `b[i, ...]`.

მაგალითად, თუ `x` აქვს რანგი 5 (ანუ მას აქვს 5 ღერძი), მაშინ :

- `x[1, 2, ...]` ექვივალენტურია `x[1, 2, :, :, :]`,
- `x[... , 3]` იგივეა, რაც `x[:, :, :, 3]` და
- `x[4, ... , 5, :]` ეს არის `x[4, :, :, 5, :]`.

```
>>> a = np.array([[0, 1, 2], [10, 12, 13]], [[100, 101, 102], [110,
112, 113]])
```

```
>>> a.shape
(2, 2, 3)
```

```
>>> a[1, ...] # იგივეა, რაც a[1, :, :] ან a[1]
array([[100, 101, 102],
       [110, 112, 113]])
```

```
>>> c[... ,2] # იგივეა, რაც a[:, :, 2]
array([[ 2, 13],
       [102, 113]])
```

მრავალგანზომილებიანი მასივის იტერირება იწყება პირველი ღერძიდან:

```
>>> for row in a:  
...     print(row)
```

შედეგი:

```
[[ 0 1 2]  
 [10 12 13]]  
[[100 101 102]  
 [110 112 113]]
```

თუ საჭიროა მთელი მასივის ცალკეულ ელემენტებად გადარჩევა, თითქოს იგი იყოს ერთგანზომილებიანი, ამისათვის შეიძლება გამოყენებულ იქნას flat ატრიბუტი:

```
>>> for el in a.flat:  
...     print(el)
```

შედეგი:

```
0  
1  
2  
10  
12  
13  
100  
101  
102  
110  
112  
113
```


6.5. მასივის ფორმის შეცვლა

მასივს აქვს ფორმა, რომელიც განისაზღვრება ცალკეული ღერძის გასწვრივ ელემენტების რიცხვით:

```
>>> a
array([[[ 0, 1, 2],
        [10, 12, 13]],
       [[100, 101, 102],
        [110, 112, 113]]])
>>> a.shape
(2, 2, 3)
```

მასივის ფორმა შეიძლება შეიცვალოს სხვადასხვა ბრძანების დახმარებით:

```
>>> a.ravel() # მასივს გარდაქმნის ბრტყლად
array([ 0, 1, 2, 10, 12, 13, 100, 101, 102, 110, 112, 113])
>>> a.shape = (6, 2) # ფორმის შეცვლა
>>> a
array([[ 0, 1],
       [ 2, 10],
       [12, 13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.transpose() # ტრანსპონირება
array([[ 0, 2, 12, 100, 102, 112],
       [ 1, 10, 13, 101, 110, 113]])
>>> a.reshape((3, 4)) # ფორმის შეცვლა
```

```
array([[ 0,  1,  2, 10],
       [ 12, 13, 100, 101],
       [102, 110, 112, 113]])
```

ravel() ფუნქციის შედეგად ელემენტების განლაგება შეესაბამება ჩვეულებრივ C ენის სტილს ანუ უფრო სწრაფად იცვლება მარჯვენა ინდექსი - a[0,0] ელემენტის მომდევნო არის a[0,1]. ravel() და reshape() ფუნქციებს შეუძლიათ, დამატებითი არგუმენტის გამოყენებით, იმუშაონ FORTRAN სტილში, რომელშიც სწრაფად შეიცვლება უფრო მარცხენა ინდექსი.

```
>>> a
array([[ 0,  1],
       [ 2, 10],
       [ 12, 13],
       [100, 101],
       [102, 110],
       [112, 113]])
>>> a.reshape((3, 4), order='F')
array([[ 0, 100,  1, 101],
       [ 2, 102, 10, 110],
       [ 12, 112, 13, 113]])
```

reshape() მეთოდი აბრუნებს თავის არგუმენტს შეცვლილი ფორმით, იმ დროს როდესაც resize() მეთოდი შეცვლის თვით მასივს:

```
>>> a.resize((2, 6))
>>> a
```

```
array([[ 0,  1,  2, 10, 12, 13],
       [100, 101, 102, 110, 112, 113]])
```

გადაწყობის ოპერაციისას თუ ერთ-ერთი არგუმენტი მოცემულია -1, მაშინ იგი ავტომატურად გამოითვლება დანარჩენი მოცემულის შესაბამისად:

```
>>> a.reshape((3, -1))
array([[ 0,  1,  2, 10],
       [12, 13, 100, 101],
       [102, 110, 112, 113]])
```

6.6. მასივების გაერთიანება

რამდენიმე მასივი შეიძლება გაერთიანდეს `hstack` და `vstack` ფუნქციების გამოყენებით.

`hstack()` აერთიანებს მასივებს პირველი ღერძის მიხედვით, ხოლო `vstack()` - ბოლო ღერძის მიხედვით:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = np.array([[5, 6], [7, 8]])
>>> np.vstack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
>>> np.hstack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

`column_stack()` ფუნქცია აერთიანებს ერთგანზომილებიან მასივს ორგანზომილებიანი მასივის სვეტების სახით:

```
>>> np.column_stack((a, b))
array([[1, 2, 5, 6],
       [3, 4, 7, 8]])
```

ანალოგიურად, სტრიქონებისთვის არის `row_stack()` ფუნქცია:

```
>>> np.row_stack((a, b))
array([[1, 2],
       [3, 4],
       [5, 6],
       [7, 8]])
```

6.7. მასივის გახლეჩვა

`hsplit()` გამოყენებით შეიძლება გაიხლიჩოს მასივი ჰორიზონტალური ღერძის გასწვრივ. უნდა მიეთითოს თუ რამდენი ერთნაირი მასივი გამოვიდეს ან რომელ სვეტებზე გაიხლიჩოს:

```
>>> a = np.arange(12).reshape((2, 6))
>>> a
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
>>> np.hsplit(a, 3) # გაიხლიჩოს 3 ნაწილად
[array([[0, 1], [6, 7]]),
 array([[2, 3], [8, 9]])]
```

```

array([[ 4,  5], [10, 11]])
>>> np.hsplit(a, (3, 4)) # a გაიჭრას მესამე და მეოთხე
# სვეტის შემდეგ
[array([[0, 1, 2], [6, 7, 8]]),
array([[3], [9]]),
array([[ 4,  5], [10, 11]])]

```

vsplit() ფუნქცია გახლექს მასივს ვერტიკალური ღერძის გასწვრივ, ხოლო array_split() საშუალებას იძლევა მიეთითოს ღერძები, სადაც უნდა მოხდეს გახლეჩვა.

6.8. ასლები და წარმოდგენები

მასივებთან მუშაობის დროს მათი მონაცემები ზოგჯერ საჭიროა გადაკოპირდეს სხვა მასივში. შესაძლებელია 3 შემთხვევა:

უბრალო მინიჭება არ ქმნის არც მასივის ასლს და არც მისი მონაცემების ასლს:

```

>>> a = np.arange(12)
>>> b = a # ახალი ობიექტი არ შექმნილა
>>> b is a #a და b ორი სახელია ერთიდაიმავე ndarray
#ობიექტისთვის
True
>>> b.shape = (3,4) # a ფორმის შეცვლა
>>> a.shape
(3, 4)

```

Python გადასცემს ცვალებად ობიექტებს, როგორც ბმულებს, ამიტომ ფუნქციის გამოძახება არ ქმნის ასლებს.

მასივების სხვადასხვა ობიექტებს შეუძლიათ გამოიყენონ ერთიდაიგივე მონაცემები. `view()` მეთოდი ქმნის მასივის ახალ ობიექტს, რომელიც არის იმავე მონაცემების წარმოდგენა:

```
>>> c = a.view()
>>> c is a
False
>>>c.base is a #c მონაცემთა წარმოდგენაა, რომელიც
#ეკუთვნის a-ს
True
>>> c.flags.owndata
False
>>>
>>> c.shape = (2,6) # ფორმა a არ შეიცვლება
>>> a.shape
(3, 4)
>>> c[0,4] = 1234 # a მონაცემები შეიცვლება
>>> a
array([[ 0, 1, 2, 3],
       [1234, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

მასივის ჭრა არის წარმოდგენა:

```
>>> s = a[:,1:3]
>>> s[:] = 10
```

```
>>> a
array([[ 0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

`copy()` მეთოდი ქმნის მასივის და მისი მონაცემების
ნამდვილ ასლს:

```
>>> d = a.copy() # შეიქმნება მასივის ახალი ობიექტი
#ახალი მონაცემებით
```

```
>>> d is a
```

```
False
```

```
>>> d.base is a # d -ს არა აქვს საერთო a-სთან
```

```
False
```

```
>>> d[0, 0] = 9999
```

```
>>> a
```

```
array([[ 0, 10, 10, 3],
       [1234, 10, 10, 7],
       [ 8, 10, 10, 11]])
```

VII თავი. გამონაკლისების დამუშავება

Python ენაზე დაპროგრამების დროს შეიძლება ადგილი ჰქონდეს ორი ტიპის შეცდომას. პირველი არის სინტაქსური შეცდომა - `syntax error`. იგი გამოჩნდება პროგრამის კოდის დაწერისას ენის სინტაქსის დარღვევის შედეგად. ასეთი შეცდომების არსებობისას პროგრამა არ კომპილირდება.

მეორე ტიპის შეცდომა არის შესრულების შეცდომა (`runtime error`). იგი გამოჩნდება უკვე კომპილირებულ პროგრამაში მისი შესრულების დროს. ასეთი შეცდომები იწოდებიან გამონაკლისებად. მაგალითად, წინა თემებში განვიხილეთ რიცხვის გარდაქმნა სტრიქონად:

```
string = "5"  
number = int(string)  
print(number)
```

მოცემული კოდი წარმატებით კომპილირდება და შესრულდება, რამდენადაც სტრიქონი "5" შეიძლება კონვერტირდეს რიცხვად. განვიხილოთ სხვა მაგალითი:

```
string = "hello"  
number = int(string)  
print(number)
```

აღნიშნული კოდის შესრულებისას გამოჩნდება გამონაკლისი `ValueError`, რამდენადაც `hello` სტრიქონი არ შეიძლება გარდაქმნას რიცხვად. ერთის მხრივ აქ, ცხადია, რომ რიცხვი არ წარმოადგენს რიცხვს, მაგრამ

შეიძლება საქმე გვექონდეს მომხმარებლის შეტანასთან, რომელმაც შეიძლება ის არ შეიტანოს, რასაც ველოდებთ:

```
string = input("შეიტანეთ რიცხვი: ")
number = int(string)
print(number)
```

გამონაკლისის აღძვრისას პროგრამის მუშაობა წყდება. ამისგან თავის დასაღწევად და გამონაკლისის დასამუშავებლად Python-ში არის try...except კონსტრუქცია, რომელსაც აქვს შემდეგი ფორმალური განსაზღვრა:

try:

ინსტრუქციები

except [ტიპი_გამონაკლისები]:

ინსტრუქციები

მთელი ძირითადი კოდი, რომელშიც პოტენციურად შეიძლება აღიძვრას გამონაკლისი, განთავსდება try გასაღებური სიტყვის შემდეგ. თუ ამ კოდში გენერირდება გამონაკლისი, მაშინ კოდის მუშაობა try კოდში შეწყდება და შესრულება გადადის except ბლოკში.

except გასაღებური სიტყვის შემდეგ შეიძლება მიეთითოს რომელი გამონაკლისი უნდა დამუშავდეს (მაგალითად, ValueError თუ KeyError). except სიტყვის შემდეგ სტრიქონზე იწერება except ბლოკის ინსტრუქციები, რომლებიც სრულდება გამონაკლისის წარმოშობისას.

განვიხილოთ გამონაკლისის დამუშავება სტრიქონის რიცხვად გარდაქმნის მაგალითზე:

try:

```
number = int(input("შეიტანეთ რიცხვი: "))
```

```
print("შეტანილი რიცხვი:", number)
```

except:

```
print("გარდაქმნამ ჩაიარა წარუმატებლად")
```

```
print("პროგრამის დასრულება")
```

შეგვაქვს სტრიქონი:

```
შეიტანეთ რიცხვი: hello
```

```
გარდაქმნამ ჩაიარა წარუმატებლად
```

```
პროგრამის დასრულება
```

როგორც კონსოლზე გამოტანიდან ჩანს, სტრიქონის შეტანისას რიცხვის გამოტანა კონსოლზე არ მოხდება, ხოლო პროგრამის შესრულება გადადის except ბლოკზე.

შევიტანოთ სწორი რიცხვი:

```
შეიტანეთ რიცხვი: 22
```

```
შეტანილი რიცხვია: 22
```

```
პროგრამის დასრულება
```

ეხლა ყველაფერი ნორმალურად შესრულდა, გამონაკლისი არ აღიძვრა და შესაბამისად except ბლოკი არ ამუშავდა.

მაგალითში მუშავდებოდა ერთდროულად ყველა გამონაკლისი, რომელიც შეიძლება აღიძვრას კოდში. ჩვენ შეგვიძლია დავაკონკრეტოთ დასამუშავებელი გამონაკლისის ტიპი, რომელიც უნდა მიეთითოს except სიტყვის შემდეგ:

try:

```
number = int(input("შეიტანეთ რიცხვი: "))
```

```

print("შეტანილი რიცხვი:", number)
except ValueError:
    print("გარდაქმნამ ჩაიარა წარუმატებლად")
print("პროგრამის დასრულება")

```

თუ სიტუაცია ისეთია, რომ პროგრამაში შეიძლება დაგენერირდეს სხვადასხვა ტიპის გამონაკლისი, მაშინ შეგვიძლია ისინი დავამუშავოთ ცალ-ცალკე დამატებითი except გამოსახულების გამოყენებით:

```

try:
    number1=int(input("შეიტანეთ პირველი რიცხვი: "))
    number2=int(input("შეიტანეთ მეორე რიცხვი: "))
    print("გაყოფის შედეგი:", number1/number2)
except ValueError:
    print("გარდაქმნამ ჩაიარა წარუმატებლად")
except ZeroDivisionError:
    print("რიცხვის ნულზე გაყოფის მცდელობა")
except Exception:
    print("ზოგადი გამონაკლისი")
print("პროგრამის დასრულება")

```

თუ წარმოიშვება გამონაკლისი სტრიქონის რიცხვად გარდაქმნის შედეგად, მაშინ იგი დამუშავდება except ValueError ბლოკის მიერ. თუ მეორე რიცხვი ტოლი იქნება ნულის, ანუ იქნება ნულზე გაყოფა, მაშინ აღიძვრება ZeroDivisionError გამონაკლისი და იგი დამუშავდება except ZeroDivisionError ბლოკის მიერ.

Exception ტიპი წარმოადგენს საერთო გამონაკლისს, რომელშიც მოხვდება ყველა გამონაკლისი სიტუაცია.

ამიტომ ასეთ შემთხვევაში ნებისმიერი გამონაკლისი, რომელიც არ წარმოადგენს ValueError ან ZeroDivisionError ტიპს, დამუშავდება except Exception ბლოკში.

გამონაკლისების დამუშავებისას შეიძლება გამოყენებულ იქნას არააუცილებელი ბლოკი - finally. მისი განმასხვავებელი თავისებურება არის, რომ იგი სრულდება იმისგან დამოუკიდებლად იყო თუ არა გენერირებული გამონაკლისი:

```
try:
    number = int(input("შეიტანეთ რიცხვი: "))
    print("შეიტანილი რიცხვი:", number)
except ValueError:
    print("რიცხვის გარდაქმნა არ მოხერხდა")
finally:
    print("try ბლოკმა დაასრულა შესრულება")
    print("პროგრამის დასრულება")
```

as ოპერატორის დახმარებით შეგვიძლია გამონაკლისის შესახებ მთელი ინფორმაცია გადავცეთ ცვლადს, რომელიც შეიძლება გამოვიყენოთ except ბლოკში:

```
try:
    number = int(input("შეიტანეთ რიცხვი: "))
    print("შეიტანილი რიცხვი:", number)
except ValueError as e:
    print("შეტყობინება გამონაკლისის შესახებ", e)
    print("პროგრამის დასრულება")
```

არაკორექტული შეტანის მაგალითი:

შეიტანეთ რიცხვი: fdsf

შეტყობინება გამონაკლისის შესახებ invalid literal for int() with base 10: 'fdsf

პროგრამის დასრულება

ზოგჯერ საჭიროა ხელით დაგენერირდეს ესათუის გამონაკლისი. ამისათვის გამოიყენება raise ოპერატორი:

try:

```
number1=int(input("შეიტანეთ პირველი რიცხვი:"))
```

```
number2 = int(input("შეიტანეთ მეორე რიცხვი: "))
```

```
if number2 == 0:
```

```
    raise Exception("მეორე რიცხვი არ უნდა იყოს 0")
```

```
print("ორი რიცხვის გაყოფის შედეგი:",
```

```
number1/number2)
```

except ValueError:

```
    print("შეტანილია არაკორექტული მონაცემები")
```

except Exception as e:

```
    print(e)
```

```
    print("პროგრამის დასრულება")
```

გამონაკლისის გამოძახებისას შეგვიძლია მას გადავცეთ შეტყობინება, რომელიც შემდეგ შეიძლება გამოტანილ იქნას მომხმარებლისთვის:

შეიტანეთ პირველი რიცხვი: 1

შეიტანეთ მეორე რიცხვი: 0

მეორე რიცხვი არ უნდა იყოს 0

პროგრამის დასრულება

VIII თავი. ფაილები

8.1. ფაილის გახსნა და დახურვა

Python ენას აქვს სხვადასხვა ტიპის ფაილებთან მუშაობის შესაძლებლობა, მაგრამ პირობითად ეს ფაილები შეიძლება დავყოთ ორ სახეობად: ტექსტური და ბინარული. ტექსტურია ფაილები cvs, txt, html გაფართოებით, ზოგადად ნებისმიერი ფაილი, რომელიც ინახავს ინფორმაციას ტექსტური სახით. ბინარული ფაილებია: გამოსახულება, აუდიო, ვიდეოფაილები და სხვა. ფაილის ტიპის მიხედვით მასთან მუშაობა შეიძლება იყოს განსხვავებული.

ფაილებთან მუშაობის დროს უნდა დავიცვათ ოპერაციების ზოგიერთი მიმდევრობა:

1. ფაილის გახსნა **open()** მეთოდის გამოყენებით;
2. ფაილის წაკითხვა **read()** მეთოდის დახმარებით ან ჩაწერა ფაილში **write()** მეთოდის საშუალებით;
3. ფაილის დახურვა **close()** მეთოდით.

ფაილთან მუშაობის დასაწყებად, იგი უნდა გაიხსნას **open()** ფუნქციის დახმარებით, რომლის სინტაქსია:

```
open(file, mode)
```

ფუნქციის პირველი პარამეტრი წარმოადგენს გზას ფაილამდე. ეს გზა შეიძლება იყოს აბსოლუტური ანუ იწყებოდეს დისკის აღმნიშვნელი სიმბოლოთი, მაგალითად: C://somedir/somefile.txt. შეიძლება იყოს დისკის მითითების გარეშე მაგალითად:

somefile.txt - ასეთ შემთხვევაში ფაილის ძებნა განხორციელდება Python-ის გაშვებული სკრიპტის განთავსების მიხედვით.

მეორე გადასაცემი არგუმენტი - mode დააყენებს ფაილის გახსნის რეჟიმს იმის მიხედვით, თუ რის გაკეთებას ვაპირებთ. არსებობს 4 ზოგადი რეჟიმი:

- **r (Read)** - ფაილი გაიხსნება წასაკითხად. თუ ფაილი ნაპოვნი არ არის, გენერირდება გამონაკლისი FileNotFoundError;
- **w (Write)** - ფაილი გაიხსნება ჩასაწერად. თუ ფაილი არ არის, მაშინ იგი შეიქმნება. თუ მსგავსი ფაილი უკვე არის, მაშინ იგი ახლიდან შეიქმნება და შესაბამისად ძველი მონაცემები მასში წაიშლება.
- **a (Append)** - ფაილი გაიხსნება ჩასამატებლად. თუ ფაილი არ არის, მაშინ იგი შეიქმნება, ხოლო თუ არის მაშინ მონაცემები ჩაიწერება ფაილის ბოლოში.
- **b (Binary)** - გამოიყენება ბინარული ფაილის შესაქმნელად. გამოიყენება სხვა რეჟიმებთან ერთად - w ან r.

ფაილთან მუშაობის დასრულების შემდეგ იგი აუცილებლად უნდა დაიხუროს close() მეთოდით. მოცემული მეთოდი გაათავისუფლებს ფაილთან დაკავშირებულ ყველა გამოყენებულ რესურსს.

```
მაგალითად, გავხსნათ ტექსტური ფაილი: hello.txt  
myfile = open("hello.txt", "w")  
myfile.close()
```

ფაილის გახსნისას ან მასთან მუშაობის დროს შეიძლება წავაწყდეთ სხვადასხვა გამონაკლისს, მაგალითად, მასთან არ არის წვდომა და ა.შ. ამ შემთხვევაში პროგრამა იქნება მცდარი, ხოლო მისი შესრულება არ მიაღწევს close მეთოდის გამოძახებამდე და შესაბამისად ფაილი არ დაიხურება.

ამ შემთხვევაში შეგვიძლია დავამუშავოთ გამონაკლისები:

```
try:
```

```
    somefile = open("hello.txt", "w") #გაიხსნა ფაილი  
ჩასაწერად
```

```
try:
```

```
    somefile.write("hello world")#ჩაიწერა ფაილში  
except Exception as e:
```

```
    print(e)
```

```
finally:
```

```
    somefile.close() #დაიხურა ფაილი
```

```
except Exception as ex:
```

```
    print(ex)
```

ამ შემთხვევაში ფაილთან მთელი მუშაობა ჩაშენებულ try ბლოკში. თუ აღიძვრება რაიმე გამონაკლისი, მაშინ ნებისმიერ შემთხვევაში finally ბლოკში ფაილი დაიხურება.

არის უფრო მოხერხებული კონსტრუქცია:

```
with open(file, mode) as file_obj:
```

```
    ინსტრუქციები
```


ეს კონსტრუქცია განსაზღვრავს გახსნილი ფაილისთვის file_obj ცვლადს და ასრულებს ინსტრუქციის ნაკრებს. მათი შესრულების შემდეგ ფაილი ავტომატურად იხურება. იქამდე, თუ ინსტრუქციის შესრულებისას with ბლოკში აღიძვრება რაიმე გამონაკლისი, ფაილი დაიხურება.

წინა მაგალითი, შეიძლება გადავწეროთ შემდეგნაირად:

```
with open("hello.txt", "w") as somefile:  
    somefile.write("hello world")
```

8.2. ტექსტური ფაილები

ტექსტური ფაილის ჩასაწერად გახსნისთვის, საჭიროა გამოიყენებულ იქნას w რეჟიმი. შემდეგ ჩაწერისთვის გამოიყენება write(str) მეთოდი, რომელშიც გადაეცემა ჩასაწერი სტრიქონი. უნდა აღინიშნოს, რომ ჩაიწერება სახელდობრ, სტრიქონი. თუ საჭიროა რიცხვის ან სხვა ტიპის მონაცემების ჩაწერა, საჭიროა წინასწარ მათი კონვერტირება სტრიქონად. ჩავწეროთ რაიმე ინფორმაცია hello.txt ფაილში.

```
with open("hello.txt", "w") as file:  
    file.write("hello world")
```

თუ გავხსნით საქალაქდეს, რომელშიც იმყოფება Python-ის მიმდინარე სკრიპტი, მაშინ იქ ვნახავთ hello.txt ფაილს. თუ გვინდა აღნიშნული ფაილი შეიქმნას სხვა მისამართზე, მაშინ უნდა მიეთითოს ეს მისამართი.

მაგალითად, იგივე ფაილი შევქმნათ d დისკზე, რისთვისაც გამოვიყენოთ კოდი:

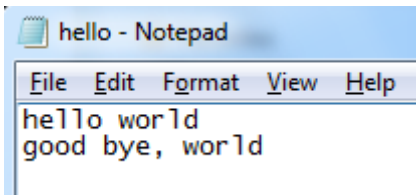
```
with open("d:\hello.txt", "w") as file:  
    file.write("hello world")
```

შექმნილი ფაილი შეიძლება გაიხსნას ნებისმიერ ტექსტურ რედაქტორში და სურვილისამებრ შეიცვალოს მისი შემადგენლობა.

დავამატოთ ფაილს კიდევ ერთი სტრიქონი:

```
with open("d:\hello.txt", "a") as file:  
    file.write("\ngood bye, world")
```

დამატება ხდება ფაილის ბოლო სიმბოლოს შემდეგ. ამიტომ თუ გვინდა ტექსტი დავამატოთ ახალი სტრიქონიდან, დასამატებელი ტექსტის წინ უნდა გამოვიყენოთ "\n" სიმბოლო. ჯამში hello.txt ფაილს ექნება შემდეგი შინაარსი:



ფაილში ჩაწერის კიდევ ერთი ხერხია print() სტანდარტული მეთოდი, რომელიც გამოიყენება მონაცემების გამოსატანად კონსოლზე:

```
with open("d:\hello.txt", "a") as hello_file:  
    print("\nHello, world", file=hello_file)
```

print მეთოდში მეორე პარამეტრის სახით გადაეცემა ფაილის დასახელება file პარამეტრით, ხოლო პირველი პარამეტრი არის ფაილში ჩასაწერი სტრიქონი.

8.3. ფაილის წაკითხვა

ფაილის წასაკითხად იგი გაიხსნება r (Read) რეჟიმში და შემდეგ შესაძლებელი იქნება მისი შემადგენლობის წაკითხვა სხვადასხვა მეთოდით:

- readline(): ფაილიდან კითხულობს ერთ სტრიქონს;
- read(): კითხულობს ფაილის მთელ შემადგენლობას ერთ სტრიქონად;
- readlines(): კითხულობს ფაილის ყველა სტრიქონს სიაში.

მაგალითად, წავიკითხოთ ზემოთ ჩაწერილი ფაილი სტრიქონ-სტრიქონ:

```
with open("d:\hello.txt", "r") as file:  
    for line in file:  
        print(line, end="")
```

მიუხედავად იმისა, რომ readline() მეთოდს ცხადად არ ვიყენებთ ცალკეული სტრიქონის წასაკითხად, ფაილის გადარჩევისას ეს მეთოდი ავტომატურად გამოიძახება ცალკეული ახალი სტრიქონის მისაღებად. ამიტომ ციკლში ხელით აზრი არ აქვს readline მეთოდის გამოძახებას. რადგანაც სტრიქონები გამოიყოფა "\n" სტრიქონზე გადასვლის სიმბოლოთი, მომდევნო

სტრიქონზე ზედმეტი გადატანის გამოსარიცხად print ფუნქციაში გადაეცემა end="" მნიშვნელობა.

გამოვიძახოთ ცხადი სახით readline() მეთოდი ცალკეული სტრიქონის წასაკითხად:

```
with open("d:\hello.txt", "r") as file:
    str1 = file.readline()
    print(str1, end="")
    str2 = file.readline()
    print(str2)
```

გამოიტანს შედეგს:

```
...
== RESTART: C:/Users/Guliko/AppData/Local/Programs/1
hello world
good bye, world
```

readline მეთოდის გამოყენება შეიძლება ფაილის სტრიქონებად წასაკითხად while ციკლში:

```
with open("d:\hello.txt", "r") as file:
    line = file.readline()
    while line:
        print(line, end="")
        line = file.readline()
```

თუ ფაილი არ არის დიდი, იგი შეიძლება ერთიანად იქნას წაკითხული read() მეთოდის გამოყენებით:

```
with open("d:\hello.txt", "r") as file:
    content = file.read()
    print(content)
```

ასევე გამოვიყენოთ `readlines()` მეთოდი მთლიანი ფაილის წასაკითხად სტრიქონების სიის სახით:

```
with open("d:\hello.txt", "r") as file:
    contents = file.readlines()
    str1 = contents[0]
    str2 = contents[1]
    print(str1, end="")
    print(str2)
```

ფაილის წაკითხვისას შეიძლება წავაწყდეთ იმ პრობლემას, რომ მისი კოდირება არ ემთხვეოდეს ASCII-ს. ამ შემთხვევაში ჩვენ ცხადი სახით შეგვიძლია მივუთითოთ კოდირება `encoding` პარამეტრის დახმარებით:

```
filename = "d:\hello.txt"
with open(filename, encoding="utf8") as file:
    text = file.read()
```

შევადგინოთ მცირე ზომის სკრიპტი, რომელიც მომხმარებლის მიერ შეტანილ სტრიქონებს ჩაწერს ფაილში, შემდეგ მოხდება ფაილის შინაარსის წაკითხვა და ეკრანზე გამოტანა:

```
# ფაილის სახელი
FILENAME = "d:\messages.txt"
# განვსაზღვროთ ცარიელი სია
messages = list()
for i in range(4):
    message = input("შეიტანეთ სტრიქონი"+str(i+1) + " ")
```

```

    messages.append(message + "\n")
# სიის ჩაწერა ფაილში
with open(FILENAME, "a") as file:
    for message in messages:
        file.write(message)
# წაკითხვა ფაილიდან
print("წაკითხული შეტყობინება")
with open(FILENAME, "r") as file:
    for message in file:
        print(message, end="")
მიიღება შედეგი:
== RESTART: C:/Users/Guliko/AppData/Local/I
შეიტანეთ სტრიქონი 1: Sandro
შეიტანეთ სტრიქონი 2: Lomjaria
შეიტანეთ სტრიქონი 3: Anna
შეიტანეთ სტრიქონი 4: Karden
წაკითხული შეტყობინება
Sandro
Lomjaria
Anna
Karden
>>>

```

8.4. CSV ფაილები

ფაილური ფორმატებიდან ერთ-ერთი გავრცელებულია csv ფორმატი, რომელიც ინახავს ინფორმაციას მოხერხებული სახით. csv ფაილში ცალკეული სტრიქონი წარმოადგენს ცალკეულ ჩანაწერს ან სტრიქონს, რომელიც შედგება მძიმეებით გამოყოფილი ცალკეული სვეტისგან. ამიტომაც ჰქვია ფორმატს Comma Separated Values. მაგრამ

თუნდაც csv ფორმატი ეს არის ტექსტური ფაილების ფორმატი, Python სამუშაოს გასამარტივებლად უზრუნველყოფს csv სპეციალური ჩაშენებული მოდულის გამოყენებას.

განვიხილოთ მოდულის მუშაობა მაგალითზე:

```
import csv
FILENAME = "users.csv"
users = [
    ["Tom", 28],
    ["Alice", 23],
    ["Bob", 34]
]
with open(FILENAME, "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerows(users)
with open(FILENAME, "a", newline="") as file:
    user = ["Sam", 31]
    writer = csv.writer(file)
    writer.writerow(user)
```

ფაილში ჩაიწერება ორგანზომილებიანი სია - ფაქტობრივად ცხრილი, სადაც ცალკეული სტრიქონი წარმოადგენს ერთ მომხმარებელს. ხოლო ცალკეული მომხმარებელი შეიცავს ორ ველს - სახელი და ასაკი. ფაქტობრივად ცხრილი შედგება სამი სტრიქონისა და ორი სვეტისგან.

ფაილის ჩასაწერად გახსნისას მესამე პარამეტრის სახით მიეთითება `newline=""` მნიშვნელობა - ცარიელი სტრიქონი საშუალებას იძლევა ფაილიდან კორექტულად იქნას წაკითხული სტრიქონები ოპერაციული სისტემისგან დამოუკიდებლად.

ჩაწერისთვის ჩვენ გვინდა მივიღოთ `writer` ობიექტი, რომელიც დაბრუნდება `csv.writer(file)` ფუნქციის მიერ. ამ ფუნქციას გადაეცემა გახსნილი ფაილი. ხოლო თვით ჩაწერა ხდება `writer.writerows(users)` მეთოდით. ეს მეთოდი მიიღებს სტრიქონების ნაკრებს. ჩვენს შემთხვევაში ეს არის ორგანზომილებიანი სია.

თუ საჭიროა ერთი ჩანაწერის დამატება, რომელიც წარმოადგენს ერთგანზომილებიან სიას, მაგალითად, `["Sam", 31]`, მაშინ ამ შემთხვევაში შეიძლება გამოძახებულ იქნას `writer.writerow(user)` მეთოდი.

საბოლოოდ, სკრიპტის შესრულების შემდეგ იმავე საქალაქში აღმოჩნდება `users.csv` ფაილი, რომელსაც ექნება შემდეგი შემადგენლობა:

Tom,28

Alice,23

Bob,34

Sam,31

ფაილიდან წაკითხვისთვის პირიქით დაგვჭირდება შევექმნათ `reader` ობიექტი:

```
import csv
```



```

FILENAME = "users.csv"
with open(FILENAME, "r", newline="") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row[0], " - ", row[1])

```

reader ობიექტის მიღებისას შეგვიძლია ციკლში გადავარჩიოთ მისი ყველა სტრიქონი:

```

Tom - 28
Alice - 23
Bob - 34
Sam - 31

```

განხილულ მაგალითში თითოეული ჩანაწერი ან სტრიქონი წარმოადგენდა ცალკეულ სიას, მაგალითად, ["Sam", 31]. გარდა ამისა csv მოდულს აქვს სპეციალური დამატებითი შესაძლებლობები ლექსიკონებთან სამუშაოდ. კერძოდ, csv.DictWriter() ფუნქცია აბრუნებს writer ობიექტს, რომელიც ფაილში ჩაწერის საშუალებას იძლევა. csv.DictReader() ფუნქცია აბრუნებს reader ობიექტს ფაილიდან წასაკითხად. მაგალითად:

```

import csv
FILENAME = "users.csv"
users = [
    {"name": "Tom", "age": 28},
    {"name": "Alice", "age": 23},
    {"name": "Bob", "age": 34}
]

```

```

with open(FILENAME, "w", newline="") as file:
    columns = ["name", "age"]
    writer = csv.DictWriter(file, fieldnames=columns)
    writer.writeheader()
    # რამდენიმე სტრიქონის ჩაწერა
    writer.writerows(users)
    user = {"name": "Sam", "age": 41}
    # ერთი სტრიქონის ჩაწერა
    writer.writerow(user)
with open(FILENAME, "r", newline="") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row["name"], "-", row["age"])

```

სტრიქონის ჩაწერა ასევე შეიძლება `writerow()` და `writerows()` მეთოდების დახმარებით. ამ შემთხვევაში თითოეული სტრიქონი წარმოადგენს ცალკეულ ლექსიკონს და გარდა ამისა, ხდება სვეტების სათაურების ჩაწერა `writeheader()` მეთოდის დახმარებით, ხოლო `csv.DictWriter` მეთოდში მეორე პარამეტრის სახით გადაეცემა სვეტების ნაკრები.

სტრიქონების წაკითხვისას, სვეტების დასახელებების გამოყენებით, შეგვიძლია მივმართოთ სტრიქონების შიგნით ცალკეულ მნიშვნელობებს: `row["name"]`.

8.5. ბინარული ფაილები

ბინარული ფაილები ტექსტურისგან განსხვავებით ინახავენ ინფორმაციას ბაიტების ნაკრების სახით. მასთან სამუშაოდ Python-ში საჭიროა ჩამენებული მოდული **pickle**. ეს მოდული წარადგენს ორ მეთოდს:

- **dump(obj, file)** - ჩაწერს obj ობიექტს ბინარულ file ფაილში;
- **load(file)** - კითხულობს მონაცემებს ბინარული ფაილიდან ობიექტში.

ბინარული ფაილის წასაკითად ან ჩასაწერად გახსნისას უნდა გავითვალისწინოთ, რომ ჩვენ გვჭირდება გამოვიყენოთ "b" რეჟიმი ჩაწერის ("w") ან წაკითხვის ("r") რეჟიმთან დამატებით.

დავუშვათ, საჭიროა ორი ობიექტის შენახვა:

```
import pickle
FILENAME = "user.dat"
name = "თომა"
age = 19
with open(FILENAME, "wb") as file:
    pickle.dump(name, file)
    pickle.dump(age, file)
with open(FILENAME, "rb") as file:
    name = pickle.load(file)
    age = pickle.load(file)
    print("სახელი:", name, "\tასაკი:", age)
```

dump ფუნქციის დახმარებით თანმიმდევრულად ჩაიწერება ორი ობიექტი. ამიტომ ფაილის წაკითხვისას

ასევე, თანმიმდევრულად load ფუნქციის მეშვეობით შეგვიძლია წავიკითხოთ ეს ობიექტები.

პროგრამის შედეგი:

სახელი: თომა ასაკი: 28

მსგავსად შეგვიძლია შევინახოთ და ფაილიდან ამოვიღოთ ობიექტების ნაკრები:

```
import pickle
```

```
FILENAME = "users.dat"
```

```
users = [
```

```
    ["თომა", 28, True],
```

```
    ["ალისა", 23, False],
```

```
    ["რობი", 34, False]
```

```
]
```

```
with open(FILENAME, "wb") as file:
```

```
    pickle.dump(users, file)
```

```
with open(FILENAME, "rb") as file:
```

```
    users_from_file = pickle.load(file)
```

```
    for user in users_from_file:
```

```
        print("სახელი:", user[0], "\tასაკი:", user[1],
```

```
              "\tდაქორწინებული:", user[2])
```

იმის მიხედვით თუ რომელი ობიექტი ჩავწერეთ dump ფუნქციით, იგივე ობიექტი დაბრუნდება load ფუნქციით ფაილის წაკითხვისას.

პროგრამის შედეგი:

```

== RESTART: C:/Users/Guliko/AppData/Local/Programs/Python/
სახელი: თომა      ასაკი: 28      დაქორწინებული: True
სახელი: ალისა     ასაკი: 23      დაქორწინებული: False
სახელი: რობი     ასაკი: 34      დაქორწინებული: False
>>>

```

8.5.1. shelve მოდული

ბინარულ ფაილებთან სამუშაოდ Python-ში შეიძლება გამოყენებულ იქნას კიდევ ერთი მოდული - shelve. იგი ინახავს ფაილში ობიექტებს განსაზღვრული გასაღებით. შემდეგ ამ გასაღების მიხედვით შეუძლია ამოიღოს ადრე შენახული ობიექტი ფაილიდან. shelve მოდულით მონაცემებთან მუშაობის პროცესი გვაგონებს ლექსიკონებთან მუშაობას, რომლებიც ასევე იყენებენ გასაღებებს ობიექტების შესანახად და ამოსაღებად.

ფაილის გასახსნელად shelve მოდული იყენებს open() ფუნქციას:

```

open(გზა_ფაილამდე[,flag="c"[,protocol=None[,
writeback=False]]])

```

სადაც flag პარამეტრმა შეიძლება მიიღოს მნიშვნელობები:

- c: ფაილის გაიხსნება წასაკითხად და ჩასაწერად (მნიშვნელობა დუმილით). თუ ფაილი არ არსებობს, მაშინ იგი შეიქმნება;
- r: ფაილი გაიხსნება მხოლოდ წასაკითხად;
- w: ფაილი გაიხსნება ჩასაწერად;

- n: ფაილი გაიხსნება ჩასაწერად. თუ იგი არ არსებობს, მაშინ შეიქმნება. თუ ფაილი არსებობს, მაშინ მას გადაეწერება.

ფაილთან კავშირის დასახურად გამოიძახება `close()` მეთოდი:

```
import shelve
d = shelve.open(filename)
d.close()
```

ფაილი შეიძლება გაიხსნას `with` ოპერატორის დახმარებით. შევინახოთ და წავიკითხოთ ფაილიდან რამდენიმე ობიექტი:

```
import shelve
FILENAME = "states2"
with shelve.open(FILENAME) as states:
    states["London"] = "Great Britain"
    states["Paris"] = "France"
    states["Tbilisi"] = "Georgia"
    states["Madrid"] = "Spain"
with shelve.open(FILENAME) as states:
    print(states["London"])
    print(states["Tbilisi"])
```

მონაცემთა ჩაწერა ითვალისწინებს მნიშვნელობის დაყენებას განსაზღვრული გასაღებისთვის:

```
states["London"] = "Great Britain"
```

ხოლო ფაილიდან წაკითხვა, გასაღების მიხედვით მნიშვნელობის მიღების ექვივალენტურია.:

```
print(states["London"])
```

გასაღების სახით გამოიყენება სტრიქონული მნიშვნელობები.

მონაცემთა წაკითხვისას, თუ მოთხოვნილი გასაღები არ არის, მაშინ გენერირდება გამონაკლისი. ამ შემთხვევაში წინასწარ შეგვიძლია გადავამოწმოთ გასაღების არსებობა **in** ოპერატორის მეშვეობით:

```
with shelve.open(FILENAME) as states:  
    key = "Brussels"  
    if key in states:  
        print(states[key])
```

შეგვიძლია გამოვიყენოთ `get()` მეთოდი. მეთოდის პირველი პარამეტრია - გასაღები, რომლის მიხედვით უნდა იქნას მიღებული მნიშვნელობა, მეორე - მნიშვნელობა დუმილით, რომელიც დაბრუნდება თუ გასაღები არ იქნება ნაპოვნი.

```
with shelve.open(FILENAME) as states:  
    state = states.get("Brussels", "Undefined")  
    print(state)
```

for ციკლის გამოყენებით შეიძლება ფაილიდან ყველა მნიშვნელობის გადარჩევა:

```
with shelve.open(FILENAME) as states:  
    for key in states:  
        print(key, " - ", states[key])
```

keys() მეთოდი აბრუნებს ფაილიდან ყველა გასაღებს, ხოლო values() მეთოდი - ყველა მნიშვნელობას:

```
with shelve.open(FILENAME) as states:
```

```
    for city in states.keys():
```

```
        print(city, end=" ")    # London Paris Tbilisi Madrid
```

```
    print()
```

```
    for country in states.values():
```

```
        print(country, end=" ")#Great Britain France Georgia
```

Spain

შეიძლება გამოვიყენოთ **items()** მეთოდი, რომელიც აბრუნებს კორტეჟების ნაკრებს. ცალკეული კორტეჟი შეიცავს გასაღებს და მნიშვნელობას.

```
with shelve.open(FILENAME) as states:
```

```
    for state in states.items():
```

```
        print(state)
```

გამოიტანს შედეგს:

```
("London", "Great Britain")
```

```
("Paris", "France")
```

```
("Tbilisi", "Georgia")
```

```
("Madrid", "Spain")
```

მონაცემთა განახლება

მონაცემთა შეცვლისთვის საკმარისია გასაღების მიხედვით მინიჭებულ იქნას ახალი მნიშვნელობა, ხოლო მონაცემთა დასამატებლად უნდა განისაზღვროს ახალი გასაღები:

```
import shelve
```



```

FILENAME = "states2"
with shelve.open(FILENAME) as states:
    states["London"] = "Great Britain"
    states["Paris"] = "France"
    states["Tbilisi"] = "Georgia"
    states["Madrid"] = "Spain"
with shelve.open(FILENAME) as states:
    states["London"] = "United Kingdom"
    states["Brussels"] = "Belgium"
for key in states:
    print(key, " - ", states[key])

```

მიღებული შედეგი:

```

== RESTART: C:/Users/Guliko/App
London - United Kingdom
Paris - France
Tbilisi - Georgia
Madrid - Spain
Brussels - Belgium
>>>

```

მონაცემთა წაშლა

წასაშლელად (ამავდროულად შეიძლება შედეგის მიღებაც) შეიძლება `pop()` ფუნქციის გამოყენება, რომელსაც გადაეცემა ელემენტის გასაღები და მნიშვნელობა დუმილით, თუ გასაღები არ იქნება ნაპოვნი:

```

with shelve.open(FILENAME) as states:
    state = states.pop("London", "NotFound")

```

```
print(state)
```

წასაშლელად, აგრეთვე, შეიძლება `del` ოპერატორის გამოყენება:

```
with shelve.open(FILENAME) as states:
```

```
del states["Madrid"] #ვშლით ობიექტს Madrid გასაღებით
```

ყველა ელემენტის წასაშლელად შეიძლება გამოვიყენოთ `clear()` მეთოდი:

```
with shelve.open(FILENAME) as states:
```

```
states.clear()
```

8.6. OS მოდული და მუშაობა ფაილურ სისტემასთან

კატალოგებთან და ფაილებთან მუშაობის მთელ რიგ შესაძლებლობებს წარადგენს `os` ჩაშენებული მოდული. მართალია იგი შეიცავს მრავალ ფუნქციას, მაგრამ განვიხილოთ მათგან მხოლოდ ძირითადები:

- `mkdir()` - ქმნის ახალ საქალაქს;
- `rmdir()` - წაშლის საქალაქს;
- `rename()` - გადაარქმევს ფაილს;
- `remove()` - წაშლის ფაილს.

საქალაქს შესაქმნელად, როგორც განვიხილეთ, გამოიყენება `mkdir()` ფუნქცია, რომელსაც გადაეცემა გზა შესაქმნელ საქალაქდემდე:

```
import os
```

```
# მიმდინარე სკრიპტზე დამოკიდებული გზა
```

```
os.mkdir("hello")
# აბსოლუტური გზა
os.mkdir("c://somedir")
os.mkdir("c://somedir/hello")
```

საქალაქდეს წასაშლელად გამოიყენება `rmdir()` ფუნქცია, რომელსაც გადაეცემა გზა წასაშლელ საქალაქდემდე:

```
import os
# მიმდინარე სკრიპტზე დამოკიდებული გზა
os.rmdir("hello")
# აბსოლუტური გზა
os.rmdir("c://somedir/hello")
```

ფაილის გადასარქმევად გამოიძახება `rename(source, target)` ფუნქცია, პირველი პარამეტრი არის გზა ფაილამდე, ხოლო მეორე - ფაილის ახალი სახელი. მისამართი (გზა) შეიძლება იყოს როგორც აბსოლუტური, ასევე დამოკიდებული. მაგალითად, `C://SomeDir/` საქალაქდემი განთავსებულია `somefile.txt` ფაილი, დავარქვათ მას ახალი სახელი - `hello.txt`:

```
import os
os.rename("C://SomeDir/somefile.txt",
"C://SomeDir/hello.txt")
```

ფაილის წასაშლელად გამოიძახება `remove()` ფუნქცია, რომელსაც გადაეცემა გზა ფაილამდე:

```
import os
```

```
os.remove("C://SomeDir/hello.txt")
```

თუ შევეცდებით გავხსნათ ფაილი, რომელიც არ არსებობს, მაშინ Python გამოსცემს გამონაკლისს - FileNotFoundError. თავის დაზღვევის მიზნით შეგვიძლია გამოვიყენოთ კონსტრუქცია try...except. ფაილის გახსნამდე შეიძლება შემოწმდეს არსებობს თუ არა იგი os.path.exists(path) მეთოდის დახმარებით. ამ მეთოდს გადაეცემა გზა, რომელიც აუცილებელია შემოწმდეს:

```
filename = input("შეიტანეთ გზა ფაილისკენ: ")
if os.path.exists(filename):
    print("მითითებული ფაილი არსებობს")
else:
    print("ფაილი არ არსებობს")
```

IX თავი. მოდულებთან მუშაობა

9.1. მოდულის შექმნა, ჩართვა import და from ინსტრუქციებით

Python ენაში მოდული ეწოდება ნებისმიერ ფაილს პროგრამული კოდით. თითოეულ პროგრამას შეუძლია იმპორტირება გაუკეთოს მოდულს და მიიღოს წვდომა მის კლასებთან, ფუნქციებთან და ობიექტებთან. უნდა აღინიშნოს, რომ მოდული შეიძლება იყოს დაწერილი არამარტო Python-ზე, არამედ, მაგალითად, C ან C++ ენებზე.

9.1.1. მოდულის ჩართვა სტანდარტული ბიბლიოთეკიდან

მოდულის ჩართვა შეიძლება import ინსტრუქციის დახმარებით. მაგალითად, ჩავრთოთ მოდული os მიმდინარე კატალოგის მისაღებად:

```
>>> import os  
>>> os.getcwd()
```

შედეგი:

```
'C:\\Users\\Guliko\\AppData\\Local\\Programs\\Python\\Python  
36-32'
```

import გასაღებური სიტყვის შემდეგ მიეთითება მოდულის სახელი. ერთი ინსტრუქციით შეიძლება ჩავრთოთ რამდენიმე მოდული, მაგრამ ეს არ არის რეკომენდებული, რადგან ადაბლებს კოდის წაკითხვა-

დობას. იმპორტირება გავუკეთოთ time და random მოდულებს.

```
>>> import time, random
>>> time.time()
1376047104.056417
>>> random.random()
0.9874550833306869
```

მოდულების იმპორტირების შემდეგ მისი სახელი გახდება ცვლადი, რომლის მეშვეობითაც შეიძლება წვდომის მიღება მოდულის ატრიბუტებთან. მაგალითად, შეიძლება მივმართოთ e კონსტანტას, რომელიც განთავსებულია math მოდულში:

```
>>> import math
>>> math.e
2.718281828459045
```

თუ მოდულის მითითებული ატრიბუტი არ იქნება ნაპოვნი, წარმოიშვება AttributeError გამონაკლისი, ხოლო თუ არ მოხერხდება მოდულის პოვნა იმპორტირებისთვის, მაშინ - ImportError:

```
>>> import notexist
Traceback (most recent call last):
  File "", line 1, in
    import notexist
ImportError: No module named 'notexist'
>>> import math
>>> math.E
```

Traceback (most recent call last):

File "", line 1, in
math.E

AttributeError: 'module' object has no attribute 'E'

9.1.2. ფსევდონიმების გამოყენება

თუ მოდულის სახელი ძალიან გრძელია ან თუ არ მოგვწონს სხვადასხვა მიზეზების გამო, მაშინ მისთვის შეიძლება ფსევდონიმის შექმნა, as გასაღებური სიტყვის დახმარებით.

```
>>> import math as m  
>>> m.e  
2.718281828459045
```

ამ შემთხვევაში math მოდულის ყველა ატრიბუტთან წვდომა ხორციელდება მხოლოდ m ცვლადის დახმარებით, ხოლო math ცვლადი ამ პროგრამაში უკვე აღარ იქნება. რათქმაუნდა ამის შემდეგ პროგრამაში თუ დავწერთ import math, მაშინ მოდული იქნება წვდომადი როგორც m, ასევე math სახელით.

9.1.3. from ინსტრუქცია

მოდულის განსაზღვრული ატრიბუტების ჩართვა შეიძლება from ინსტრუქციით. მას აქვს რამდენიმე ფორმატი:

```
from <მოდულის სახელი> import <ატრიბუტი 1> [ as  
<ფსევდონიმი 1> ], [<ატრიბუტი 2>[ as <ფსევდონიმი 2>] ...]
```

```
from <მოდულის სახელი> import *
```

პირველი ფორმატი უზრუნველყოფს ჩაერთოს მოდულიდან მხოლოდ თქვენ მიერ მითითებული ატრიბუტები. გრძელი სახელებისთვის შეიძლება ფსევდონიმების დანიშვნა, რომელიც უნდა მიეთითოს as გასაღებური სიტყვის შემდეგ.

```
>>> from math import e, ceil as c
```

```
>>> e
```

```
2.718281828459045
```

```
>>> c(4.6)
```

```
5
```

იმპორტირებული ატრიბუტი თუ ბევრია, შეიძლება განთავსდეს რამდენიმე სტრიქონზე, კოდის უკეთ წაკითხვის თვალსაზრისით:

```
>>> from math import (sin, cos,
```

```
...     tan, atan)
```

from ინსტრუქციის შემდეგი ფორმატი მოდულიდან თითქმის ყველა ცვლადის ჩართვის საშუალებას იძლევა. მაგალითისთვის გავაკეთოთ sys მოდულის ყველა ატრიბუტის იმპორტირება:

```
>>> from sys import *
```

```
>>> version
```

```
'3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC  
v.1600 32 bit (Intel)]'
```

```
>>> version_info
```



```
sys.version_info(major=3,      minor=3,      micro=2,
releaselevel='final', serial=0)
```

უნდა აღინიშნოს, რომ ყველა ატრიბუტი არ იქნება იმპორტირებული. თუ მოდულში განსაზღვრულია `__all__` ცვლადი, მაშინ ჩაერთვება მხოლოდ ამ სიის ატრიბუტები. თუ `__all__` ცვლადი არ არის განსაზღვრული, მაშინ ჩაერთვება ყველა ატრიბუტი, რომელიც არ იწყება ქვედა ხაზგასმით. გარდა ამისა, უნდა იქნას გათვალისწინებული, რომ მოდულიდან ყველა ატრიბუტის იმპორტირებამ შეიძლება დაარღვიოს მთავარი პროგრამის სახელთა სივრცე, რამდენადაც ერთნაირსახელებიანი ცვლადები ერთმანეთზე გადაეწერებიან.

9.1.4. საკუთარი მოდულის შექმნა

შექმნათ `mymodule.py` ფაილი, რომელშიც განვსაზღვროთ რაიმე ფუნქციები:

```
def hello():
    print('Hello, world!')
def fib(n):
    a = b = 1
    for i in range(n - 2):
        a, b = b, a + b
    return b
```

იმავე საქალაქში შექმნათ სხვა ფაილი, მაგალითად, `main.py`:

```
import mymodule
```

```
mymodule.hello()
print(mymodule.fib(10))
```

პროგრამის შედეგი:

Hello, world!

55

გაითვალისწინეთ, რომ მოდულის სახელი არ უნდა ემთხვეოდეს გასაღებურ სიტყვას და არ უნდა იწყებოდეს ციფრით. მოდული შეიძლება შეინახოთ იმავე საქალაქში, სადაც არის ძირითადი პროგრამა ან კატალოგში, სადაც არის დაყენებული python. მოდულის პეზის გზები მითითებულია sys.path ცვლადში.

9.2. ძირითადი ჩაშენებული მოდულები

Python ენას აქვს მთელი რიგი ჩაშენებული მოდულები, რომლებიც შეგვიძლია გამოვიყენოთ პროგრამებში. განვიხილოთ მათგან ძირითადები.

9.2.1. random მოდული

random მოდული მართავს შემთხვევითი რიცხვების გენერირებას, მისი ძირითადი ფუნქციებია:

- random() - აგენერირებს შემთხვევით რიცხვს 0.0 -დან 1.0-მდე;
- randint() - აბრუნებს შემთხვევით რიცხვს განსაზღვრული დიაპაზონიდან;
- randrange() - აბრუნებს შემთხვევით რიცხვს რიცხვთა განსაზღვრული ნაკრებიდან;

- shuffle() - გადააადგილებს სიას;
- choice() - აბრუნებს სიის შემთხვევით ელემენტს.

random() ფუნქცია, როგორც ავლნიშნეთ, აბრუნებს მცურავწერტილიან შემთხვევით რიცხვებს 0.0 -დან 1.0-მდე შუალედში. თუ საჭიროა რიცხვები უფრო დიდი დიაპაზონიდან, მაგალითად 0-დან 100-მდე, შეგვიძლია, შესაბამისად, random() ფუნქციის შედეგი გავამრავლოთ 100-ზე.

```
import random
number = random.random() # მნიშვნელობა 0.0-დან 1.0-
მდე
print(number)
number = random.random() * 100 # მნიშვნელობა 0.0-დან
100.0-მდე
print(number)
```

randint(min, max) აბრუნებს შემთხვევით მთელ რიცხვს min და max მნიშვნელობების შუალედში.

```
import random
number = random.randint(20, 35) # მნიშვნელობა 20-
დან 35-მდე
print(number)
```

randrange() ფუნქციას, რომელიც აბრუნებს შემთხვევით მთელ რიცხვს რიცხვების განსაზღვრული ნაკრებიდან, აქვს სამი ფორმა:

- `randrange(stop)` - რიცხვთა ნაკრების სახით, რომელთაგან უნდა მოხდეს შემთხვევითი მნიშვნელობის ამოღება, გამოყენებული იქნება დიაპაზონი 0-დან `stop` რიცხვამდე;
- `randrange(start, stop)` - რიცხვთა ნაკრები წარმოადგენს დიაპაზონს `start` რიცხვიდან `stop` რიცხვამდე;
- `randrange(start, stop, step)` - რიცხვთა ნაკრები წარმოადგენს დიაპაზონს `start` რიცხვიდან `stop` რიცხვამდე, თანაც ცალკეული რიცხვი დიაპაზონში განსხვავდება წინასგან `step` ბიჯით.

```
import random
```

```
number = random.randrange(10) # მნიშვნელობა [0 – 10]
```

```
print(number)
```

```
number = random.randrange(2, 10) # მნიშვნელობა 2, 3, 4, 5, 6, 7, 8, 9, 10 დიაპაზონში
```

```
print(number)
```

```
number = random.randrange(2, 10, 2) # მნიშვნელობა 2, 4, 6, 8, 10 დიაპაზონში
```

```
print(number)
```

სიებთან სამუშაოდ `random` მოდულში განსაზღვრულია ორი ფუნქცია: `shuffle()` ფუნქცია გადაადგილებს სიას შემთხვევითი სახით, ხოლო `choice()` ფუნქცია აბრუნებს სიიდან ერთ შემთხვევით ელემენტს:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8]
```

```
random.shuffle(numbers)
```

```
print(numbers) # [2, 6, 1, 7, 3, 8, 5, 4]
```

```
random_number = random.choice(numbers)
```

```
print(random_number)
```

9.2.2. math მოდული

math ჩაშენებული მოდული Python-ში წარადგენს ფუნქციების ნაკრებს მათემატიკური, ტრიგონომეტრიული და ლოგარითმული ოპერაციების შესასრულებლად. განვიხილოთ ძირითადი ფუნქციებიდან ზოგიერთი:

- `pow(num, power)` - აბრუნებს `num` რიცხვს `power` ხარისხში;
- `sqrt(num)` - `num` რიცხვის კვადრატული ფესვი;
- `ceil(num)` - რიცხვის დამრგვალება უახლოეს უდიდეს მთელ რიცხვამდე;
- `floor(num)` - რიცხვის დამრგვალება უახლოეს უმცირეს მთელ რიცხვამდე;
- `factorial(num)` - რიცხვის ფაქტორიალი;
- `degrees(rad)` - რადიანიდან გრადუსში გადაყვანა;
- `radians(grad)` - გრადუსიდან რადიანში გადაყვანა;
- `cos(rad)` - კუთხის კოსინუსი რადიანებში;
- `sin(rad)` - კუთხის სინუსი რადიანებში;
- `tan(rad)` - კუთხის ტანგენსი რადიანებში;
- `acos(rad)` - კუთხის არკკოსინუსი რადიანებში;
- `asin(rad)` - კუთხის არკსინუსი რადიანებში;
- `atan(rad)` - კუთხის არკტანგენსი რადიანებში;
- `log(n, base)` - `n` რიცხვის ლოგარითმი `base` ფუძით;
- `log10(n)` - `n` რიცხვის ათობითი ლოგარითმი.

ზოგიერთი ფუნქციის გამოყენების მაგალითი:

```
import math
```

```

# 2 -ის 3 ხარისხი
n1 = math.pow(2, 3)
print(n1) # 8
# იგივე ოპერაცია შეიძლება ასეც შესრულდეს
n2 = 2**3
print(n2)
# ფესვის ამოღება
print(math.sqrt(9)) # 3
# უახლოესი უდიდესი მთელი რიცხვი
print(math.ceil(4.56)) # 5
# უახლოესი უმცირესი მთელი რიცხვი
print(math.floor(4.56)) # 4
# რადიანიდან გრადუსში გადაყვანა
print(math.degrees(3.14159)) # 180
# გრადუსიდან რადიანში გადაყვანა
print(math.radians(180)) # 3.1415.....
# კოსინუსი
print(math.cos(math.radians(60))) # 0.5
# სინუსი
print(math.sin(math.radians(90))) # 1.0
# ტანგენსი
print(math.tan(math.radians(0))) # 0.0
# ლოგარითმი
print(math.log(8,2)) # 3.0
# ათობითი ლოგარითმი
print(math.log10(100)) # 2.0

```

math მოდული, აგრეთვე, წარადგენს მთელ რიგ ჩაშენებულ მუდმივებს, როგორცაა PI და E:

```
import math
radius = 30
# 30 რადიუსიანი წრის ფართობი
area = math.pi * math.pow(radius, 2)
print(area)
# 10-ის ნატურალური ლოგარითმი
number = math.log(10, math.e)
print(number)
```

9.2.3. locale მოდული

რიცხვების ფორმატირებისას Python დუმილით იყენებს ანგლოსაქსურ სისტემას, რომლის დროსაც მთელი რიცხვის თანრიგები ერთმანეთისგან გამოიყოფა მძიმეებით, ხოლო წილადი ნაწილი მთელისგან გამოიყოფა წერტილით. ევროპის კონტინენტზე, მაგალითად, გამოიყენება სხვა სისტემა, რომლის დროსაც თანრიგები გამოიყოფა წერტილით, ხოლო წილადი და მთელი ნაწილი - მძიმით:

```
# ანგლოსაქსური სისტემა
1,234.567
# ევროპული სისტემა
1.234,567
```

ფორმატირების პრობლემის გადასაწყვეტად Python-ში არის ჩაშენებული locale მოდული.

ლოკალური კულტურის დასაყენებლად locale მოდულში განსაზღვრულია setlocale() ფუნქცია. იგი იღებს ორ პარამეტრს:

setlocale(category, locale)

პირველი პარამეტრი მიუთითებს კატეგორიაზე, რომლისთვისაც გამოიყენება ფუნქცია - რიცხვებისთვის, ვალუტისთვის ან ორივესთვის ერთად. პარამეტრის მნიშვნელობის სახით შეგვიძლია გადავცეთ შემდეგი კონსტანტებიდან ერთ-ერთი:

LC_ALL - ლოკალიზება ვრცელდება ყველა კატეგორიაზე - რიცხვების, ვალუტის, თარიღის და ა.შ. ფორმატირებაზე;

LC_NUMERIC-ლოკალიზება ვრცელდება რიცხვებზე;

LC_MONETARY-ლოკალიზება ვრცელდება ვალუტაზე;

LC_TIME-ლოკალიზება ვრცელდება თარიღსა და დროზე;

LC_CTYPE-ლოკალიზება სიმბოლოების ზედა ან ქვედა რეგისტრში გადაყვანისას;

LC_COLLATE - ლოკალიზება სტრიქონების შედარებისას.

setlocale ფუნქციის მეორე პარამეტრი მიუთითებს ლოკალურ კულტურაზე, რომელიც უნდა იქნას გამოყენებული. OC Windows-თვის შეიძლება გამოვიყენოთ ქვეყნის კოდი ISO მიხედვით, ორი სიმბოლოსგან შედგენილი. მაგალითად: Georgia –ge, Usa –

us, Russian –ru, ხოლო MacOS სისტემისთვის საჭიროა მიეთითოს ენის კოდი და ქვეყნის კოდი, მაგალითად: ინგლისურისთვის ამერიკაში - en_US, რუსულისთვის რუსეთში - ru_RU, ქართულისთვის საქართველოში - ge-Ge, გერმანულისთვის გერმანიაში - de_DE. დუმილით ფაქტობრივად გამოიყენება კულტურა en_US.

უშუალოდ რიცხვისა და ვალუტის ფორმატირებისთვის locale მოდული წარადგენს ორ ფუნქციას:

currency(num) - აფორმატებს ვალუტას;

format(str, num) – num რიცხვი გადაჰყავს str სტრიქონში. გადაყვანისას რიცხვის ტიპიდან გამომდინარე გამოიყენება შემდეგი სიმბოლოები:

- d - მთელი რიცხვებისთვის;
- f - მცურავწერტილიანი რიცხვებისთვის;
- e - რიცხვების ექსპონენციალური ჩანაწერისთვის.

აღნიშნული სიმბოლოს წინ იწერება % - პროცენტის ნიშანი, მაგალითად: "%d".

წილადი რიცხვების გამოტანისას f სიმბოლოს წინ წერტილის შემდეგ შეიძლება მიეთითოს წილად ნაწილში თუ რამდენი ნიშანი უნდა აისახოს: %.2f # ორი ნიშანი წილად ნაწილში.

განვიხილოთ რიცხვების და ვალუტის ლოკალიზება ამერიკულისთვის:

```
import locale
```

```
locale.setlocale(locale.LC_ALL, "us") #Windows-თვის
```

```
# locale.setlocale(locale.LC_ALL, "en_US")# MacOS-თვის
```

```

number = 12345.6789
formatted = locale.format("%f", number)
print(formatted) # 12345,678900
formatted = locale.format("%.2f", number)
print(formatted) # 12345,68
formatted = locale.format("%d", number)
print(formatted) # 12345
formatted = locale.format("%e", number)
print(formatted) # 1,234568e+04
money = 234.678
formatted = locale.currency(money)
print(formatted) # 234,68 €

```

თუ კონკრეტული კოდის ნაცვლად მეორე პარამეტრის სახით გადაეცემა ცარიელი სტრიქონი, მაშინ Python გამოიყენებს კულტურას, რომელიც გამოიყენება მიმდინარე სამუშაო კომპიუტერზე, ხოლო `getlocale()` ფუნქციის დახმარებით შეიძლება ამ კულტურის მიღება:

```

import locale
locale.setlocale(locale.LC_ALL, "")
number = 12345.6789
formatted = locale.format("%.02f", number)
print(formatted) #12345.68
print(locale.getlocale()) #('English_United States', '1252')

```

9.2.4. decimal მოდული

მცურავწერტილიან რიცხვებთან მუშაობის დროს არ მიიღება მთლად ზუსტი შედეგი:

```
number = 0.1 + 0.1 + 0.1
```

```
print(number) # 0.30000000000000004
```

პრობლემა შეიძლება გადავწყვიტოთ `round()` ფუნქციის გამოყენებით, რომელიც დაამრგვალებს რიცხვს. არის სხვა ხერხიც, რომელიც მდგომარეობს ჩაშენებული `decimal` მოდულის გამოყენებაში.

ამ მოდულში რიცხვებთან სამუშაოდ გასაღებური კომპონენტი არის `Decimal` კლასი. მისი გამოყენებისთვის უნდა შეიქმნას მისი ობიექტი კონსტრუქტორის გამოყენებით. კონსტრუქტორში გადაეცემა სტრიქონული მნიშვნელობა, რომელიც წარმოადგენს რიცხვს:

```
from decimal import Decimal
```

```
number = Decimal("0.1")
```

ამის შემდეგ `Decimal` ობიექტის გამოყენება შეიძლება არითმეტიკულ ოპერაციებში:

```
from decimal import Decimal
```

```
number = Decimal("0.1")
```

```
number = number + number + number
```

```
print(number) # 0.3
```

`Decimal`-თან ოპერაციებში შეიძლება მთელი რიცხვების გამოყენება:

```
number = Decimal("0.1")
```

```
number = number + 2
```

არ შეიძლება ოპერაციებში წილადი რიცხვების: float და Decimal არევა:

```
number = Decimal("0.1")
number = number + 0.1 # აქ იქნება შეცდომა
```

დამატებითი ნიშნების დახმარებით შეიძლება განისაზღვროს რამდენი სიმბოლო იქნება რიცხვის წილად ნაწილში:

```
number = Decimal("0.10")
number = 3 * number
print(number) # 0.30
```

სტრიქონი "0.10" განსაზღვრავს ორ ნიშანს წილად ნაწილში, მაშინაც თუ ბოლო სიმბოლოები იქნება ნული. შესაბამისად, "0.100" წარმოადგენს სამ ნიშანს წილად ნაწილში.

9.2.5. quantize() მეთოდი

Decimal ობიექტებს აქვთ quantize() მეთოდი, რომელიც რიცხვის დამრგვალების საშუალებას იძლევა. ამ მეთოდში პირველი არგუმენტის სახით გადაეცემა ასევე, Decimal ობიექტი, რომელიც მიუთითებს რიცხვის დამრგვალების ფორმატს:

```
from decimal import Decimal
number = Decimal("0.444")
number = number.quantize(Decimal("1.00"))
print(number) # 0.44
```

```

number = Decimal("0.555678")
print(number.quantize(Decimal("1.00"))) # 0.56
number = Decimal("0.999")
print(number.quantize(Decimal("1.00"))) # 1.00

```

გამოყენებული სტრიქონი "1.00" მიუთითებს, რომ დამრგვალდება წილად ნაწილში ორი ნიშნით. ღუმლით დამრგვალება აღიწერება ROUND_HALF_EVEN მუდმივით, რომლის დროსაც რიცხვი მრგვალდება მეტობით, თუ იგი კენტია, ხოლო მისი წინამორბედი არის 4. მაგალითად:

```

from decimal import Decimal, ROUND_HALF_EVEN
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"), ROUND_HALF_EVEN))
#10.02
number = Decimal("10.035")
print(number.quantize(Decimal("1.00"),ROUND_HALF_EVEN))
#10.04

```

დამრგვალების სტრატეგია quantize-ში გადაეცემა მეორე პარამეტრის სახით. სტრიქონი "1.00" აღნიშნავს, რომ დამრგვალდება წილადი ნაწილის ორ თანრიგამდე. მაგრამ პირველ შემთხვევაში "10.025" - მეორე ნიშანი არის 2 - ლუწი რიცხვი, ამიტომ მიუხედავად იმისა, რომ შემდეგი ციფრია 5, ორი არ დამრგვალდება სამამდე.

მეორე შემთხვევაში "10.035" - მეორე ნიშნად არის 3 - კენტი რიცხვი, ამიტომ იგი მრგვალდება 4-მდე.

დამრგვალებისას ასეთი ქცევა, შეიძლება არცთუისე სასურველი იყოს. ასეთ შემთხვევაში იგი შეიძლება ხელახლა განისაზღვროს, ჩამოთვლილთაგან ერთ-ერთი კონსტანტის გამოყენებით:

- **ROUND_HALF_UP** - რიცხვის დამრგვალება მეტობით, თუ მის შემდეგ მოდის ციფრი 5 ან მეტი;
- **ROUND_HALF_DOWN**-რიცხვის დამრგვალება ნაკლებობით, როდესაც მის შემდეგ მოდის 5 ან მეტი.

```
number = Decimal("10.026")
print(number.quantize(Decimal("1.00"),ROUND_HALF_DOWN))
#10.03
```

```
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"),ROUND_HALF_DOWN))
#10.02
```

ROUND_05UP - ამრგვალებს მხოლოდ 0-ს 1-მდე, თუ მის შემდეგ მოდის 5:

```
number = Decimal("10.005")
print(number.quantize(Decimal("1.00"), ROUND_05UP))
# 10.01
```

```
number = Decimal("10.025")
print(number.quantize(Decimal("1.00"), ROUND_05UP))
# 10.02
```

ROUND_CEILING - რიცხვს ამრგვალეხს მეტობით, მიუხედავად იმისა, თუ რომელი ციფრი მოდის მის შემდეგ:

```
number = Decimal("10.021")  
print(number.quantize(Decimal("1.00"),ROUND_CEILING))  
# 10.03
```

```
number = Decimal("10.025")  
print(number.quantize(Decimal("1.00"),ROUND_CEILING))  
# 10.03
```

ROUND_FLOOR - არ ამრგვალეხს რიცხვს, მიუხედავად იმისა, თუ რომელი ციფრი მოდის მის შემდეგ:

```
number = Decimal("10.021")  
print(number.quantize(Decimal("1.00"), ROUND_FLOOR))  
# 10.02
```

```
number = Decimal("10.025")  
print(number.quantize(Decimal("1.00"), ROUND_FLOOR))  
# 10.02
```

X თავი

ობიექტზე ორიენტირებული დაპროგრამება

10.1. კლასები და ობიექტები

Python მხარს უჭერს დაპროგრამების ობიექტზე ორიენტირებულ პარადიგმას, რაც ნიშნავს, რომ შესაძლებელია პროგრამის კომპონენტები განისაზღვროს კლასების სახით.

კლასი არის შაბლონი ანუ ობიექტის ფორმალური აღწერა, ხოლო ობიექტი - კლასის ეგზემპლარი, მისი რეალური განხორციელება. კოდის თვალსაზრისით კლასი აერთიანებს ფუნქციების და ცვლადების ნაკრებს, რომლებიც ასრულებენ გარკვეულ ამოცანას. კლასის ფუნქციებს ასევე, უწოდებენ მეთოდებს. ისინი განსაზღვრავენ კლასის ქცევას. კლასის ცვლადებს ჰქვიათ ატრიბუტები - ისინი ინახავენ კლასის მდგომარეობას.

კლასი განისაზღვრება class გასაღებური სიტყვით:

```
class კლასის_სახელი:
```

```
    კლასის_მეთოდები
```

კლასის ობიექტის შესაქმნელად გამოიყენება შემდეგი სინტაქსი:

```
ობიექტის_სახელი = კლასის_სახელი ([პარამეტრები])
```

მაგალითად, განვსაზღვროთ მარტივი Person კლასი, რომელიც წარმოადგენს ადამიანს:

```
class Person:
```

```
    name = "Sandro"
```



```

def display_info(self):
    print("gamarjobaT, me mqvia", self.name)
person1 = Person()
person1.display_info() # gamarjobaT, me mqvia Sandro
person2 = Person()
person2.name = "Anna"
person2.display_info() # gamarjobaT, me mqvia Anna

```

Person კლასი განსაზღვრავს name ატრიბუტს, რომელიც ინახავს ადამიანის სახელს, ხოლო display_info მეთოდს გამოაქვს ინფორმაცია ადამიანის შესახებ.

ნებისმიერი კლასის მეთოდის განსაზღვრისას უნდა გავითვალისწინოთ, რომ ყველა მათგანმა პირველი პარამეტრის სახით უნდა მიიღოს ბმული მიმდინარე ობიექტზე, რომელიც პირობის თანახმად არის self (c++ -ში არის ანალოგი - გასაღებური სიტყვა this). ამ ბმულით კლასის შიგნით შეგვიძლია მივმართოთ ამავე კლასის მეთოდებს ან ატრიბუტებს. კერძოდ, self.name გამოსახულებით შეიძლება მივიღოთ მომხმარებლის სახელი.

Person კლასის განსაზღვრის შემდეგ შევქმნათ მისი ორი ობიექტი: person1 და person2. ობიექტის სახელის გამოყენებით, შეგვიძლია მივმართოთ კლასის მეთოდებს და ატრიბუტებს. მოცემულ შემთხვევაში ორივე ობიექტისთვის ვიძახებთ display_info() მეთოდს, რომელიც სტრიქონს გამოიტანს კონსოლზე, მეორე ობიექტისთვის ცვვლით name ატრიბუტს. ამ დროს display_info მეთოდის

გამოძახებისას არ უნდა მნიშვნელობის გადაცემა self პარამეტრისთვის.

განვიხილოთ მაგალითი. შევქმნათ კლასი მართკუთხედი:

```
class Rectangle:  
    color = "green"  
    width = 100  
    height = 100
```

კლასის ატრიბუტებზე წვდომა შეიძლება მივიღოთ შემდეგი სახით:

```
ობიექტის_სახელი.ატრიბუტი
```

გამოვაცხადოთ მართკუთხედი კლასის ობიექტი და დავბეჭდოთ ამ ობიექტის ატრიბუტი - ფერი:

```
rect1 = Rectangle()  
print(rect1.color)
```

დავამატოთ ჩვენ მიერ შექმნილ კლასს მეთოდი - ეს ფუნქციაა, რომელიც კლასის შიგნით მდებარეობს და ასრულებს განსაზღვრულ სამუშაოს, რომელიც ყველაზე მეტად გვთავაზობს შექმნილი ობიექტის ატრიბუტებთან წვდომას. მაგალითად, Rectangle კლასს დავუმატოთ მეთოდი, რომელიც გამოითვლის მართკუთხედის ფართობს. იმისათვის, რომ მეთოდმა იცოდეს თუ რომელ ობიექტთან მუშაობს, არგუმენტად მას უნდა გადაეცეს self პარამეტრი, რომლითაც იგი მიიღებს წვდომას თავის მონაცემებთან.

```

class Rectangle:
    color = "green"
    width = 100
    height = 100
    def square(self):
        return self.width * self.height

```

პროგრამას ექნება საბოლოო სახე:

```

class Rectangle:
    color = "green"
    width = 100
    height = 100
    def square(self):
        return self.width*self.height

```

```

rect1 = Rectangle()
print(rect1.color)
print(rect1.square())
rect2 = Rectangle()
rect2.width = 200
rect2.color = "brown"
print(rect2.color)
print(rect2.square())

```

პროგრამის შედეგი:

```

== RESTART: C:/Users/Guliko.
green
10000
brown
20000
>>> .

```

10.2. კონსტრუქტორი

კლასის კონსტრუქტორი საშუალებას იძლევა მიეთითოს ობიექტს პარამეტრები მისი შექმნის დროს. ამდენად, ჩნდება შესაძლებლობა შეიქმნას ობიექტები წინასწარ მოცემული ატრიბუტებით. კლასის კონსტრუქტორი არის მეთოდი: `__init__(self)`.

ზემოთ, როცა შევქმენით Person კლასის ობიექტები, გამოვიყენეთ კონსტრუქტორი დუმილით, რომელიც არაცხადად აქვს ყველა კლასს:

```
person1 = Person()
```

```
person2 = Person()
```

კლასში შეიძლება ცხადად განისაზღვროს კონსტრუქტორი სპეციალური მეთოდის დახმარებით, რომელსაც ეწოდება `__init__()`. მაგალითად, შევცვალოთ Person კლასი მასზე კონსტრუქტორის დამატებით:

```
class Person:
```

```
    # კონსტრუქტორი
```

```
    def __init__(self, name):
```

```
        self.name = name # ვაყენებთ სახელს
```

```
    def display_info(self):
```

```
        print("გამარჯობა, მე მქვია", self.name)
```

```
person1 = Person("სანდრო")
```

```
person1.display_info() #გამარჯობა, მე მქვია სანდრო
```

```
person2 = Person("ანა")
```

```
person2.display_info() #გამარჯობა, მე მქვია ანა
```

პირველი პარამეტრის სახით კონსტრუქტორი მიიღებს ბმულს მიმდინარე ობიექტზე - self. ზოგჯერ კონსტრუქტორებში დაყენდება კლასის ატრიბუტები. მოცემულ შემთხვევაში მეორე პარამეტრის სახით კონსტრუქტორში გადაეცემა მომხმარებლის სახელი, რომელიც დაყენდება self.name ატრიბუტისთვის. თანაც ატრიბუტისთვის არ არის აუცილებელი განისაზღვროს კლასში name ცვლადი, როგორც ეს იყო Person კლასის წინა ვერსიაში. self.name = name მნიშვნელობის დაყენება უკვე არაცხადად ქმნის name ატრიბუტს.

```
person1 = Person("სანდრო")
```

```
person2 = Person("ანა")
```

შედეგად მიიღება:

გამარჯობა, მე მქვია სანდრო

გამარჯობა, მე მქვია ანა

ზემოთ განხილულ მაგალითში, მართკუთხედის შექმნის დროს ფერის, სიგრძისა და სიგანის მისათითებლად Rectangle კლასს დავამატოთ შემდეგი კონსტრუქტორი:

```
class Rectangle:
```

```
    def __init__(self,color="green", width=100, height=100):
```

```
        self.color = color
```

```
        self.width = width
```

```
        self.height = height
```

```
    def square(self):
```

```
        return self.width * self.height
```

```

rect1 = Rectangle()
print(rect1.color)
print(rect1.square())
rect1 = Rectangle("yellow", 23, 34)
print(rect1.color)
print(rect1.square())
პროგრამის შედეგი:
...
== RESTART: C:/Users/Guliko/AppData,
green
10000
yellow
782
>>> .

```

10.3. დესტრუქტორი

ობიექტთან მუშაობის დასრულების შემდეგ შეგვიძლია გამოვიყენოთ `del` ოპერატორი მისი მახსიერებიდან ამოშლისთვის:

```

person1 = Person("მართა")
del person1 # მახსიერებიდან წაშლა
# person1.display_info() # ეს მეთოდი არ იმუშავებს,
რადგან person1 უკვე წაშლილია მახსიერებიდან

```

უნდა აღინიშნოს, რომ პრინციპში ამის გაკეთება არააუცილებელია, რამდენადაც სკრიპტის მუშაობის დასრულების შემდეგ ყველა ობიექტი ავტომატურად წაიშლება მახსიერებიდან.

გარდა ამისა, კლასში შეგვიძლია განვსაზღვროთ დესტრუქტორი, `__del__` ჩაშენებული ფუნქციის რეალიზებით, რომელიც გამოიძახება ან `del` ოპერატორის გამოძახების შედეგად ან ობიექტის ავტომატური წაშლისას. მაგალითად:

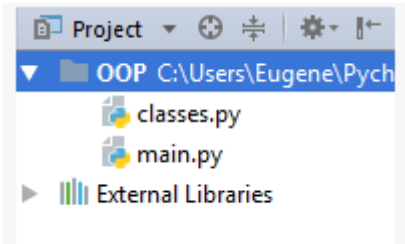
```
class Person:
    # კონსტრუქტორი
    def __init__(self, name):
        self.name = name # ვაყენებთ სახელს
    def __del__(self):
        print(self.name, "წაშლილია მეხსიერებიდან")
    def display_info(self):
        print("გამარჯობა, მე მქვია", self.name)
person1 = Person("თომა")
person1.display_info() # გამარჯობა, მე მქვია თომა
del person1 # წაშლილია მეხსიერებიდან
person2 = Person("ჯეკი")
person2.display_info() # გამარჯობა, მე მქვია ჯეკი
del person2 # წაშლილია მეხსიერებიდან
```

კონსოლზე გამოტანილი შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/Local/I
გამარჯობა, მე მქვია თომა
თომა წაშლილია მეხსიერებიდან
გამარჯობა, მე მქვია ჯეკი
ჯეკი წაშლილია მეხსიერებიდან
>>> .
```

10.4. კლასის განსაზღვრა მოდულში და მოდულის ჩართვა პროგრამაში

როგორც წესი, კლასები განთავსდება ცალკეულ მოდულებში და შემდეგ იმპორტირდებიან პროგრამის ძირითად სკრიპტში. დავუშვათ პროექტში გვაქვს ორი ფაილი: main.py (პროგრამის ძირითადი სკრიპტი) და classes.py (სკრიპტი კლასის განსაზღვრით).



classes.py ფაილში განვსაზღვროთ ორი კლასი:

```
class Person:
```

```
    # კონსტრუქტორი
```

```
    def __init__(self, name):
```

```
        self.name = name # ვაყენებთ სახელს
```

```
    def display_info(self):
```

```
        print("გამარჯობა, მე მქვია", self.name)
```

```
class Auto:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def move(self, speed):
```

```
        print(self.name, "მიდის სიჩქარით", speed, "კმ/სთ")
```


Person კლასთან ერთად განსაზღვრულია კლასი Auto, რომელიც არის მანქანა და აქვს move მეთოდი და name ატრიბუტი. ჩავრთოთ ეს კლასები და გამოვიყენოთ ისინი main.py სკრიპტში:

```
from classes import Person, Auto
tom = Person("Tom")
tom.display_info()
bmw = Auto("BMW")
bmw.move(65)
```

კლასების ჩართვა ხდება ზუსტად ისევე, როგორც ფუნქციები მოდულიდან. შეგვიძლია ჩავრთოთ მთელი მოდული გამოსახულებით:

```
import classes
```

ან ჩავრთოთ ცალკე კლასები, როგორც ზედა მაგალითში:

შედეგად მიიღება:

გამარჯობა, მე მქვია Tom

BMW მიდის სიჩქარით 65 კმ/სთ

10.5. ინკაფსულაცია

კლასებში დუმილით ატრიბუტები არის საერთო წვდომის, რაც ნიშნავს, რომ პროგრამის ნებისმიერი ადგილიდან შეიძლება ობიექტის ატრიბუტის მიღება და მისი შეცვლა. მაგალითად:

```
class Person:
```

```
    def __init__(self, name, age):
```

```

self.name = name # ვაყენებთ სახელს
self.age = age # ვაყენებთ ასაკს
def display_info(self):
    print("სახელი:", self.name, "\tასაკი:", self.age)
tom = Person("Tom", 23)
tom.name="ადამიანი-ობობა" #ვცვლით name ატრიბუტს
tom.age = -129 # ვცვლით age ატრიბუტს
tom.display_info() # სახელი: ადამიანი-ობობა ასაკი: -129

```

გამოიტანს შედეგს:

```

== RESTART: C:/Users/Guliko/AppData/Local/
სახელი: ადამიანი-ობობა ასაკი: -129
>>>

```

მოცემულ შემთხვევაში შეგვიძლია მონაცემებს, მაგალითად, ადამიანის ასაკს ან სახელს მივანიჭოთ არაკორექტული მნიშვნელობა, დავუშვათ, მივუთითოთ უარყოფითი ასაკი. მსგავსი ქცევა არ არის სასურველი, ამიტომ დგება ობიექტის ატრიბუტებზე წვდომის კონტროლის საკითხი.

მოცემულ პრობლემასთან მჭიდრო კავშირშია ინკაფსულაციის ცნება. იგი არის ობიექტზე ორიენტირებული დაპროგრამების ფუნდამენტური კონცეფცია. იგი ხელს უშლის ობიექტის ატრიბუტებზე პირდაპირ წვდომას. ატრიბუტების დასამალად დაპროგრამების Python ენაში ისინი უნდა გამოვაცხადოთ როგორც კერძო ანუ დახურული და შევზღუდოთ წვდომა

მათზე სპეციალური მეთოდებით, რომლებსაც ასევე ჰქვიათ თვისებები.

შევცვალოთ ზემოთ განსაზღვრული კლასი, მათში თვისებების განსაზღვრით:

```
class Person:
    def __init__(self, name, age):
        self.__name = name # ვაყენებთ სახელს
        self.__age = age   # ვაყენებთ ასაკს
    def set_age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("დაუშვებელი ასაკი")
    def get_age(self):
        return self.__age
    def get_name(self):
        return self.__name
    def display_info(self):
        print("სახელი:",self.__name, "\tასაკი:", self.__age)
tom = Person("თომა", 23)
tom.__age = 43      # age ატრიბუტი არ შეიცვლება
tom.display_info() # სახელი: თომა ასაკი: 23
tom.set_age(-3486) # დაუშვებელი ასაკი
tom.set_age(25)
tom.display_info() # სახელი: თომა ასაკი: 25
გამოიტანს შედეგს:
```

```

== RESTART: C:/Users/Guliko/AppData/Local/
სახელი: თომას ასაკი: 23
დაუშვებელი ასაკი
სახელი: თომას ასაკი: 25
>>>

```

ვერძო ატრიბუტის შესაქმნელად დასაწყისში მისი დასახელება ჩაისმება ორმაგ ტირეში: `self.__name`. ასეთ ატრიბუტს შეგვიძლია მივმართოთ მხოლოდ იმავე კლასიდან, მაგრამ ვერ მივმართავთ ამ კლასის გარედან. მაგალითად, მნიშვნელობის მინიჭება ამ ატრიბუტზე არაფერს არ მოგვცემს:

```
tom.__age = 43
```

ხოლო მისი მნიშვნელობის მიღების მცდელობა მიგვიყვანს შესრულების შეცდომამდე:

```
print(tom.__age)
```

შეიძლება დაგვჭირდეს, მომხმარებლის ასაკის გარედან მინიჭება. ამისათვის იქმნება თვისებები. ერთი თვისების გამოყენებით, შეგვიძლია მივიღოთ ატრიბუტის მნიშვნელობა:

```
def get_age(self):
```

```
    return self.__age
```

მოცემულ მეთოდს ხშირად უწოდებენ „გეტერს“ ან აქსესუარს.

ასაკის შესაცვლელად განსაზღვრულია სხვა თვისება:

```
def set_age(self, value):
```

```
    if value in range(1, 100):
```

```
        self.__age = value
```

else:

```
print("დაუშვებელი ასაკი")
```

აქ ჩვენ შეგვიძლია გადავწყვიტოთ პირობის მიხედვით საჭიროა თუ არა ასაკის შეცვლა. მოცემულ მეთოდს ხშირად უწოდებენ „სეტერს“ ან მუტატორს.

ცალკეული კერძო ატრიბუტისთვის არ არის აუცილებელი მსგავსი წყვილი თვისების შექმნა. როგორც, ზედა მაგალითში ადამიანის სახელი შეგვიძლია დავაყენოთ მხოლოდ კონსტრუქტორიდან, ხოლო მიღებისთვის განსაზღვრულია `get_name` მეთოდი.

თვისებების ანოტაციები

ზემოთ ვნახეთ როგორ შევქმნათ თვისება. Python ენას აქვს კიდეც ერთი - თვისებების განსაზღვრის უფრო ელეგანტური ხერხი. ეს ხერხი ითვალისწინებს ანოტაციების გამოყენებას, რომლებიც იწყება `@` სიმბოლოთი.

თვისება-გეტერის შექმნისთვის თვისების წინ ისმება ანოტაცია `@property`.

თვისება-სეტერის შექმნისთვის თვისების წინ ისმება ანოტაცია `სეტერის_თვისების_სახელი.setter`. გადავწეროთ `Person` კლასი ანოტაციების გამოყენებით:

```
class Person:
```

```
    def __init__(self, name, age):
```

```
        self.__name = name # ვაყენებთ სახელს
```

```
        self.__age = age # ვაყენებთ ასაკს
```

```
    @property
```

```

def age(self):
    return self.__age
@age.setter
def age(self, age):
    if age in range(1, 100):
        self.__age = age
    else:
        print("დაუშვებელი ასაკი")
@property
def name(self):
    return self.__name
    def display_info(self):
        print("სახელი:",self.__name, "\tასაკი:", self.__age)
tom = Person("თომა", 23)
tom.display_info()    # სახელი: თომა ასაკი: 23
tom.age = -3486       # დაუშვებელი ასაკი
print(tom.age)       # 23
tom.age = 36
tom.display_info()   # სახელი: თომა ასაკი: 36

```

გამოიტანს შედეგს:

```

== RESTART: C:/Users/Guliko/AppData/Local/
სახელი: თომა ასაკი: 23
დაუშვებელი ასაკი
23
სახელი: თომა ასაკი: 36
>>>

```

მივაქციოთ ყურადღება, რომ თვისება-„სეტერი“ განისაზღვრება თვისება-„გეტერის“ შემდეგ. გარდა ამისა, „სეტერიც“ და „გეტერიც“ სახელდება ერთნაირად - age. რადგანაც „გეტერს“ ჰქვია age, ამიტომ „სეტერის“ წინ დაყენდება ანოტაცია @age.setter. ამის შემდეგ „გეტერსაც“ და „სეტერსაც“ მივმართავთ tom.age გამოსახულებით.

10.6. მემკვიდრეობითობა

მემკვიდრეობითობის ორგანიზებაში მონაწილეობს მინიმუმ ორი კლასი: მშობელი კლასი და მემკვიდრე კლასი. შესაძლებელია მრავლობითი მემკვიდრეობითობაც, რომლის დროსაც მემკვიდრე კლასი მიიღება რამდენიმე მშობელი კლასისგან. მშობელი ანუ საბაზო კლასიდან მემკვიდრე ანუ წარმოებული კლასის შექმნის სინტაქსია:

```
class წარმოებული_კლასის_სახელი(მშობელი_კლასი1,  
[მშობელი_კლასი2, მშობელი_კლასი_ n])
```

ზემოთ განხილულ მაგალითში შემოვიტანოთ მემკვიდრეობითობა:

```
class Figure:  
    def __init__(self, color):  
        self.color = color  
    def get_color(self):  
        return self.color  
class Rectangle(Figure):
```

```

def __init__(self, color, width=100, height=100):
    super().__init__(color)
    self.width = width
    self.height = height
def square(self):
    return self.width*self.height
rect1 = Rectangle("blue")
print(rect1.get_color())
print(rect1.square())
rect2 = Rectangle("red", 25, 70)
print(rect2.get_color())
print(rect2.square())

```

პროგრამის შედეგი:

```

== RESTART: C:/Users/Guliko/AppData/Lc
blue
10000
red
1750
>>>

```

განვიხილოთ მაგალითი, შევექმნათ საბაზო კლასი Animal, ხოლო შემდეგ შევექმნათ მისგან წარმოებული კლასი - Cat, რომელიც გამოიყენებს საბაზო კლასის ატრიბუტსაც:

```

class Animal :
    # Constructor
    def __init__(self, name):
        # Animal კლასს აქვს 1 ატრიბუტი: 'name'.

```



```

        self.name= name
    # მეთოდი (method):
    def showInfo(self):
        print ("I'm " + self.name)
    # მეთოდი (method):
    def move(self):
        print ("moving ...")
    # Cat კლასი მიიღება Animal კლასიდან
class Cat (Animal):
    def __init__(self, name, age, height):
        #გამოიძახება კონსტრუქტორი მშობელი კლასის -
        (Animal)
        #რათა მშობელი კლასის 'name' ატრიბუტს
        მიემაგროს მნიშვნელობა
        super().__init__(name)
        self.age = age
        self.height = height
        #ხელახლა განვსაზღვროთ (override) იგივე
        დასახელების მეთოდი, რაც მშობელ კლასშია
    def showInfo(self):
        print ("I'm " + self.name)
        print (" age " + str(self.age))
        print (" height " + str(self.height))
tom = Cat("Tom", 3, 20)
print ("Call move() method")
tom.move()

```

```
print ("\n")
print ("Call showInfo() method")
tom.showInfo()
```

პროგრამის შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/
Call move() method
moving ...

Call showInfo() method
I'm Tom
  age 3
  height 20
>>>
```

დღემილით, წარმოებული კლასი მემკვიდრეობით იღებს მეთოდებს საბაზო კლასიდან, მაგრამ მეკვიდრე კლასს შეუძლია ხელახლა განსაზღვროს (override) მშობელი კლასის მეთოდები:

```
class Animal :
    # Constructor
    def __init__(self, name):
        # Animal კლასს აქვს 1 ატრიბუტი: 'name'.
        self.name= name
    # მეთოდი (method):
    def showInfo(self):
        print ("I'm " + self.name)
    # მეთოდი (method):
    def move(self):
```

```

    print ("moving ...")
# Mouse კლასი მიიღება Animal კლასიდან
class Mouse (Animal):
    def __init__(self, name, age, height):
#გამოიძახება კონსტრუქტორი მშობელი კლასის -
(Animal)
    #რათა მშობელი კლასის 'name' ატრიბუტს მიემაგროს
მნიშვნელობა
        super().__init__(name)
        self.age = age
        self.height = height
    #ხელახლა განვსაზღვროთ (override) იგივე
დასახელების მეთოდი, რაც მშობელ კლასშია
    def showInfo(self):
        # გამოვიძახოთ მშობელი კლასის მეთოდი
        super().showInfo()
        print (" age " + str(self.age))
        print (" height " + str(self.height))
    #ხელახლა განვსაზღვროთ (override) იგივე
დასახელების მეთოდი, რაც მშობელ კლასშია
    def move(self):
        print ("Mouse moving ...")
jerry = Mouse("Jerry", 3, 5)
print ("Call move() method")
jerry.move()
print ("\n")

```

```
print ("Call showInfo() method")
jerry.showInfo()
```

პროგრამის შედეგი:

```
= RESTART: C:/Users/Guliko/AppData/Local,
Call move() method
Mouse moving ...

Call showInfo() method
I'm Jerry
  age 3
  height 5
>>>
```

10.7. მრავლობითი მემკვიდრეობითობა

Python მხარს უჭერს მრავლობით მემკვიდრეობითობას. წარმოებული კლასი შეიძლება შეიქმნას ორი ან მეტი საბაზო კლასიდან. მშობელ კლასებს შეიძლება ჰქონდეთ ერთნაირი ატრიბუტები ან მეთოდები. წარმოებული კლასი პრიორიტეტულად იღებს მემკვიდრეობითობის სიიდან ატრიბუტებს და მეთოდებს.

განვიხილოთ მაგალითი:

```
class Horse:
    maxHeight = 200; # სანტიმეტრი
    def __init__(self, name, horsehair):
        self.name = name
        self.horsehair = horsehair
    def run(self):
        print ("Horse run")
```

```

def showName(self):
    print ("Name: (House's method): ", self.name)
def showInfo(self):
    print ("Horse Info")
class Donkey:
    def __init__(self, name, weight):
        self.name = name
        self.weight = weight
    def run(self):
        print ("Donkey run")
    def showName(self):
        print ("Name: (Donkey's method): ", self.name)
    def showInfo(self):
        print ("Donkey Info")
# Mule კლასი მემკვიდრეობითობას იღებს Horse და
Donkey კლასებიდან
class Mule(Horse, Donkey):
    def __init__(self, name, hair, weight):
        Horse.__init__(self, name, hair)
        Donkey.__init__(self, name, weight)
    def run(self):
        print ("Mule run")
    def showInfo(self):
        print ("-- Call Mule.showInfo: --")
        Horse.showInfo(self)
        Donkey.showInfo(self)

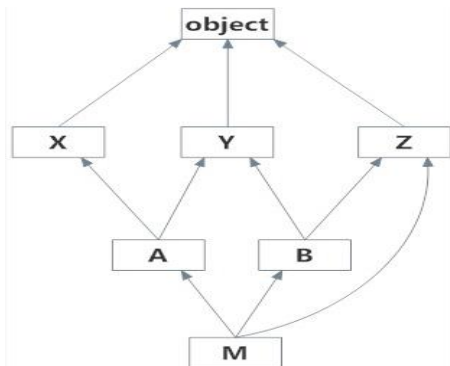
```

```
# ---- Test -----
# 'maxHeight' ცვლადი მემკვიდრეობით მიღებულია
Horse კლასიდან
print ("Max height ", Mule.maxHeight)
mule = Mule("Mule", 20, 1000)
mule.run()
mule.showName()
mule.showInfo()
პროგრამის შედეგი:
```

```
RESTART: C:/Users/Guliko/AppData/Local
dr.py
Max height 200
Mule run
Name: (House's method): Mule
-- Call Mule.showInfo: --
Horse Info
Donkey Info
>>>
```

10.8. mro() მეთოდი

mro() მეთოდი მოცემული კლასის მშობელი კლასების სის დათვალიერების საშუალებას იძლევა. ვნახოთ მაგალითზე:



Python ენაში pass (pass statement) ბრძანება null (ცარიელი) ბრძანების მსგავსია. იგი არაფერს არ აკეთებს და გამოიყენება მაშინ, როდესაც კლასს ან მეთოდს არა აქვს შინაარსი.

```
class X: pass
class Y: pass
class Z: pass
class A(X,Y): pass
class B(Y,Z): pass
class M(B,A,Z): pass
print(M.mro())
```

პროგრამის შედეგი:

```
RESTART: C:/Users/Guliko/AppData/Local/P
[<class '__main__.M'>, <class '__main__.B
n__.X'>, <class '__main__.Y'>, <class '__
>>>
```

10.9. isinstance და issubclass ფუნქციები

isinstance ფუნქცია ამოწმებს არის თუ არა რაიმე გარკვეული კლასის ობიექტი.

issubclass ფუნქცია ამოწმებს არის თუ არა ესათუის კლასი სხვა კლასის მემკვიდრე.

განვიხილოთ მოცემული ფუნქციების სადემონსტრაციო მაგალითი:

```
class A: pass
```

```

class B(A): pass
# True
print ("isinstance('abc', object): ",isinstance('abc', object))
# True
print ("isinstance(123, object): ",isinstance(123, object))
b = B()
a = A()
# True
print ("isinstance(b, A): ", isinstance(b, A) )
print ("isinstance(b, B): ", isinstance(b, B) )
# False
print ("isinstance(a, B): ", isinstance(a, B) )
# B is subclass of A? ==> True
print ("issubclass(B, A): ", issubclass(B, A) )
# A is subclass of B? ==> False
print ("issubclass(A, B): ", issubclass(A, B) )

```

პროგრამის შედეგი:

```

RESTART: C:/Users/Guliko/AppData/1
isinstance('abc', object): True
isinstance(123, object): True
isinstance(b, A): True
isinstance(b, B): True
isinstance(a, B): False
issubclass(B, A): True
issubclass(A, B): False
>>>

```


10.10. პოლიმორფიზმი, აბსტრაქტული მეთოდი

პოლიმორფიზმი საშუალებას იძლევა ერთნაირად მივმართოთ ობიექტებს, რომელთაც აქვთ ერთნაირი ტიპის ინტერფეისი, ობიექტის შიდა რეალიზაციის მიუხედავად. სხვა სიტყვებით რომ ვთქვათ, პოლიმორფიზმი ითვალისწინებს ერთნაირი სახელების მქონე მეთოდების სხვადასხვა რეალიზებას. ეს ძალიან სასარგებლოა მემკვიდრეობითობის დროს, როდესაც მემკვიდრე კლასში შეიძლება ხელახლა განვსაზღვროთ მშობელი კლასის მეთოდები.

მაგალითში შექმნილია ორი კლასი - English და French. ორივე კლასს აქვს greeting() მეთოდი. ორივე ქმნის სხვადასხვა მისალმებას. შექმნილია ორი სხვადასხვა ობიექტი მოცემული კლასებიდან და გამოძახებულია ამ ორი ობიექტის მოქმედება ერთ intro ფუნქციაში:

```
class English:
    def greeting(self):
        print ("Hello")
class French:
    def greeting(self):
        print ("Bonjour")
def intro(language):
    language.greeting()
flora = English()
aalase = French()
```

intro(flora)

intro(aalase)

პროგრამის შედეგი:

```
RESTART: C:/Users/Guliko/AppData/I
Hello
Bonjour
>>>
```

აბსტრაქტული მეთოდის ან აბსტრაქტული კლასის ცნება არის ისეთ ენებში, როგორცაა Java, C#, მაგრამ სრულად არ განსაზღვრება Python-ში. თუმცა მაინც არის მისი განსაზღვრის ხერხი. აბსტრაქტული (abstract) არის კლასი, რომელშიც განსაზღვრულია აბსტრაქტული მეთოდები, რომლებიც წარმოებულმა კლასმა ხელახლა უნდა განსაზღვროს (override) თუ უნდა მისი გამოყენება. აბსტრაქტული მეთოდები ყოველთვის იწვევენ გამონაკლისს NotImplementedError.

განვიხილოთ მაგალითი:

#აბსტრაქტული კლასი (Abstract class).

```
class AbstractDocument :
```

```
    def __init__(self, name):
```

```
        self.name = name
```

#მეთოდის გამოყენება შეუძლებელია, რამდენადაც ყოველთვის ამოაგდებს შეცდომას

```
    def show(self):
```

```
        raise NotImplementedError("Subclass must implement abstract method")
```

```

class PDF(AbstractDocument):
    #ხელახლა განისაზღვროს მშობელი კლასის
მეთოდი
    def show(self):
        print ("Show PDF document:", self.name)
class Word(AbstractDocument):
    def show(self):
        print ("Show Word document:", self.name)
# -----
documents = [ PDF("Python tutorial"),
              Word("Java IO Tutorial"),
              PDF("Python Date & Time Tutorial") ]
for doc in documents:
    doc.show()

```

პროგრამის შედეგი:

```

== RESTART: C:/Users/Guliko/AppData/Local/Progra
Show PDF document: Python tutorial
Show Word document: Java IO Tutorial
Show PDF document: Python Date & Time Tutorial
>>>

```

მოყვანილი მაგალითი დემონსტრირებს უკეთეს პოლიმორფიზმს Python-ში. Document ობიექტი შეიძლება წარმოდგენილი იყოს სხვადასხვა ფორმებში: (PDF, Word, Excel და სხვა)

10.11. ოპერატორების გადატვირთვა

ოპერატორების გადატვირთვა Python-ში არის შესაძლებლობა სპეციალური მეთოდების დახმარებით კლასებში ხელახლა განისაზღვროს ენის სხვადასხვა ოპერატორები. ასეთი მეთოდები შემოსაზღვრულია წინ და ბოლოში ორმაგი ხაზგასმით. ოპერატორად ჩაითვლება არამხოლოდ `+`, `-`, `*`, `/` ნიშნები, არამედ ენის სინტაქსის სპეციფიკა, რომელიც უზრუნველყოფს ოპერაციებს: ობიექტის შექმნა, ობიექტის გამოძახება როგორც ფუნქციის, ობიექტის ელემენტზე მიმართვა ინდექსის მიხედვით, ობიექტის გამოტანა და სხვა.

ჩვენ უკვე გამოვიყენეთ ოპერატორების გადატვირთვის ზოგიერთი მეთოდი. ესენია:

- `__init__()` – კლასის კონსტრუქტორი, რომელიც გამოიძახება ობიექტების შექმნისას;
- `__del__()` – კლასის დესტრუქტორი, რომელიც გამოიძახება ობიექტების წაშლისას;
- `__str__()` – ობიექტის გარდაქმნა სტრიქონულ წარმოდგენად. გამოიძახება, როდესაც ობიექტი გადაეცემა `print()` და `str()` ფუნქციებს;
- `__add__()` – შეკრების ოპერატორის გადატვირთვის მეთოდი. გამოიძახება, როდესაც ობიექტი მონაწილეობს შეკრების ოპერაციაში ოპერანდთან მარცხენა მხრიდან;
- `__setattr__()` – გამოიძახება, როდესაც ობიექტის ატრიბუტზე სრულდება მინიჭება.

Python ენაში ოპერატორების გადატვირთვის მრავალი სხვა მეთოდია. განვიხილოთ მათგან ყველაზე ხშირად გამოყენებადი. სინამდვილეში, ოპერატორების გადატვირთვა სამომხმარებლო კლასებში გამოიყენება არცთუ ისე ხშირად, თუ არ ჩავთვლით კონსტრუქტორს. მაგრამ, რადგან ობიექტზე ორიენტირებულ დაპროგრამებაში არსებობს ასეთი შესაძლებლობები, მიზანშეწონილია განვიხილოთ ზოგიერთი შემთხვევა.

ოპერატორების გადატვირთვის შესაძლებლობა უზრუნველყოფს სამომხმარებლო კლასის მსგავსებას Python-ის ჩაშენებულ კლასებთან. ვიცით, რომ ყველა ჩაშენებული მონაცემთა ტიპი Python-ში არის კლასი. შედეგად ყველა ობიექტს შეიძლება ჰქონდეს ერთნაირი ინტერფეისი. თუ თქვენი კლასი ითვალისწინებს ობიექტის ელემენტზე მიმართვას ინდექსის მიხედვით, მაგალითად `a[0]`, ეს შეიძლება უზრუნველყოფილ იქნას.

დავუშვათ გვაქვს B კლასი-აგრეგატი, რომელიც სიაში შეიცავს A კლასის ობიექტებს:

```
class A:
    def __init__(self, arg):
        self.arg = arg
    def __str__(self):
        return str(self.arg)
class B:
    def __init__(self, *args):
        self.aList = []
```

```
for i in args:
```

```
    self.aList.append(A(i))
```

```
group = B(5, 10, 'abc')
```

სიის ელემენტის მისაღებად შეგვიძლია მივმართოთ aList ველს ინდექსით:

```
print(group.aList[1])
```

პროგრამის შედეგი:

```
RESTART: C:/Users/Guliko/AppData/  
10  
>>>
```

მაგრამ, საინტერესოა, როგორ ამოვიღებთ ელემენტს ინდექსით თვით ობიექტიდან და არა მისი ველიდან:

```
class B:
```

```
    def __init__(self, *args):
```

```
        self.aList = []
```

```
        for i in args:
```

```
            self.aList.append(A(i))
```

```
    def __getitem__(self, i):
```

```
        return self.aList[i]
```

```
group = B(5, 10, 'abc')
```

```
print(group.aList[1]) # გამოიტანს 10
```

```
print(group[0]) # 5
```

```
print(group[2]) # abc
```

პროგრამის შედეგი:

```
RESTART: C:/Users/Guliko/AppData/Lc
10
5
abc
>>>
```

ამას აკეთებს კლასის ობიექტები, რომლებიც Python-ში ჩაშენებული კლასები-მიმდევრობების (სიები, სტრიქონები, კორტეჟები) ობიექტების მსგავსია. მეთოდი `__getitem__()` გადატვირთავს ინდექსის მიხედვით ელემენტის ამოღების ოპერაციას. სხვა სიტყვებით, ეს მეთოდი გამოიძახება, როდესაც ობიექტზე სრულდება ელემენტის ამოღების ოპერაცია: `ობიექტი[ინდექსი]`.

ზოგჯერ საჭიროა, რომ ობიექტის ქცევა ფუნქციის მსგავსი იყოს. რაც ნიშნავს, რომ თუ გვაქვს `a` ობიექტი, მაშინ შეგვიძლია მას მივმართოთ ფუნქციის ნოტაციით, ანუ მას შემდეგ დავსვათ მრგვალი ფრჩხილები და გადავცეთ არგუმენტები საჭიროების შემთხვევაში:

```
a = A()
a()
a(3, 4)
```

`__call__()` მეთოდი ავტომატურად გამოიძახება, როდესაც ობიექტს მიმართავენ, როგორც ფუნქციას. მაგალითად, აქ მეორე სტრიქონში მოხდება მეთოდის გამოიძახება: `__call__()` რაიმე კლასი:

```
ობიექტი = კლასი()
ობიექტი([შესაძლო არგუმენტები])
განვიხილოთ მაგალითი:
class Changeable:
```

```

def __init__(self, color):
    self.color = color
def __call__(self, newcolor):
    self.color = newcolor
def __str__(self):
    return "%s" % self.color
canvas = Changeable("green")
frame = Changeable("blue")
canvas("red")
frame("yellow")
print (canvas, frame)

```

მოცემულ მაგალითში კლასის კონსტრუქტორის დახმარებით ობიექტების შექმნისას დაყენდება მათი ფერი. თუ მოითხოვება მისი შეცვლა, საკმარისია ობიექტზე მიმართვა ისე, როგორც ფუნქციაზე და არგუმენტის სახით გადაეცეს ახალი ფერი. ასეთი მიმართვა ავტომატურად გამოიძახებს `__call__()` მეთოდს, რომელიც, მოცემულ შემთხვევაში, შეცვლის ობიექტის `color` ატრიბუტს.

Python-ში `__str__()` მეთოდის გარდა არის მისი მსგავსი ქცევის მიხედვით, მაგრამ უფრო დაბალდონიანი მეთოდი `__repr__()`. ორივე მეთოდი აბრუნებს სტრიქონს.

თუ კლასში არის მხოლოდ `__str__()` მეთოდი, მაშინ ობიექტზე მიმართვისას ინტერპრეტატორში `print()` ფუნქციის გარეშე იგი არ გამოიძახება:


```

>>> class A:
    def __str__(self):
        return "This is object of A"

>>> a = A()
>>> print(a)
This is object of A

>>> a
<__main__.A instance at 0x7fe964a4cdd0>

>>> str(a)
'This is object of A'

>>> repr(a)
'<__main__.A instance at 0x7fe964a4cdd0>'

```

ობიექტის სტრიქონად გარდაქმნის მცდელობა `str()` ჩაშენებული ფუნქციის დახმარებით, აგრეთვე იძახებს `__str__()` მეთოდს. ანუ `__str__()` მეთოდი გადატვირთავს არა `print()` ფუნქციას, არამედ `str()` ფუნქციას. `print()` ფუნქცია ისეა მოწყობილი, რომ თვითონ იძახებს `str()`-ს თავისი არგუმენტებისთვის.

Python-ში არის ჩაშენებული `repr()` ფუნქცია, რომელიც ისევე, როგორც `str()` ობიექტს გარდაქმნის სტრიქონად, მაგრამ „ნედლ“ სტრიქონად. განვიხილოთ მაგალითი:

```

>>> a = '3 + 2'
>>> b = repr(a)
>>> a
'3 + 2'
>>> b

```

```

"3 + 2"
>>> eval(a)
5
>>> eval(b)
'3 + 2'
>>> c = "Hello\nWorld"
>>> d = repr(c)
>>> c
'Hello\nWorld'
>>> d
"Hello\nWorld"
>>> print(c)
Hello
World
>>> print(d)
'Hello\nWorld'

```

eval ფუნქცია გარდაქმნის გადაცემულ სტრიქონს პროგრამულ კოდში, რომელიც იქვე სრულდება. print() ფუნქცია ასრულებს გადასვლას ახალ სტრიქონზე, თუ მასში არის გამოყენებული \n სიმბოლო. repr() ფუნქცია ასრულებს მოქმედებას, რომელიც მიმართულია ინტერპრეტაციისგან სტრიქონის დაცვაზე, ტოვებს მას „ნედლად“ ანუ საწყის ფორმაში.

```

>>> c = "Hello\nWorld"
>>> c # ანალოგი print(repr(c))
'Hello\nWorld'
>>> print(c) # ანალოგი print(str(c))
Hello

```

World

თუმცა უმეტესი შემთხვევისთვის `repr()` და `str()` შორის განსხვავება არ არის მნიშვნელოვანი. ამიტომ კლასში ადვილია განისაზღვროს ერთი მეთოდი `__repr__()`. იგი გამოძახებული იქნება სტრიქონებად გარდაქმნის ყველა შემთხვევისთვის:

```
>>> class A:
...     def __repr__(self):
...         return "It's obj of A"
>>> a = A()
>>> a
It's obj of A
>>> repr(a)
"It's obj of A"
>>> str(a)
"It's obj of A"
>>> print(a)
It's obj of A
```

თუ საჭიროა მონაცემთა განსხვავებული სახით გამოტანა, მაშინ კლასში უნდა განისაზღვროს ოპერატორების გადატვირთვის ორივე მეთოდი.

განვიხილოთ კლასის კონსტრუქტორის გადატვირთვის მაგალითი. `__init__` მეთოდი გადატვირთავს კლასის კონსტრუქტორს:

```
>>> class A:
...     def __init__(self, name):
...         self.name = name
>>> a = A('Python')
>>> print(a.name) #შედეგი: Python
```

XI თავი

თარიღთან და დროსთან მუშაობა

11.1. datetime მოდული

თარიღსა და დროსთან მუშაობის ძირითადი ფუნქციები განთავსებულია datetime მოდულში შემდეგი კლასების სახით:

date

time

datetime

11.1.1. date კლასი

თარიღებთან სამუშაოდ ვისარგებლოთ date კლასით, რომელიც განსაზღვრულია datetime მოდულში. date ობიექტის შესაქმნელად შეგვიძლია გამოვიყენოთ date კონსტრუქტორი, რომელიც თანმიმდევრულად იღებს სამ პარამეტრს: წელი, თვე, რიცხვი.

```
date(year, month, day)
```

შევქმნათ რაიმე თარიღი:

```
import datetime
```

```
yesterday = datetime.date(2018,10,1)
```

```
print(yesterday) # 2018-10-01
```

მიმდინარე თარიღის მისაღებად ვისარგებლოთ today() მეთოდით :

```
from datetime import date
```

```
today = date.today()
print(today)    # 2018-10-01
print("{}.{}.{}".format(today.day, today.month, today.year))
day, month, year თვისებებით შეიძლება მივიღოთ
ცალკე დღე, თვე და წელი.
```

11.1.2. time კლასი

დროსთან სამუშაოდ გამოიყენება time კლასი. მისი კონსტრუქტორის გამოყენებით, შეიძლება შეიქმნას დროის ობიექტი:

```
time([hour] [, min] [, sec] [, microsec])
```

კონსტრუქტორი თანმიმდევრულად იღებს საათებს, წუთებს, წამებს და მიკროწამებს. ყველა პარამეტრი არააუცილებელია და თუ რომელიმე პარამეტრს არ გადავცემთ, შესაბამისი მნიშვნელობა ჩაითვლება ნულად.

```
from datetime import time
current_time = time()
print(current_time)    # 00:00:00
current_time = time(16, 25)
print(current_time)    # 16:25:00
current_time = time(16, 25, 45)
print(current_time)    # 16:25:45
```

11.1.3. datetime კლასი

datetime კლასი ერთსახელიანი მოდულიდან აერთიანებს თარიღსა და დროსთან მუშაობის

შესაძლებლობებს. `datetime` ობიექტის შესაქმნელად შეიძლება შემდეგი კონსტრუქტორის გამოყენება:

```
datetime(year, month, day [, hour] [, min] [, sec] [,
microsec])
```

პირველი-სამი პარამეტრი, რომლებიც წარმოადგენენ წელს, თვეს და დღეს, აუცილებელია. დანარჩენი არააუცილებელია და თუ არ მივუთითებთ მათ მნიშვნელობას, დუმილით ინიციალიზდება ნული.

```
from datetime import datetime
deadline = datetime(2018, 10, 1)
print(deadline) # 2018-10-01 00:00:00
deadline = datetime(2018,10, 1, 4, 30)
print(deadline) #2018-10-01 04:30:00
```

მიმდინარე თარიღისა და დროის გამოსატანად გამოვიძახოთ `now()` მეთოდი:

```
from datetime import datetime
now = datetime.now()
print(now) # 2018-10-01 10:29:00.083843
print("{}.{:}.{:}{:}".format(now.day,now.month, now.year,
now.hour, now.minute)) # 1.10.2018 10:29
print(now.date())
print(now.time())
```

პროგრამის შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/Lc
2018-10-01 10:29:00.083843
1.10.2018 10:29
2018-10-01
10:29:00.083843
>>>
```

day, month, year, hour, minute, second თვისებებით შეიძლება მივიღოთ თარიღისა და დროის ცალკეული მნიშვნელობები. date() და time() მეთოდებით შეიძლება მივიღოთ ცალკე თარიღი და დრო შესაბამისად.

11.1.4. სტრიქონის თარიღად გარდაქმნა

strptime() მეთოდი სტრიქონის თარიღად გარდაქმნის საშუალებას იძლევა. ეს მეთოდი იღებს ორ პარამეტრს:

```
strptime(str, format)
```

პირველი str პარამეტრი არის თარიღისა და დროის სტრიქონული განსაზღვრა, ხოლო მეორე - ფორმატი, რომელიც განსაზღვრავს თუ როგორ არის განთავსებული თარიღისა და დროის ცალკეული ნაწილები ამ სტრიქონში.

ფორმატის განსაზღვრისთვის შეგვიძლია გამოვიყენოთ შემდეგი კოდი:

%d: დღე რიცხვის სახით

%m: თვის რიგითი ნომერი

%y: წელი ორი თანრიგის სახით

%Y: წელი 4 თანრიგის სახით

%H: საათი 24 საათიანი ფორმატით

%M: წუთი

%S: წამი

სხვადასხვა ფორმატების გამოყენება:

```
from datetime import datetime
```

```
deadline = datetime.strptime("10/10/2018", "%d/%m/%Y")
```

```

print(deadline) # 2018-10-10 00:00:00
deadline = datetime.strptime("10/10/2018 12:30",
"%d/%m/%Y %H:%M")
print(deadline) # 2018-10-10 12:30:00
deadline = datetime.strptime("10-10-2018 12:30", "%m-
%d-%Y %H:%M")
print(deadline) # 2018-10-10 12:30:00

```

11.2. ოპერაციები თარიღებზე

11.2.1. თარიღისა და დროის ფორმატირება

date და time ობიექტების ფორმატირებისთვის ორივე კლასში გათვალისწინებულია strftime(format) მეთოდი. ეს მეთოდი მიიღებს მხოლოდ ერთ პარამეტრს, რომელიც მიუთითებს ფორმატზე, რომელშიც უნდა გარდაიქმნას თარიღი ან დრო.

ფორმატის განსაზღვრისთვის შეგვიძლია გამოვიყენოთ ფორმატირების ჩამოთვლილი კოდებიდან ერთ-ერთი:

- **%a**: კვირის დღის აბრევიატურა. მაგალითად, Wed - სიტყვისგან Wednesday (დუმილით გამოიყენება ინგლისური დასახელება);
- **%A**: კვირის დღე სრულად, მაგალითად, Wednesday;
- **%b**: თვის დასახელების აბრევიატურა. მაგალითად, Oct (October);
- **%B**: თვის დასახელება სრულად. მაგალითად, October;
- **%d**: თვის დღე შევსებული ნულით. მაგალითად, 01;

- **%m**: თვის ნომერი, მუცხებული ნულით. მაგალითად, 05;
- **%y**: წელი ორი თანრიგით;
- **%Y**: წელი 4 თანრიგით;
- **%H**: საათი 24 საათიანი ფორმატით. მაგალითად, 13;
- **%I**: საათი 12 საათიანი ფორმატით. მაგალითად, 01;
- **%M**: წუთი;
- **%S**: წამი;
- **%f**: მიკროწამი;
- **%p**: მაჩვენებელი AM/PM;
- **%c**: თარიღი და დრო ფორმატირებული მიმდინარე Locale-თი;
- **%x**: თარიღი, ფორმატირებული მიმდინარე Locale-თი;
- **%X**: დრო, ფორმატირებული მიმდინარე Locale-თი;

სხვადასხვა ფორმატის გამოყენების მაგალითები:

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d"))
print(now.strftime("%d/%m/%Y"))
print(now.strftime("%d/%m/%y"))
print(now.strftime("%d %B %Y (%A)"))
print(now.strftime("%d/%m/%y %I:%M"))
```

პროგრამის შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/Loca
2018-10-01
01/10/2018
01/10/18
01 October 2018 (Monday)
01/10/18 11:05
>>>
```

დუმილით თვის და დღის დასახელებას გამოიტანს ინგლისურად. თუ გვინდა მიმდინარე Locale-ის გამოყენება, წინასწარ უნდა დავაყენოთ locale მოდულის გამოყენებით:

```
from datetime import datetime
import locale
locale.setlocale(locale.LC_ALL, "")
now = datetime.now()
print(now.strftime("%d %B %Y (%A)"))
```

11.2.2. თარიღსა და დროზე არითმეტიკული ოპერაციები

თარიღებთან მუშაობის დროს ხშირად საჭიროა რაიმე თარიღს დაემატოს ან გამოაკლდეს დროის რაიმე შუალედი. ამისათვის datetime მოდულში განსაზღვრულია timedelta კლასი. ფაქტობრივად, ეს კლასი განსაზღვრავს დროის რაღაც პერიოდს.

დროის შუალედის განსაზღვრისთვის შეიძლება timedelta კონსტრუქტორის გამოყენება:

```
timedelta([days] [, seconds] [, microseconds] [,
milliseconds] [, minutes] [, hours] [, weeks])
```

კონსტრუქტორში თანმიმდევრულად გადავცემთ დღეებს, წამებს, მიკროწამებს, მილიწამებს, წუთებს, საათებს და კვირებს.

განვსაზღვროთ რამდენიმე პერიოდი:

```
from datetime import timedelta
```

```

three_hours = timedelta(hours=3)
print(three_hours)    # 3:00:00
three_hours_thirty_minutes=timedelta(hours=3,
minutes=30) # 3:30:00
two_days = timedelta(2) # 2 days, 0:00:00
two_days_three_hours_thirty_minutes=
timedelta(days=2, hours=3, minutes=30) # 2 days, 3:30:00

```

timedelta გამოყენებით შეგვიძლია შევკრიბოთ ან გამოვაკლოთ თარიღები. მაგალითად, მივიღოთ თარიღი, რომელიც იქნება ორი დღის შემდეგ:

```

from datetime import timedelta, datetime
now = datetime.now()
print(now)
two_days = timedelta(2)
in_two_days = now + two_days
print(in_two_days)

```

მიღებული შედეგი:

```

== RESTART: C:/Users/Guliko/AppData/
2018-10-01 11:21:42.700725
2018-10-03 11:21:42.700725
>>>

```

გავიგოთ რა დრო იყო 10 საათისა და 15 წუთის უკან, ფაქტობრივად მიმდინარე დროს უნდა გამოვაკლოთ 10 საათი და 15 წუთი:

```

from datetime import timedelta, datetime
now = datetime.now()

```

```
till_ten_hours_fifteen_minutes=now-
timedelta(hours=10, minutes=15)
print(till_ten_hours_fifteen_minutes)
```

მიღებული შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/
2018-10-01 01:09:46.244223
>>>
```

11.2.3. timedelta თვისება

timedelta კლასს აქვს რამდენიმე თვისება, რომელთა დახმარებით შეგვიძლია მივიღოთ დროითი შუალედი:

- days - აბრუნებს დღეების რაოდენობას;
- seconds - აბრუნებს წუთების რაოდენობას;
- microseconds-აბრუნებს მიკროწამების რაოდენობას.

total_seconds() მეთოდი აბრუნებს წამების საერთო რაოდენობას, რომელშიც შედის დღეებიც, თვით წამებიც და მიკროწამები.

მაგალითად, გავიგოთ რა დროითი პერიოდია ორ თარიღს შორის:

```
from datetime import timedelta, datetime
now = datetime.now()
twenty_two_may = datetime(2018, 10, 4)
period = twenty_two_may - now
print("{} დღე {} წამი {}
მიკროწამი".format(period.days,period.seconds,
period.microseconds))
```

```
# 1 დღე 47995 წამი 792386 მიკროწამი
print("სულ: {} წამი".format(period.total_seconds()))
# სულ: 134328.322065 წამი
```

11.2.4. თარიღების შედარება

რიცხვებისა და სტრიქონების მსგავსად თარიღები შეიძლება შევადაროთ შედარების სტანდარტული ოპერატორების დახმარებით:

```
from datetime import datetime
now = datetime.now()
deadline = datetime(2018, 10, 1)
if now > deadline:
    print("პროექტის ჩაბარების ვადა გავიდა")
elif now.day == deadline.day and now.month ==
deadline.month and now.year == deadline.year:
    print("პროექტის ჩაბარების ვადაა დღეს")
else:
    period = deadline - now
    print("დარჩა {} დღე".format(period.days))
print("ეხლა არის:", now)
```

პროგრამის შედეგი:

```
== RESTART: C:/Users/Guliko/AppData/Local/
პროექტის ჩაბარების ვადა გავიდა
ეხლა არის: 2018-10-02 10:57:40.143283
>>>
```

XII თავი

გრაფიკული ინტერფეისის შექმნა

12.1. დანართის ფანჯრის შექმნა

დღევანდელ პერიოდში მრავალი პროგრამა იყენებს გრაფიკულ ინტერფეისს, რომელიც გაცილებით მოხერხებულია მომხმარებლისთვის, ვიდრე კონსოლი. Python დაპროგრამების ენის დახმარებით, ასევე შეიძლება შეიქმნას გრაფიკული პროგრამები. ამისათვის Python -ში დუმილით გამოიყენება სპეციალური კომპონენტების ნაკრები - tkinter, რომელიც მისაწვდომია ცალკეული ჩაშენებული მოდულის სახით, რომელიც შეიცავს ყველა საჭირო გრაფიკულ კომპონენტს - ღილაკებს, ტექსტურ ველებს და ა.შ.

გრაფიკული პროგრამების შექმნის საბაზო მომენტი არის ფანჯრის შექმნა. შემდეგ ფანჯარაში დაემატება გრაფიკული ინტერფეისის ყველა დანარჩენი კომპონენტი. დასაწყისში შევქმნათ მარტივი ფანჯარა:

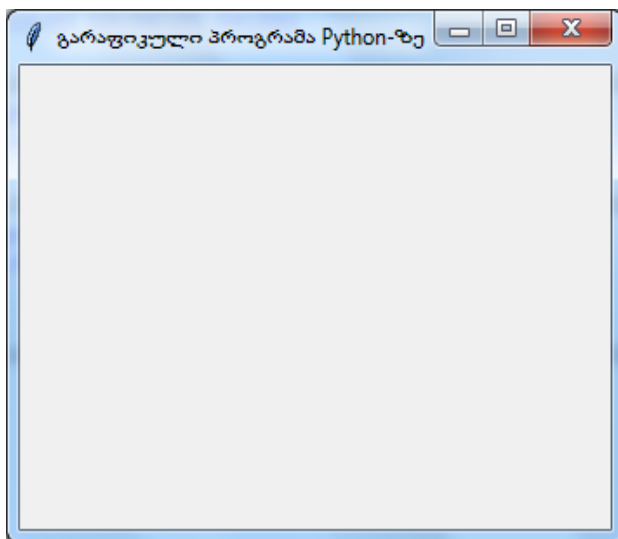
```
from tkinter import *  
root = Tk()  
root.title("გრაფიკული პროგრამა Python-ზე")  
root.geometry("400x300")  
root.mainloop()
```

გრაფიკული ფანჯრის შესაქმნელად გამოიყენება **Tk()** კონსტრუქტორი, რომელიც განსაზღვრულია **tkinter** მოდულში. შესაქმნელი ფანჯარა root ცვლადს მიენიჭება

და ამ ცვლადით შეგვიძლია ვმართოთ ფანჯრის ატრიბუტები. კერძოდ, `title()` მეთოდის დახმარებით შეიძლება ფანჯრის სათაურის დაყენება, `geometry()` მეთოდის დახმარებით - ფანჯრის ზომები. ზომების დასაყენებლად `geometry()` მეთოდში გადაეცემა სტრიქონი ფორმატში "სიგანე x სიმაღლე". თუ დანართის ფანჯრის შექმნისას `geometry()` მეთოდი არ გამოიძახება, მაშინ ფანჯარა იკავებს იმ სივრცეს, რომელიც საჭიროა შიდა შემადგენლობის განსათავსებლად.

ფანჯრის ასახვისთვის საჭიროა `mainloop()` მეთოდის გამოძახება, რომელიც გაუშვებს ფანჯრის ხდომილების დამუშავების ციკლს მომხმარებელთან ურთიერთობისთვის.

კოდის გაშვების შედეგად მივიღებთ ცარიელ ფანჯარას:

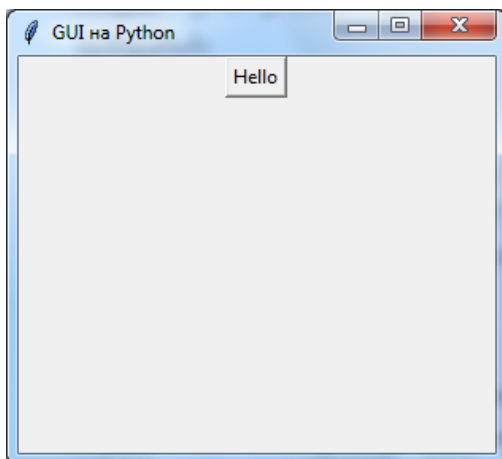


12.2. ღილაკები

tkinter შეიცავს კომპონენტების ნაკრებს, რომელთაგან ერთ-ერთია ღილაკი. დავამატოთ ღილაკი ფანჯარას:

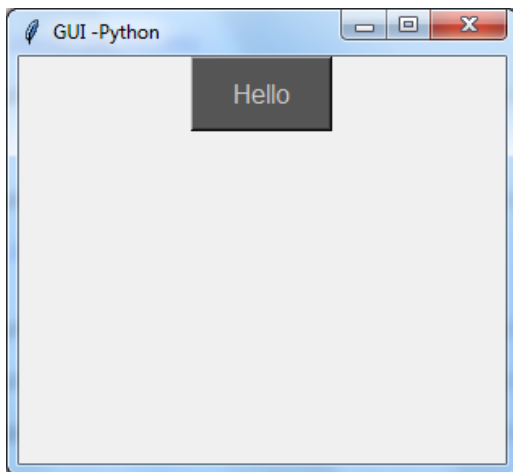
```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
btn = Button(text="Hello")
btn.pack()
root.mainloop()
```

ღილაკის შესაქმნელად გამოიყენება Button() კონსტრუქტორი. ამ კონსტრუქტორში text პარამეტრის დახმარებით შეიძლება ღილაკის ტექსტის შექმნა. ელემენტების ხილვადობისთვის გამოიძახება pack() მეთოდი. მიიღება შედეგი:



ცალკეულ კომპონენტს, მათ შორის ღილაკს, აქვს მთელი რიგი ატრიბუტები, რომლებიც გავლენას ახდენენ მის ვიზუალიზებაზე და რომლებიც შეიძლება ავაწყოთ კონსტრუქტორით:

```
from tkinter import *
root = Tk()
root.title("GUI -Python")
root.geometry("300x250")
btn = Button(text="Hello", # ღილაკის ტექსტი
             background="#555",# ღილაკის ფონის ფერი
             foreground="#ccc",# ტექსტის ფერი
             padx="20", #შეწევა საზღვრიდან ჰორიზონტალურად
             pady="8", # შეწევა საზღვრიდან ვერტიკალურად
             font="16" ) # შრიფტის სიმაღლე
btn.pack()
root.mainloop()
```



padx, pady, font პარამეტრები იღებენ რიცხვით მნიშვნელობას, ხოლო background და foreground პარამეტრები იღებენ ფერის თექვსმეტობით მნიშვნელობას. font პარამეტრი შეიცავს შრიფტის განსაზღვრას.

Button კონსტრუქტორს შეუძლია მიიღოს შემდეგი პარამეტრები:

Button (master, options)

master პარამეტრი წარმოადგენს ბმულს მშობელ კონტეინერზე. განხილულ შემთხვევაში ეს შეიძლება ყოფილიყო თვით გრაფიკული ფანჯარა და ჩვენ შეგვეძლო დაგვეწერა:

```
from tkinter import *  
root = Tk()  
root.title("GUI - Python")  
root.geometry("300x250")  
btn = Button(root, text="Hello")  
btn.pack()
```

თუ კოდში იქმნება ერთი ფანჯარა, მაშინ ღილაკი და სხვა ნებისმიერი ელემენტი დუმილით განთავსდება ამ ფანჯარაში. ამ შემთხვევაში პირველი პარამეტრი შეიძლება გამოვტოვოთ, როგორც ზედა მაგალითებში. თუ კოდში იქმნება რამდენიმე ფანჯარა, მაშინ შეგვიძლია Button კონსტრუქტორს გადავცეთ ბმული საჭირო ფანჯარაზე. მეორე პარამეტრი - options წარმოადგენს პარამეტრების ნაკრებს, რომლებიც შეგვიძლია დავაყენოთ მათი სახელების მიხედვით:

- **activebackground:** ღილაკის ფერი, როდესაც იგი იმყოფება დაჭერილ მდგომარეობაში;
- **activeforeground:** ღილაკის ტექსტის ფერი დაჭერილ მდგომარეობაში;
- **bd:** საზღვრის სისქე (დუმილით 2)
- **bg/background:** ღილაკის ფონის ფერი;
- **fg/foreground:** ღილაკის ტექსტის ფერი;
- **font:** ტექსტის შრიფტი, მაგალითად, font="Arial 14" - Arial შრიფტი, 14px სიმაღლით, ან font=("Verdana", 13, "bold") - Verdana შრიფტი 13px სიმაღლით მსხვილი სისქით;
- **height:** ღილაკის სიმაღლე;
- **highlightcolor:** ღილაკის ფერი, როდესაც იგი ფოკუსშია;
- **image:** გამოსახულება ღილაკზე;
- **justify:** ტექსტის გასწორება. LEFT გაასწორებს ტექსტს მარცხენა კიდესთან, CENTER - ცენტრზე, RIGHT - მარჯვენა კიდესთან;
- **padx:** ღილაკის შეწევა ტექსტამდე მარცხნიდან და მარჯვნიდან;
- **pady:** შეწევა საზღვრიდან ტექსტამდე ზევიდან და ქვევიდან;
- **relief:** განსაზღვრავს საზღვრის ტიპს, შეიძლება მიიღოს მნიშვნელობები: SUNKEN, RAISED, GROOVE, RIDGE;
- **state:** აყენებს ღილაკის მდგომარეობას, შეიძლება მიიღოს მნიშვნელობები: DISABLED, ACTIVE, NORMAL (დუმილით);
- **text:** დააყენებს ღილაკის ტექსტს;

- **textvariable**: ახორციელებს მიზმას StringVar ელემენტზე;
- **underline**: მიუთითებს ღილაკის ტექსტზე იმ სიმბოლოს ნომერზე, რომელიც გაიხაზება. დუმილით, მნიშვნელობა -1, ანუ არცერთი სიმბოლო არ გაიხაზება;
- **width**: ღილაკის სიგანე;
- **wraplength**: დადებითი მნიშვნელობის დროს ღილაკზე ტექსტი გადაიტანება სხვა სტრიქონზე.

12.2.1. ღილაკის დაჭერის დამუშავება

ღილაკზე დაჭერის დასამუშავებლად საჭიროა კონსტრუქტორში დაყენდეს command პარამეტრი, ფუნქციაზე ბმულის მინიჭებით, რომელიც ამუშავდება დაჭერისას:

```

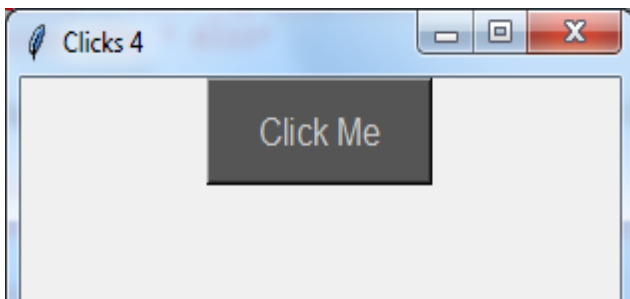
from tkinter import *
clicks = 0
def click_button():
    global clicks
    clicks += 1
    root.title("Clicks {}".format(clicks))
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
btn=Button(text="Click Me", background="#555",
foreground="#ccc",
           padx="20",pady="8",font="16",
command=click_button)

```

```
btn.pack()
```

```
root.mainloop()
```

კოდში დაჭერის დამმუშავებელი არის `click_button` ფუნქცია. ამ ფუნქციაში იცვლება გლობალური ცვლადი - `clicks`, რომელიც ინახავს დაწკაპებათა როცხვს და მისი მნიშვნელობა გამოვა ფანჯრის სათაურში. ასეთი სახით, ღილაკზე სათითაოდ დაჭერისას ამუშავდება `click_button` ფუნქცია და დაწკაპებათა რაოდენობა გაიზრდება:



12.3. ელემენტების თვისებების შეცვლა

ღილაკის ატრიბუტის, მაგალითად, ტექსტის შესაცვლელად შეიძლება `StringVar` შუალედური შუალედური კომპონენტის გამოყენება. მას აქვს ორი მეთოდი:

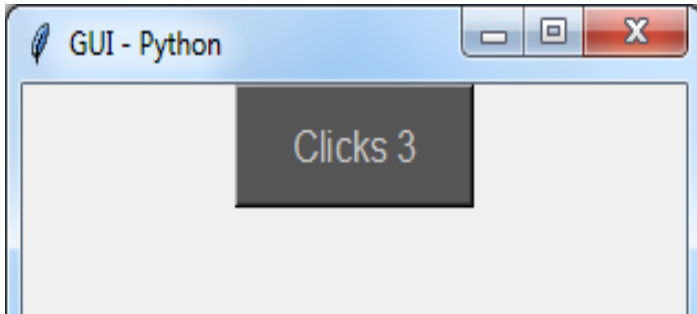
- `get()` - აბრუნებს სტრიქონს `StringVar` - დან;
- `set(str)` - გადაჰყავს სტრიქონი `StringVar`-ში.

`StringVar` ობიექტის ვიზუალური ელემენტის ტექსტთან დასაკავშირებლად ამ ელემენტის კონსტრუქ-

ტორში უნდა დაყენდეს textvariable პარამეტრი. დავუშვათ, ღილაკზე დაჭერით გვინდა ტექსტის შეცვლა:

```
from tkinter import *
clicks = 0
def click_button():
    global clicks
    clicks += 1
    buttonText.set("Clicks {}".format(clicks))
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
buttonText = StringVar()
buttonText.set("Clicks {}".format(clicks))
btn=Button(textvariable=buttonText, background="#555",
foreground="#ccc",padx="20",pady="8",font="16",
command=click_button)
btn.pack()
root.mainloop()
```

იქმნება buttonText ცვლადი, რომელსაც აქვს StringVar ტიპი. მისთვის დაყენდება საწყისი მნიშვნელობა, შემდეგ btn ცვლადი უკავშირდება ამ ტექსტს textvariable პარამეტრით. საბოლოო ჯამში ღილაკზე დაჭერისას შეიცვლება მისი ტექსტი:



StringVar კომპონენტის გარდა მისაწვდომია მთელი რიგი მსგავსი კომპონენტები მონაცემთა სხვა ტიპებისთვის:

- **IntVar**
- **BooleanVar**
- **DoubleVar**

მაგალითად, შეგვეძლოს სტრიქონი დაგვეკავშირებინა IntVar ცვლადთან და გამოგვეტანა დაწკაპებათა რაოდენობა:

```
from tkinter import *
def click_button():
    clicks.set(clicks.get() + 1)
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
clicks = IntVar()
clicks.set(0)
btn = Button(textvariable=clicks, background="#555",
             foreground="#ccc",
             padx="20",pady="8",font="Verdana13",
```

```
command=click_button)
```

```
btn.pack()
```

```
root.mainloop()
```



არის სხვა ხერხი, რომლის გამოყენება მოხერხებულია მაშინ, როდესაც საჭიროა კომპონენტის არა მარტო ტექსტის, არამედ სხვა პარამეტრის ცვლილება. ეს ხერხი მდგომარეობს ელემენტისთვის config() მეთოდის გამოძახებაში. მაგალითად, გამოვიყენოთ config მეთოდი:

```
from tkinter import *
clicks = 0
def click_button():
    global clicks
    clicks += 1
    btn.config(text="Clicks {}".format(clicks))
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
btn=Button(text="Clicks 0", background="#555",
           foreground="#ccc",padx="20",pady="8",font="16",
           command=click_button)
btn.pack()
root.mainloop()
```


12.4. ელემენტის პოზიციონირება

ვანჯარაში ელემენტების პოზიციონირებისთვის გამოიყენება სხვადასხვა ხერხი, რომელთაგან ყველაზე მარტივია ელემენტისთვის pack() მეთოდის გამოძახება. ეს მეთოდი მიიღებს შემდეგ პარამეტრებს:

- expand - თუ True-ს ტოლია ელემენტი იკავებს კონტეინერის მთელ სივრცეს;
- fill - განსაზღვრავს, ელემენტი გაიშლება თუ არა, რათა შეავსოს თავისუფალი სივრცე ირგვლივ. ამ პარამეტრმა შეიძლება მიიღოს შემდეგი მნიშვნელობები: NONE (დუმილით, ელემენტი არ გაიშლება), X (გაიშლება მხოლოდ ჰორიზონტალზე), Y (გაიშლება მხოლოდ ვერტიკალზე) და BOTH (ჰორიზონტალზე და ვერტიკალზე).
- side - ელემენტს გაათანაბრებს კონტეინერის ერთი მხრიდან. შეიძლება მიიღოს მნიშვნელობები: TOP, BOTTOM, LEFT, RIGHT.

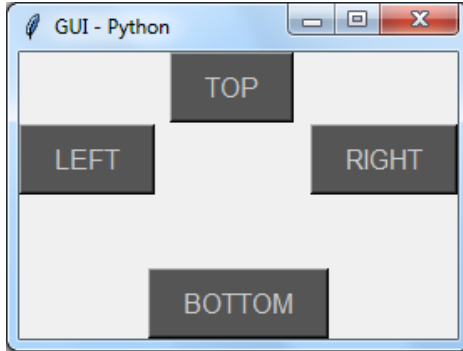
მაგალითად, გავშალოთ ღილაკი მთელ ფორმაზე, expand და fill პარამეტრების გამოყენებით:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
btn1 = Button(text="CLICK ME", background="#555",
              foreground="#ccc",
              padx="15", pady="6", font="15")
```

```
btn1.pack(expand = True, fill=BOTH)
root.mainloop()
```

გამოვიყენოთ side პარამეტრი:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
btn1 = Button(text="BOTTOM", background="#555",
              foreground="#ccc",
              padx="15", pady="6", font="15")
btn1.pack(side=BOTTOM)
btn2 = Button(text="RIGHT", background="#555",
              foreground="#ccc",
              padx="15", pady="6", font="15")
btn2.pack(side=RIGHT)
btn3 = Button(text="LEFT", background="#555",
              foreground="#ccc",
              padx="15", pady="6", font="15")
btn3.pack(side=LEFT)
btn4 = Button(text="TOP", background="#555",
              foreground="#ccc",
              padx="15", pady="6", font="15")
btn4.pack(side=TOP)
root.mainloop()
```



side და fill პარამეტრების კომბინირებით ელემენტი შეიძლება გაიშალოს ვერტიკალურად:

```
btn1 = Button(text="CLICK ME", background="#555",  
foreground="#ccc",  
padx="15", pady="6", font="15")  
btn1.pack(side=LEFT, fill=Y)
```

place მეთოდი

place() მეთოდი პოზიციონირების პარამეტრების უფრო ზუსტად აწყობის საშუალებას იძლევა. იგი იღებს შემდეგ პარამეტრებს:

height და width - ელემენტის სიმაღლე და სიგანე პიქსელებში;

relheight და relwidth - ასევე სიმაღლე და სიგანე, მაგრამ მნიშვნელობის სახით გამოიყენება float რიცხვი 0.0 და 1.0 შუალედში, რომელიც მიუთითებს სიმაღლიდან და სიგანიდან მშობელი კონტეინერის წილზე;

- x და y - ელემენტის განთავსება ჰორიზონტზე და ვერტიკალზე პიქსელებში კონტეინერის მარცხენა ზედა კუთხესთან მიმართებით;
- relx და rely - ასევე, გვაძლევს ელემენტის განთავსებას ჰორიზონტზე და ვერტიკალზე, მაგრამ მნიშვნელობის სახით გამოიყენება float რიცხვი 0.0 და 1.0 შუალედში, რომელიც მიუთითებს სიმალიდან და სიგანიდან მშობელი კონტეინერის წილზე;
- bordermode- გვაძლევს ელემენტის საზღვრის ფორმატს. შეიძლება მიიღოს INSIDE (დუმილით) და OUTSIDE მნიშვნელობა;
- anchor - დააყენებს ელემენტის გაშლის ოპციებს. შეიძლება მიიღოს მნიშვნელობები: n, e, s, w, ne, nw, se, sw, c, რომლებიც არის შემდეგი შემოკლებები: North (ჩრდილოეთი - ზედა), South (სამხრეთი - ქვედა), East (აღმოსავლეთი - მარჯვენა), West (დასავლეთი - მარცხენა) და Center (ცენტრზე).

მაგალითად, განვათავსოთ ღილაკი სიგანით 130 პიქსელი და სიმალით 30 პიქსელი ფანჯრის ცენტრზე:

```
from tkinter import *
clicks = 0
def click_button():
    global clicks
    clicks += 1
    btn.config(text="Clicks {}".format(clicks))
root = Tk()
```

```

root.title("GUI - Python")
root.geometry("300x250")
btn = Button(text="Clicks 0", background="#555",
             foreground="#ccc",
             padx="20", pady="8", font="16",
             command=click_button)
btn.place(relx=.5, rely=.5, anchor="c", height=30,
          width=130, bordermode=OUTSIDE)
root.mainloop()

```

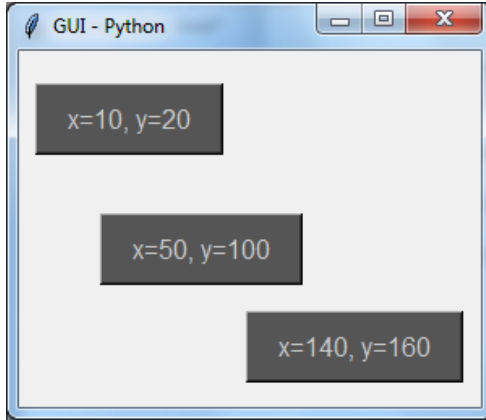
უნდა აღინიშნოს, რომ `place()` მეთოდის გამოყენებისას არ უნდა გამოიყენოთ `pack()` მეთოდი, რათა ელემენტი გახდეს ხილვადი.

განვათავსოთ სამი ღილაკი:

```

from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
btn1 = Button(text="x=10, y=20", background="#555",
             foreground="#ccc", padx="14", pady="7", font="13")
btn1.place(x=10, y=20)
btn2 = Button(text="x=50, y=100", background="#555",
             foreground="#ccc", padx="14", pady="7", font="13")
btn2.place(x=50, y=100)
btn3 = Button(text="x=140, y=160", background="#555",
             foreground="#ccc", padx="14", pady="7", font="13")
btn3.place(x=140, y=160)
root.mainloop()

```



grid მეთოდი

grid მეთოდი ელემენტის პოზიციონირებისთვის იყენებს სხვა მიდგომას. იგი ელემენტის განსაზღვრულ უჯრაში განთავსების საშუალებას იძლევა. grid მეთოდი მიიღებს შემდეგ პარამეტრებს:

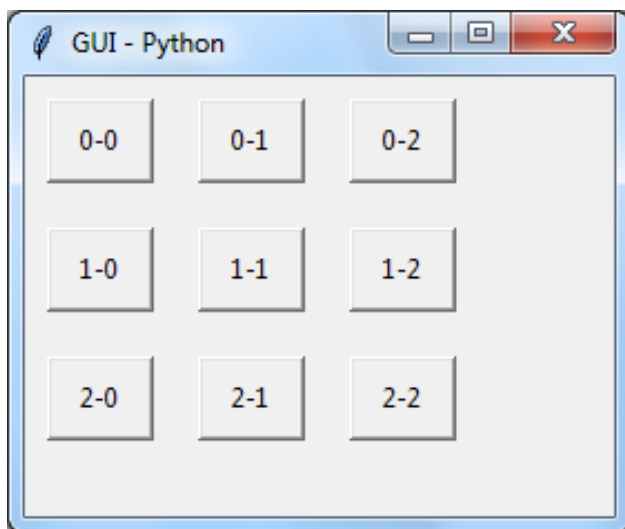
- column - სვეტის ნომერი, ათვლა იწყება ნულიდან;
- row - სტრიქონის ნომერი, ათვლა იწყება ნულიდან;
- colspan-რამდენი სვეტი უნდა დაიკავოს ელემენტმა;
- rowspan-რამდენი სტრიქონი უნდა დაიკავოს ელემენტმა;
- padx და pady - შეწევები ჰორიზონტალურად და ვერტიკალურად შესაბამისად ელემენტის საზღვრიდან მის ტექსტამდე;
- padx და pady - შეწევები ჰორიზონტალურად და ვერტიკალურად შესაბამისად ბადის უჯრის საზღვრიდან ელემენტის საზღვრამდე;

sticky - ელემენტის გათანაბრება უჯრაში, თუ უჯრა ელემენტზე დიდია. შეიძლება მიიღოს მნიშვნელობები: n,

e, s, w, ne, nw, se, sw, რომლებიც უჩვენებენ გათანაბრებების შესაბამის მიმართულებას.

განვსაზღვროთ ბადე 9 დილაკისგან:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
for r in range(3):
    for c in range(3):
        btn = Button(text="{}-{}".format(r,c))
        btn.grid(row=r, column=c, ipadx=10, ipady=6,
                padx=10, pady=10)
root.mainloop()
```



12.5. Label - ტექსტური ჭდე

Python-ში ტექსტური ჭდეები წარმოდგენილია Label ელემენტით. ეს ელემენტი სტატიკური ტექსტის გამოტანის საშუალებას იძლევა, რედაქტირების შესაძლებლობის გარეშე.

Label ელემენტის შესაქმნელად გამოიყენება კონსტრუქტორი, რომელიც იღებს ორ პარამეტრს:

Label(master, options)

master პარამეტრი წარმოადგენს ბმულს მშობელ კონტეინერზე, ხოლო options პარამეტრი წარმოადგენს შემდეგ სახელდებულ პარამეტრებს:

- **anchor** - ტექსტის პოზიციონირება;
- **bg/background** - ფონური ფერი;
- **bitmap** - ბმული გამოსახულებაზე, რომელიც აისახება ჭდეზე;
- **bd** - ჭდის საზღვრის სისქე;
- **fg/foreground** - ტექსტის ფერი;
- **font** - ტექსტის შრიფტი, მაგალითად, font="Arial 14" - Arial შრიფტი, 14px სიმაღლით;
- **height** - ელემენტის სიმაღლე;
- **cursor** - თავის მაჩვენებლის კურსორი ჭდეზე მიახლოვებისას;
- **image** - ბმული გამოსახულებაზე, რომელიც აისახება ჭდეზე;
- **justify** - ტექსტის გათანაბრება; LEFT, CENTER, RIGHT;

- **padx** - შეწვევა ელემენტის საზღვრიდან მის ტექსტამდე მარჯვნიდან და მარცხნიდან;
- **pady** - შეწვევა ელემენტის საზღვრიდან მის ტექსტამდე ზევიდან და ქვევიდან;
- **relief** - განსაზღვრავს საზღვრის ტიპს, დუმილით მნიშვნელობაა FLAT;
- **text** - დააყენებს ჭდის ტექსტს;
- **textvariable** - აკავშირებს StringVar ელემენტთან;
- **underline** - მიუთითებს ღილაკის ტექსტის იმ სიმბოლოს ნომერზე, რომელიც გაიხაზება. დუმილით მნიშვნელობაა -1, ანუ არცერთი სიმბოლო არ გაიხაზება;
- **width** - ელემენტის სიგანე;
- **wrplength** - დადებითი მნიშვნელობის დროს ტექსტის სტრიქონები გადაიტანება.

დანართის ფანჯარაში გამოვიტანოთ მარტივი ტექსტი:

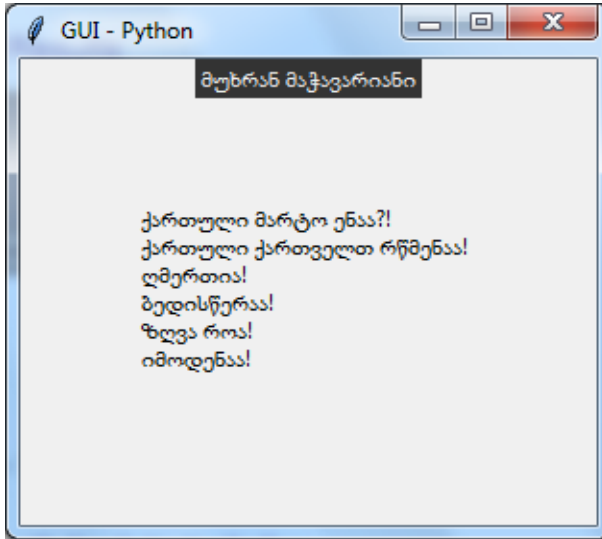
```

from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
label1 = Label(text="მუხრან მაჭავარიანი", fg="#eee",
               bg="#333")
label1.pack()
poetry = "ქართული მარტო ენაა?! \n ქართული
ქართველთ რწმენაა! \n ღმერთია! \n ბედისწერაა! \n ზღვა
როა! \n იმოდენაა!"

```

```
label2 = Label(text=poetry, justify=LEFT)
label2.place(relx=.2, rely=.3)
root.mainloop()
```

ტექსტის სხვა სტრიქონზე გადასატანად შეიძლება \n მიმდევრობის გამოყენება:



12.6. Entry - შეტანის ველი

Entry ელემენტი წარმოადგენს ტექსტის შესატან ველს. Entry კონსტრუქტორი იღებს შემდეგ პარამეტრებს:

```
Entry(master, options)
```

სადაც master არის ბმული მშობელ ფანჯარაზე, ხოლო options - შემდეგი პარამეტრების ნაკრები:

- **bg** - ფონის ფერი;
- **bd** - საზღვრის სისქე;

- **cursor** - თავის მაჩვენებლის კურსორი ტექსტურ ველთან მიტანისას;
- **fg** - ტექსტის ფერი;
- **font** - ტექსტის შრიფტი;
- **justify** - ათანაბრებს ტექსტს. მნიშვნელობებია: LEFT, CENTER, RIGHT;
- **relief** - განსაზღვრავს საზღვრის ტიპს, დუმილით მნიშვნელობაა FLAT;
- **selectbackground** - გამოყოფილი ტექსტის ფრაგმენტის ფონის ფერი;
- **selectforeground** - გამოყოფილი ტექსტის ფერი;
- **show** - იძლევა ნიღაბს შესატანი სიმბოლოებისთვის;
- **state** - ელემენტის მდგომარეობა, შეიძლება მიიღოს მნიშვნელობები: NORMAL (დუმილით) და DISABLED;
- **textvariable** - დაკავშირება StringVar ელემენტთან;
- **width** - ელემენტის სიგანე.

განვსაზღვროთ Entry ელემენტი და ღილაკზე დაჭერისას გამოვიდეს მისი ტექსტი ცალკე ფანჯარაში შეტყობინებით:

```

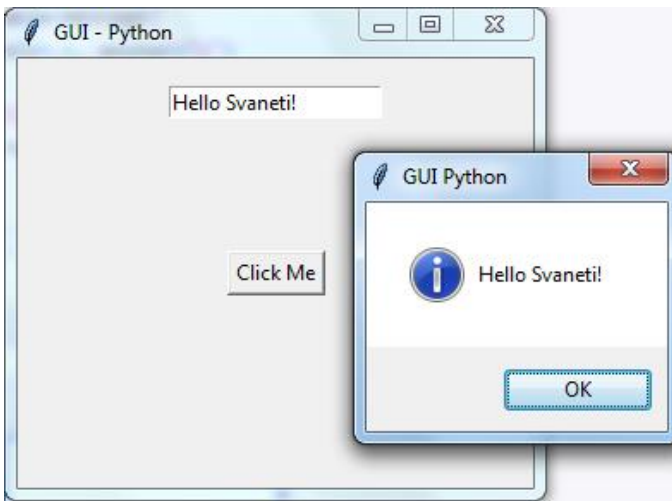
from tkinter import *
from tkinter import messagebox
def show_message():
    messagebox.showinfo("GUI Python", message.get())
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")

```

```

message = StringVar()
message_entry = Entry(textvariable=message)
message_entry.place(relx=.5, rely=.1, anchor="c")
message_button = Button(text="Click Me",
                        command=show_message)
message_button.place(relx=.5, rely=.5, anchor="c")
root.mainloop()

```



შეტყობინების გამოსატანად აქ გამოიყენება დამატებითი messagebox მოდული, რომელიც შეიცავს showinfo() ფუნქციას, რომელიც გამოიტანს ტექსტურ ველში შეტანილ ტექსტს. შეტანილი ტექსტის მისაღებად გამოიყენება StringVar კომპონენტი, როგორც იყო აღწერილი ერთ-ერთ წინა თემაში.

შევქმნათ უფრო რთული მაგალითი შეტანის ფორმით:

```

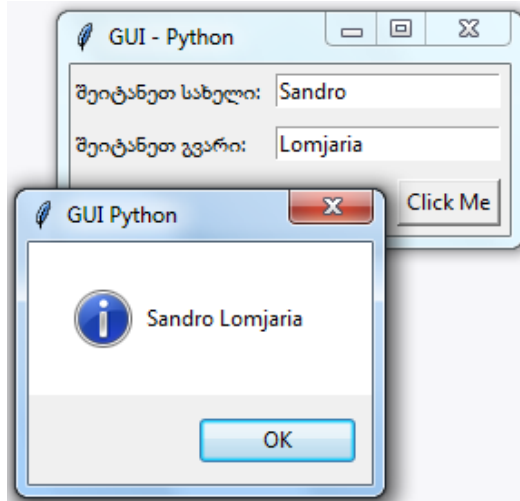
from tkinter import *

```

```

from tkinter import messagebox
def display_full_name():
    messagebox.showinfo("GUI Python", name.get() + " " +
        surname.get())
root = Tk()
root.title("GUI - Python")
name = StringVar()
surname = StringVar()
name_label = Label(text="შეიტანეთ სახელი:")
surname_label = Label(text="შეიტანეთ გვარი:")
name_label.grid(row=0, column=0, sticky="w")
surname_label.grid(row=1, column=0, sticky="w")
name_entry = Entry(textvariable=name)
surname_entry = Entry(textvariable=surname)
name_entry.grid(row=0, column=1, padx=5, pady=5)
surname_entry.grid(row=1, column=1, padx=5, pady=5)
message_button = Button(text="Click Me",
    command=display_full_name)
message_button.grid(row=2, column=1, padx=5, pady=5,
    sticky="e")
root.mainloop()

```



Entry მეთოდები

Entry ელემენტს აქვს მთელი რიგი მეთოდები. მათ შორის ძირითადებია:

- `insert(index, str)` - ტექსტურ ველში ჩასვამს სტრიქონს ინდექსის მიხედვით;
- `get()` - აბრუნებს ტექსტურ ველში შეტანილ ტექსტს;
- `delete(first, last=None)` - წაშლის სიმბოლოს `first` ინდექსით. თუ მითითებულია `last` პარამეტრი, მაშინ წაშლა მოხდება `last` ინდექსამდე. ბოლომდე წასაშლელად მეორე პარამეტრად შეიძლება Entry მნიშვნელობის გამოყენება.

განხილული მეთოდები გამოვიყენოთ პროგრამაში:

```
from tkinter import *  
from tkinter import messagebox  
def clear():
```

```

name_entry.delete(0, END)
surname_entry.delete(0, END)
def display():
    messagebox.showinfo("GUI Python", name_entry.get()
        + " " + surname_entry.get())
root = Tk()
root.title("GUI - Python")
name_label = Label(text="შეიტანეთ სახელი:")
surname_label = Label(text="შეიტანეთ გვარი:")
name_label.grid(row=0, column=0, sticky="w")
surname_label.grid(row=1, column=0, sticky="w")
name_entry = Entry()
surname_entry = Entry()
name_entry.grid(row=0, column=1, padx=5, pady=5)
surname_entry.grid(row=1, column=1, padx=5, pady=5)
# საწყისი მონაცემების ჩასმა
name_entry.insert(0, "Anna")
surname_entry.insert(0, "Karden")
display_button=Button(text="Display", command=display)
clear_button = Button(text="Clear", command=clear)
display_button.grid(row=2, column=0, padx=5, pady=5,
    sticky="e")
clear_button.grid(row=2, column=1, padx=5, pady=5,
    sticky="e")
root.mainloop()

```

პროგრამის გაშვებისას ორივე ტექსტურ ველში დაემატება ტექსტი დუმილით:

```
name_entry.insert(0, "Anna")
surname_entry.insert(0, "Karden")
```

Clear ღილაკი ასუფთავებს ორივე ველს, delete მეთოდის გამოძახებით:

```
def clear():
    name_entry.delete(0, END)
    surname_entry.delete(0, END)
```

მეორე ღილაკი get მეთოდის გამოყენებით მიიღებს შეტანილ ტექსტს:

```
def display():
    messagebox.showinfo("GUI Python", name_entry.get()
        + " " + surname_entry.get())
```

თანაც, როგორც პარამეტრიდან ჩანს არააუცილებელია Entry-ში ტექსტს მივმართოთ StringVar ტიპის ცვლადით, ეს შეგვიძლია გავაკეთოთ პირდაპირ get მეთოდით.

12.7. Checkbutton

Checkbutton ელემენტი არის ალამი, რომელსაც შეიძლება ჰქონდეს ორი მდგომარეობა: მონიშნული და მოუნიშნავი.

შევქმნათ მარტივი ალამი:

```
from tkinter import *
root = Tk()
```

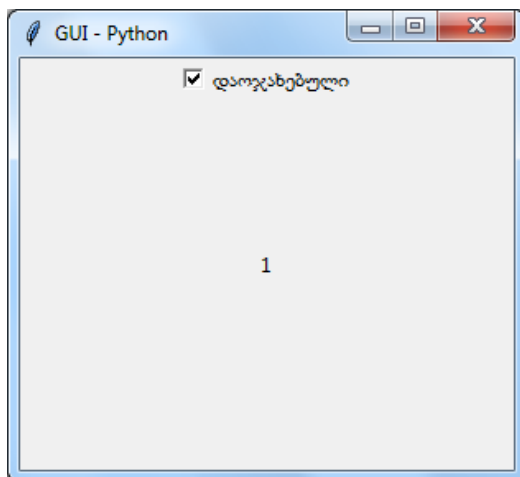


```

root.title("GUI - Python")
root.geometry("300x250")
ismarried = IntVar()
ismarried_checkbutton = Checkbutton(text="დაოჯახებული",
    variable=ismarried)
ismarried_checkbutton.pack()
ismarried_label = Label(textvariable=ismarried)
ismarried_label.place(relx=.5, rely=.5, anchor="c")
root.mainloop()

```

Checkbutton ელემენტის განხვავებული ნიშანია IntVar კომპონენტთან მიმაგრების შესაძლებლობა variable პარამეტრის მეშვეობით. მონიშნულ მდგომარეობაში IntVar დაკავშირებულ კომპონენტს აქვს 1 მნიშვნელობა, ხოლო მოუნიშნავში - 0. საბოლოო ჯამში IntVar მეშვეობით შეგვიძლია მივიღოთ მომხმარებლის მიერ მითითებული მნიშვნელობა.



Checkbox კონსტრუქტორი იღებს მთელ რიგ პარამეტრებს, რომლის დახმარებით შეიძლება აღმების ასახვის აწყობა:

Checkbox(master, options)

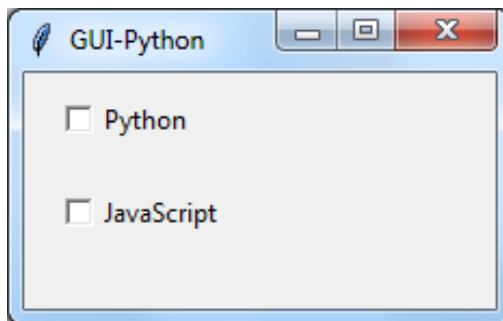
master პარამეტრი წარმოადგენს ბმულს მშობელ ფანჯარაზე, ხოლო options პარამეტრი წარმოადგენს შემდეგი პარამეტრების ნაკრებს:

- **activebackground**-აღმის ფონის ფერი დაჭერილ მდგომარეობაში;
- **activeforeground** - აღმის ტექსტის ფერი დაჭერილ მდგომარეობაში;
- **bg** - აღმის ფონის ფერი;
- **bitmap** - აღმისთვის მონოქრომული გამოსახულება;
- **bd** - საზღვარი აღმის ირგვლივ;
- **command** - ბმული ფუნქციაზე, რომელიც გამოიძახება ალაშზე დაჭერისას;
- **cursor** - კურსორი ელემენტთან მიტანისას;
- **disabledforeground** - ტექსტის ფერი DISABLED მდგომარეობაში;
- **font** - შრიფტი;
- **fg** - ტექსტის ფერი;
- **height** - ელემენტის სიმაღლე;
- **image** - გრაფიკული გამოსახულება, ელემენტზე ასახული;
- **justify** -ტექსტის გათანაბრება: CENTER, LEFT, RIGHT;

- **offvalue** - ალამთან ასოცირებული IntVar ცვლადის მნიშვნელობა მოუნიშნავ მდგომარეობაში, დუმილით ტოლია 0-ის;
- **onvalue** - ალამთან ასოცირებული IntVar ცვლადის მნიშვნელობა მონიშნულ მდგომარეობაში, დუმილით ტოლია 1-ის;
- **padx**-შეწევები ტექსტიდან მარჯვნივ და მარცხნივ ალმის საზღვრამდე;
- **pady** - შეწევები ტექსტიდან ზევით და ქვევით ალმის საზღვრამდე;
- **relief** - ალმის სტილი,დუმილით აქვს მნიშვნელობა FLAT;
- **selectcolor** - ალმის კვადრატის ფერი;
- **selectimage** - გამოსახულება ალამზე, როდესაც იგი იმყოფება მონიშნულ მდგომარეობაში;
- **state** - ელემენტის მდგომარეობა, შეიძლება მიიღოს მნიშვნელობები NORMAL (დუმილით), DISABLED და ACTIVE;
- **text** - ელემენტის ტექსტი;
- **underline** - გახაზული სიმბოლოს ინდექსი ალმის ტექსტზე;
- **variable** - ბმული ცვლადზე, როგორც წესი, IntVar ტიპი, რომელიც ინახავს ალმის მდგომარეობას;
- **width** - ელემენტის სიგანე;
- **wraplength** - ელემენტის ტექსტზე სიმბოლოების გადატანა სხვა ხაზზე.

განვიხილოთ პროგრამა ზოგიერთი განხილული პარამეტრის გამოყენებით:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
python_lang = IntVar()
python_checkbutton = Checkbutton(text="Python",
                                  variable=python_lang,
                                  onvalue=1, offvalue=0, padx=15,
                                  pady=10)
python_checkbutton.grid(row=0, column=0, sticky=W)
javascript_lang = IntVar()
javascript_checkbutton = Checkbutton(text="JavaScript",
                                      variable=javascript_lang, onvalue=1, offvalue=0,
                                      padx=15,
                                      pady=10)
javascript_checkbutton.grid(row=1, column=0, sticky=W)
root.mainloop()
```



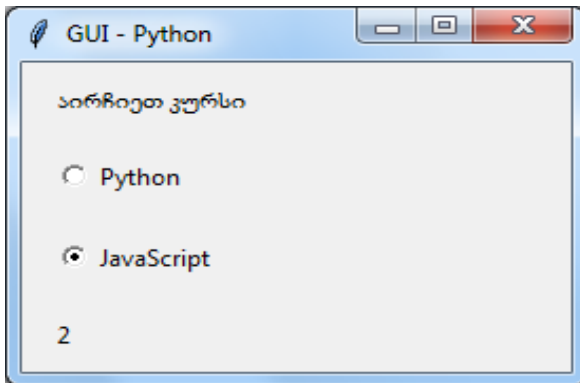
12.8. Radiobutton

Radiobutton ელემენტი არის გადამრთველი, რომელიც იმყოფება ორ მდგომარეობაში: მონიშნულ ან მონიშნავ. Checkbutton ელემენტისგან განსხვავებით შეიქმნება გადამრთველების ჯგუფი, რომელთაგან ერთდროულად შეიძლება მხოლოდ ერთი ელემენტის არჩევა.

განვიხილოთ მაგალითი:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
header = Label(text="აირჩიეთ კურსი",
               padx=15, pady=10)
header.grid(row=0, column=0, sticky=W)
lang = IntVar()
python_checkbutton = Radiobutton(text="Python",
                                  value=1, variable=lang, padx=15, pady=10)
python_checkbutton.grid(row=1, column=0, sticky=W)
javascript_checkbutton = Radiobutton(text="JavaScript",
                                      value=2, variable=lang, padx=15, pady=10)
javascript_checkbutton.grid(row=2, column=0, sticky=W)
selection = Label(textvariable=lang, padx=15, pady=10)
selection.grid(row=3, column=0, sticky=W)
root.mainloop()
```

აქ განსაზღვრულია ორი გადამრთველი, მაგრამ ორივე მიბმულია ერთ - IntVar ცვლადთან. ამავდროულად მათ აქვთ განსხვავებული მნიშვნელობები, რომლებიც დაყენდება value პარამეტრის მეშვეობით. ამიტომ ერთი გადამრთველის ჩართვისას, მეორე ავტომატურად გადავა მოუნიშნავ მდგომარეობაში.



გადამრთველის აწყობისთვის Radiobutton კონსტრუქტორი მიიღებს ორ მნიშვნელობას:

Radiobutton (master, options)

პირველი პარამეტრი - master არის ბმული მშობელ ფანჯარაზე, ხოლო მეორე პარამეტრი აერთიანებს შემდეგი პარამეტრების ნაკრებს:

- **activebackground** - გადამრთველის ფონის ფერი დაჭერილ მდგომარეობაში;
- **activeforeground** - გადამრთველის ტექსტის ფერი დაჭერილ მდგომარეობაში;
- **bg** - გადამრთველის ფონის ფერი;

- **bitmap** - გადამრთველისთვის მონოქრომული გამოსახულება;
- **borderwidth** - საზღვარი გადამრთველის ირგვლივ;
- **command** - ბმული ფუნქციაზე, რომელიც გამოიძახება გადამრთველზე დაჭერისას;
- **cursor** - კურსორი ელემენტზე მიტანისას;
- **font** - შრიფტი;
- **fg** - ტექსტის ფერი;
- **height** - ელემენტის სიმაღლე;
- **image** - ელემენტზე ასახული გრაფიკული გამოსახულება;
- **justify** - ტექსტის გათანაბრება, მიიღებს მნიშვნელობებს: CENTER, LEFT, RIGHT
- **padx** - შეწევები ტექსტიდან მარჯვნივ და მარცხნივ გადამრთველის საზღვრამდე;
- **pady** - შეწევები ტექსტიდან ზევით და ქვევით გადამრთველის საზღვრამდე;
- **relief** - გადამრთველის სტილი, დუმილით აქვს მნიშვნელობა FLAT;
- **selectcolor** - გადამრთველის წრის ფერი;
- **selectimage** - გამოსახულება გადამრთველზე, როდესაც იგი იმყოფება მონიშნულ მდგომარეობაში;
- **state** - ელემენტის მდგომარეობა, შეიძლება მიიღოს მნიშვნელობები: NORMAL (დუმილით), DISABLED და ACTIVE;
- **text** - ელემენტის ტექსტი;

- **textvariable** - დააყენებს StringVar ცვლადთან მიბმას, რომელიც იძლევა გადამრთველის ტექსტს;
- **underline** - ელემენტის ტექსტში გახაზული სიმბოლოს ინდექსი;
- **variable** - მიბმა ცვლადზე, როგორც წესი IntVar ტიპია, რომელიც ინახავს გადამრთველის მდგომარეობას;
- **value** - გადამრთველის მნიშვნელობა;
- **wraplength** - ელემენტის ტექსტში სიმბოლოების გადატანა სხვა ხაზზე.

გამოვიყენოთ განხილული პარამეტრები პროგრამაში:

```

from tkinter import *
languages = [("Python", 1), ("JavaScript", 2),
             ("C#", 3), ("Java", 4)]
def select():
    l = language.get()
    if l == 1:
        sel.config(text="არჩეულია Python")
    elif l == 2:
        sel.config(text="არჩეულია JavaScript")
    elif l == 3:
        sel.config(text="არჩეულია C#")
    elif l == 4:
        sel.config(text="არჩეულია Java")
root = Tk()
root.title("GUI - Python")

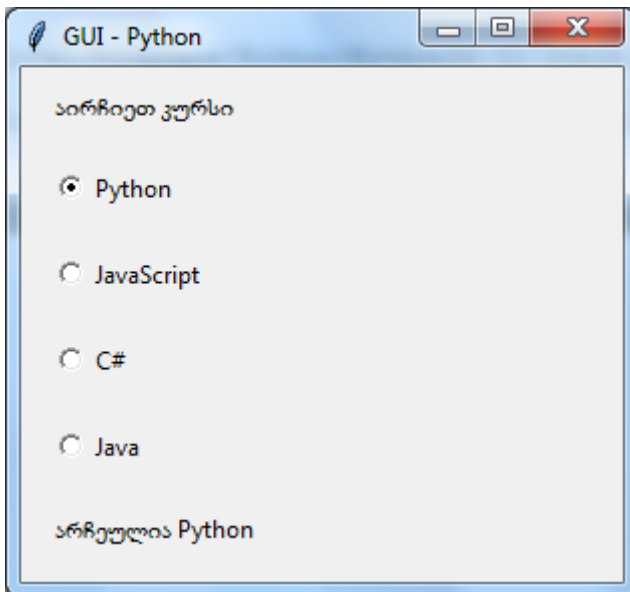
```



```

root.geometry("300x280")
header = Label(text="აირჩიეთ კურსი", padx=15,
               pady=10)
header.grid(row=0, column=0, sticky=W)
language = IntVar()
row = 1
for txt, val in languages:
    Radiobutton(text=txt, value=val, variable=language,
                padx=15, pady=10, command=select)\
        .grid(row=row, sticky=W)
    row += 1
sel = Label(padx=15, pady=10)
sel.grid(row=row, sticky=W)
root.mainloop()

```

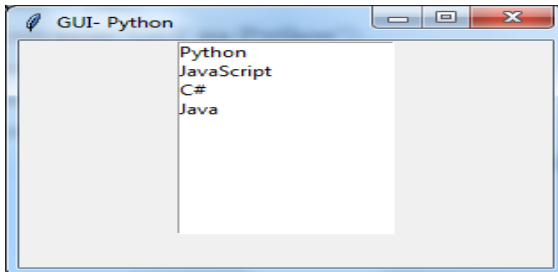


12.9. Listbox

Listbox ელემენტი წარმოადგენს ობიექტების სიას. განვსაზღვროთ მარტივი სია:

```
from tkinter import *
languages = ["Python", "JavaScript", "C#", "Java"]
root = Tk()
root.title("GUI - Python")
root.geometry("300x280")
languages_listbox = Listbox()
for language in languages:
    languages_listbox.insert(END, language)
languages_listbox.pack()
root.mainloop()
```

listbox ელემენტის შევსებისთვის for ციკლში უნდა გავიაროთ languages სიის ყველა ელემენტზე და ცალკეულ ელემენტს ვამატებთ listbox-ში insert() მეთოდით. insert()-ში პირველი არგუმენტის სახით გადაეცემა ელემენტის ჩასმის ინდექსი. თუ გვინდა ელემენტების თანმიმდევრულად დამატება, მაშინ ინდექსის ნაცვლად შეიძლება END მნიშვნელობის გამოყენება.



listbox იყენებს შემდეგ პარამეტრებს:

- **bg** - ფონის ფერი;
- **bd** - საზღვრის სისქე ელემენტის ირგვლივ;
- **cursor** - კურსორი Listbox-ზე მიტანისას;
- **font** - შრიფტის დაყენება;
- **fg** - ტექსტის ფერი;
- **height** - ელემენტის სიმაღლე სტრიქონებში; დუმილით ასახავს 10 სტრიქონს;
- **highlightcolor** - ელემენტის ფერი, როდესაც იგი იღებს ფოკუსს;
- **highlightthickness** - ელემენტის საზღვრის სისქე, როდესაც იგი ფოკუსშია;
- **relief** - დააყენებს ელემენტის სტილს, დუმილით აქვს მნიშვნელობა SUNKEN;
- **selectbackground** - გამოყოფილი ელემენტის ფონის ფერი;
- **selectmode** - განსაზღვრავს რამდენი ელემენტი შეიძლება იყოს გამოყოფილი. შეიძლება მიიღოს შემდეგი მნიშვნელობები: BROWSE, SINGLE, MULTIPLE, EXTENDED. მაგალითად, თუ საჭიროა ჩაირთოს ელემენტების მრავლობითი გამოყოფა, მაშინ შეიძლება MULTIPLE ან EXTENDED მნიშვნელობების გამოყენება;
- **width** - ელემენტების სიგანე სიმბოლოებში. დუმილით - 20 ელემენტი;
- **xscrollcommand** - ჰორიზონტალური გადახვევა;
- **yscrollcommand** - ვერტიკალური გადახვევა.

Listbox ელემენტის გამოყენებისას ზოგიერთ სირთულეს წარმოადგენს გადახვევის შექმნა. ვნახოთ როგორ უნდა ამის გაკეთება:

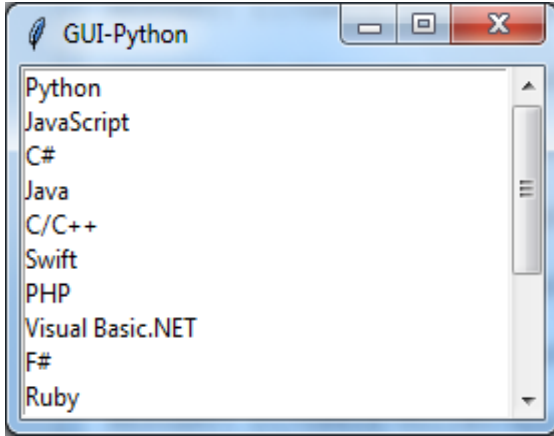
```
from tkinter import *

languages = ["Python", "JavaScript", "C#", "Java", "C/C++",
            "Swift", "PHP", "Visual Basic.NET", "F#",
            "Ruby", "Rust", "R", "Go",
            "T-SQL", "PL-SQL", "Typescript"]

root = Tk()
root.title("GUI-Python")
scrollbar = Scrollbar(root)
scrollbar.pack(side=RIGHT, fill=Y)
languages_listbox = Listbox(yscrollcommand=scrollbar.set,
                            width=40)
for language in languages:
    languages_listbox.insert(END, language)
languages_listbox.pack(side=LEFT, fill=BOTH)
scrollbar.config(command=languages_listbox.yview)
root.mainloop()
```

გადახვევის ორგანიზებისთვის იქმნება Scrollbar ელემენტი. ვერტიკალურად გადახვევისთვის მივუთითოთ პარამეტრი:

```
yscrollcommand=scrollbar.set.
```



ბოლოში scrollbar ასოცირდება ფუნქციით, რომელიც უნდა შესრულდეს გადახვევისას. მოცემულ შემთხვევაში ეს არის listbox ელემენტის yview მეთოდი. ჩვენ შევძლებთ ელემენტების გადახვევას ვერტიკალურად.

Listbox - ის ძირითადი მეთოდები

Listbox ელემენტთან სამუშაოდ გამოიყენება მთელი რიგი მეთოდები. განვიხილოთ ზოგიერთი მათგანი:

- **curselection()** - აბრუნებს გამოყოფილი ელემენტების ინდექსების ნაკრებს;
- **delete(first, last = None)** - წაშლის ელემენტებს ინდექსებით [first, last] დიაპაზონიდან. თუ მეორე პარამეტრი გამოტოვებულია, მაშინ წაშლის მხოლოდ ერთ ელემენტს first ინდექსით;
- **get(first, last = None)** - აბრუნებს კორტეჯს, რომელიც შეიცავს ელემენტების ტექსტს ინდექსებით [first, last] დიაპაზონიდან. თუ მეორე პარამეტრი გამოტოვებულია,

ლია, დაბრუნდება მხოლოდ ელემენტის ტექსტი first ინდექსით;

- **insert(index, element)** - ელემენტს ჩასვამს განსაზღვრული ინდექსით;
- **size()** - აბრუნებს ელემენტების რაოდენობას.

განვიხილოთ მაგალითი წარმოდგენილი მეთოდების გამოყენებით:

```
from tkinter import *
#გამოყოფილი ელემენტის წაშლა
def delete():
    selection = languages_listbox.curselection()
    #შევიძლია მივიღოთ წასაშლელი ელემენტი ინდექსით
    #selected_language= languages_listbox.get(selection[0])
    languages_listbox.delete(selection[0])
    #ახალი ელემენტის დამატება
def add():
    new_language = language_entry.get()
    languages_listbox.insert(0, new_language)
root = Tk()
root.title("GUI - Python")
#ტექსტური ველი და ღილაკის სიაში დასამატებლად
language_entry = Entry(width=40)
language_entry.grid(column=0, row=0, padx=6, pady=6)
add_button = Button(text="დამატება",
                    command=add).grid(column=1,
                                       row=0, padx=6, pady=6)
```

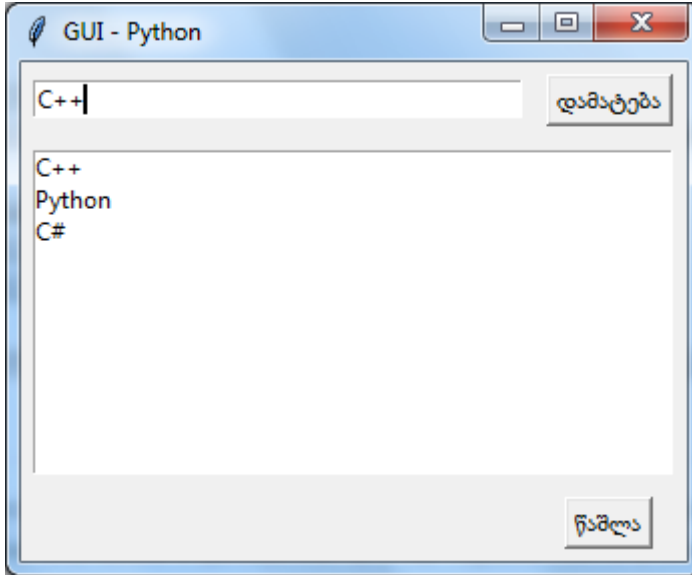
```

# ვქმნით სიას
languages_listbox = Listbox()
languages_listbox.grid(row=1, column=0, columnspan=2,
                        sticky=W+E, padx=5, pady=5)
# სიაში ვამატებთ საწყის ელემენტებს
languages_listbox.insert(END, "Python")
languages_listbox.insert(END, "C#")
delete_button = Button(text="წაშლა",
                        command=delete).grid(row=2,
                        column=1, padx=5, pady=5)
root.mainloop()

```

სიის ელემენტის მანიპულირებისთვის განსაზღვრულია ორი ღილაკი. პირველი ღილაკი იძახებს add() ფუნქციას, რომელიც იღებს ტექსტურ ველში შეტანილ მნიშვნელობას და ამატებს მას სიაში პირველ ადგილზე insert() მეთოდის გამოყენებით.

მეორე ღილაკი დაჭერისას წაშლის გამოყოფილ ელემენტს. ამისათვის ჩვენ თავდაპირველად ვიღებთ გამოყოფილ ინდექსებს curselection() მეთოდით. რამდენადაც ჩვენ შემთხვევაში გამოიყოფა მხოლოდ ერთი ელემენტი, ამიტომ ვიღებთ მის ინდექსს selection[0] გამოსახულების მეშვეობით და ამ ინდექსს გადავცემთ delete() მეთოდს წაშლისთვის.



12.10. მენიუს შექმნა

იერარქიული მენიუს შესაქმნელად გამოიყენება Menu პროგრამული მოდული. მენიუ შეიძლება შეიცავდეს მრავალ ელემენტს, თანაც ეს ელემენტები შეიძლება თვითონ წარმოადგენდეს მენიუს და შეიცავდეს სხვა ელემენტებს. მეთოდი შეირჩევა იმის მიხედვით, თუ ელემენტების რომელი ტიპი უნდა დაემატოს მენიუს. კერძოდ, მისაწვდომია შემდეგი მეთოდები:

- **add_command(options)** - მენიუში დაამატებს ელემენტს options პარამეტრის მეშვეობით;
- **add_cascade(options)** - მენიუში დაამატებს ელემენტს, რომელიც თავის მხრივ შეიძლება წარმოადგენდეს ქვემენიუს;

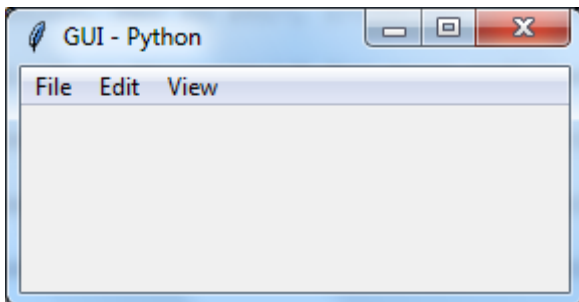
- **add_separator()** - დაამატებს გამყოფ ხაზს;
- **add_radiobutton(options)**- მენიუში დაამატებს გადამრთველს;
- **add_checkbutton(options)** - მენიუში დაამატებს ალამს.

შევქმნათ მარტივი მენიუ:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
main_menu = Menu()
main_menu.add_cascade(label="File")
main_menu.add_cascade(label="Edit")
main_menu.add_cascade(label="View")
root.config(menu=main_menu)
root.mainloop()
```

მენიუს პუნქტების დასამატებლად გამოიძახება `add_cascade()` მეთოდი. ამ მეთოდში გადაეცემა მენიუს პუნქტის პარამეტრები, მოცემულ შემთხვევაში ისინი წარმოდგენილია ტექსტური ჭდით, რომლებიც დაყენდება `label` პარამეტრის მეშვეობით.

მხოლოდ მენიუს შექმნა არასაკმარისია. საჭიროა მისი დაყენება მიმდინარე ფანჯრისთვის `menu` პარამეტრის დახმარებით `config()` მეთოდში. საბოლოოდ, გრაფიკულ ფანჯარას ექნება შემდეგი მენიუ:

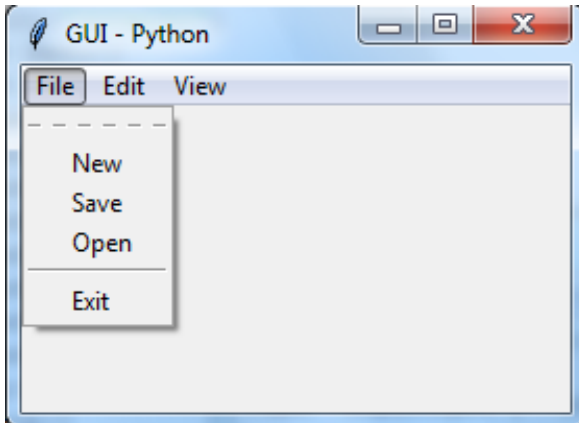


დავამატოთ ქვემენიუ:

```
from tkinter import *
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
main_menu = Menu()
file_menu = Menu()
file_menu.add_command(label="New")
file_menu.add_command(label="Save")
file_menu.add_command(label="Open")
file_menu.add_separator()
file_menu.add_command(label="Exit")
main_menu.add_cascade(label="File", menu=file_menu)
main_menu.add_cascade(label="Edit")
main_menu.add_cascade(label="View")
root.config(menu=main_menu)
root.mainloop()
```

მაგალითში განსაზღვრულია file_menu ქვემენიუ, რომელიც ემატება ძირითადი მენიუს პირველ პუნქტს menu=file_menu ოპციით:

```
main_menu.add_cascade(label="File", menu=file_menu)
```

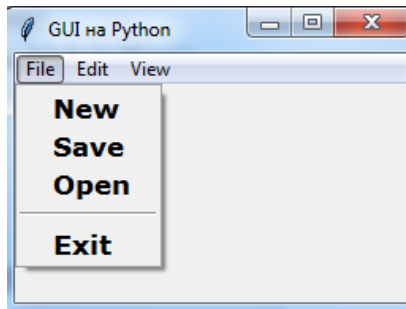


მენიუს ასაწყობად Menu კონსტრუქტორში შეიძლება გამოვიყენოთ შემდეგი ოპციები:

- **activebackground** - მენიუს აქტიური პუნქტის ფერი;
- **activeborderwidth** - მენიუს აქტიური პუნქტის საზღვრის სისქე;
- **activeforeground** - მენიუს აქტიური პუნქტის ტექსტის ფერი;
- **bg** - ფონის ფერი;
- **bd** - საზღვრის სისქე;
- **cursor** - თავის მარჯვენა კურსორი მენიუსთან მიტანისას;
- **disabledforeground** - ფერი, როდესაც მენიუ იმყოფება DISABLED მდგომარეობაში;
- **font:** - ტექსტის შრიფტი;
- **fg** - ტექსტის ფერი;

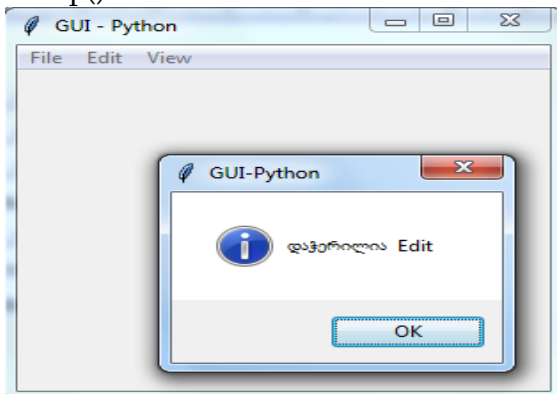
- **tearoff** - მენიუ შეიძლება ჩაიხსნას გრაფიკული ფანჯრიდან. კერძოდ, ქვემენიუს ზევით ჩანს წყვეტილი ხაზი, რომლითაც იგი შეიძლება ჩაიხსნას. თუ `tearoff=0` ქვემენიუ არ ჩაიხსნება.

```
from tkinter import *
root = Tk()
root.title("GUI-Python")
root.geometry("300x250")
main_menu = Menu()
file_menu = Menu(font=("Verdana", 13, "bold"), tearoff=0)
file_menu.add_command(label="New")
file_menu.add_command(label="Save")
file_menu.add_command(label="Open")
file_menu.add_separator()
file_menu.add_command(label="Exit")
main_menu.add_cascade(label="File", menu=file_menu)
main_menu.add_cascade(label="Edit")
main_menu.add_cascade(label="View")
root.config(menu=main_menu)
root.mainloop()
```



მენიუს ელემენტების განმასხვავებელი თავისებურებაა, რომ იგი უნდა რეაგირებდეს დაწკაპებაზე. ამისათვის მენიუს ცალკეულ ელემენტს შეიძლება მიეთითოს command პარამეტრი, რომელიც დააყენებს ბმულს ფუნქციაზე, რომელიც უნდა შესრულდეს დაწკაპებისას.

```
from tkinter import *
from tkinter import messagebox
def edit_click():
    messagebox.showinfo("GUI-Python", "დაჭერილია Edit")
root = Tk()
root.title("GUI - Python")
root.geometry("300x250")
main_menu = Menu()
main_menu.add_cascade(label="File")
main_menu.add_cascade(label="Edit",
    command=edit_click)
main_menu.add_cascade(label="View")
root.config(menu=main_menu)
root.mainloop()
```



ლიტერატურა

1. Charles R. Severance, Python for Everybody, Exploring Data Using Python 3, 2016, 247 pp;
2. Andrew Johansen, Python The Ultimate Beginner's Guide, 2016, 79pp;
3. Dave Kuhlman, A Python Book: Beginning Python, Advanced Python, and Python Exercises, 2013, 278pp;
4. С. Шапошникова, Основы программирования на Python, 2011, 44с;
5. Сузи Роман Авриевич, Язык программирования Python, Курс лекций (PDF для hunger.ru от nerezus'a);
6. John M. Zelle, Python Programming: An Introduction to Computer Science, 2002, 261pp;
7. Г. Россум, Ф.Л.Дж. Дрейк, Д. С. Откидач, Язык программирования Python, 2001, 454 с;
8. <https://younglinux.info/oopython/operators.php>