შესავალი

თუ გიმუშავიათ კომპიუტერზე, გეცოდინებათ, რომ არსებობს ამოცანები, რომელთა ავტომატიზაციაც არის საჭირო. მაგალითად თქვენ შეიძლება დაგჭირდეთ დიდი რაოდენობა ფოტო ფაილებისათვის სახელების შეცვლა, გრაფიკული პროგრამის დაწერა, თამაშის შექმნა და ა.შ.

ასეთი ამოცანებისთვის თქვენ შეგიძლიათ გამოიყენოთ პროგრამირების ენა პითონი.

ზოგიერთი მსგავსი ტიპის პროგრამების დასაწერად თქვენ შეგიძლიათ გამოიყენოთ უნიქსის "shell script"-ი ან ვინდოუსის "batch"-ფაილი. მაგრამ მათ ვერ გამოიყენებთ გრაფიკული ფანჯრების ან თამაშის დასაწერად. თქვენ შეიძლება დაწეროთ პროგრამა C/C++ ან Java-ზე, მაგრამ ამისთვის დაგჭირდებათ ბევრად უფრო მეტი დრო, ვიდრე იგივე პროგრამის პითონზე დაწერისას. პითონი არის მარტივი გამოსაყენებელი და გვეხმარება პროგრამის სწრაფად დაწერაში. პითონის გამოყენება შეიძლება "Windows"-ზე, "Mac OS X"-ზე და "Unix"-ზე.

პითონს გააჩნია მაღალი დონის მონაცემთა ტიპები, როგორიცაა მასივები და ლექსიკონები და შეცდომების შემოწმების უკეთესი სისტემა ვიდრე C-ს.

პითონი პროგრამის მოდულებად დაყოფის საშუალებას იძლევა, რომლებიც შეიძლება გამოყენებულ იქნას სხვა პითონის პროგრამის მიერ. პითონს მოყვება დიდი კოლექცია სტანდარტული მოდულების. მაგალითად სოკეტები, ფაილში ჩაწერა წაკითხვისთვის საჭირო მოდულები და ა.შ.

პითონი არის ინტერპრეტირებადი ენა. ინტერპრეტატორი შეიძლება გამოყენებულ იქნას ინტერაქტიულ რეჟიმში, რაც ამარტივებს ექსპერიმენტების ჩატარებას.

პითონზე დაწერილი პროგრამა არის კომპაქტური, ადვილად წაკითხვადი და მცირე ზომის, C, C++ ან Java-ზე დაწერილ პროგრამებთან შედარებით.

შემდეგ თავში გავეცნობით ინტერპრეტატორის გამოყენების მექანიზმს.

პითონის ინტერპრეტატორის გამოყენება

ვიწერთ პითონის ინტერპრეტატორს: http://www.python.org/ და ვაყენებთ.
(როგორც წესი ყენდება "C://python27"-ზე).

ამის შემდეგ შევდივართ Start-ში მაუსის მარჯვენა ღილაკით ვაწვებით Computer-ზე, და ვირჩევთ properties. გამოსულ ფანჯარაში ვირჩევთ "Advanced System Settings", შემდეგ ვირჩევთ Environment Variables, ვირჩევთ Path-ს და ვამატებთ ჩანაწერს C:\python27; ამის შემდეგ cmd-ში შეგვიძლია ავკრიფოთ python და ამით მივწვდეთ პითონის ინტერპრეტატორს. ინტერპრეტატორიდან გამოსვლა შეიძლება "Ctrl-Z Enter"-ღილაკების კომბინაციით.

პითონის ინტერაქტიულ რეჟიმში გამოსაყენებლად უბრალოდ უნდა ავკრიფოთ: python (cmd-ში).

პითონზე დაწერილი პროგრამის გასაშვებად უნდა ავკრიფოთ: python test.py, სადაც test.py არის გასაშვები პროგრამა.

თუ გვინდა, პროგრამისთვის არგუმენტების გადაცემა ვწერთ:

python test.py arg1, arg2, ..

ახლა დავწეროთ ჩვენი პირველი პროგრამა პითონზე, შევქმნათ test.py ფაილი და ავკრიფოთ შემდეგი კოდი:

პროგრამული კოდი

```
import sys
  print ("hello " + sys.argv[1])
```

შევასრულოთ პროგრამა: python test.py world

ეკრანზე გამოვა შეტყობინება: hello world

ენა და სინტაქსი

1. იდენტიფიკატორები და ცვლადების გამოცხადება

პითონში განსხვავებით ისეთი ენებისაგან, როგორიცაა ჯავა, ცვლადების გამოცხადებისას არ მიეთითება ცვლადის ტიპი, რაც იმას ნიშნავს, რომ ასეთი ცვლადი შეიძლება შეიცავდეს ნებისმიერი ტიპის მნიშვნელობას.

მაგალითად:

ჯავა-ში ცვლადი აღიწერება შემდეგნაირად

```
int x = 0;
```

ხოლო პითონში გვექნება:

```
x = 0
```

ან

```
x = "Hello World"
```

მარტივი მაგალითი პითონში ცვლადების გამოყენებისა:

პროგრამული კოდი

```
>>> x = 3
>>> y = 3.14
>>> x = x*y
>>> print (x)
9.42
```

2. კოდის სტრუქტურა

განსხვავებით ჯავასაგან რომელიც ბლოკის აღსაწერად იყენებს ფრჩხილებს, პითონი იყენებს ცარიელ სიმბოლოებს(ჰარი), რაც კოდს ხდის ადვილად წასაკითხს და აგრეთვე თავიდან გვაშორებს ზედმეტ სიმბოლოებს.

მაგალითი:

ჯავა-ზე გვაქვს

პროგრამული კოდი

```
int x = 100;
  if(x > 0){
     System.out.print("Hello World");
  }
  else {
     System.out.print("Hello Bob");
  }
```

პითონზე იგივე კოდი ჩაიწერება შემდეგნაირად

პროგრამული კოდი

```
x = 100
```

```
if x > 100:
    print("Hello World")
else:
    print("Hello Bob")
```

3. ოპერატორები

პითონში გამოიყენება იგივე ოპერატორები, რაც ნებისმიერ სხვა ენაში ეს არის +,-,*,/.

4. გამოსახულებები

გამოსახულებების მაგალითები:

პროგრამული კოდი

```
>>> x + y
>>> x - y
>>> x * y
>>> x / y
```

5. ფუნქციები

ფუნქციების აღსაწერად გამოიყენება def სიტყვა

მაგალითი აღვწეროთ ფუნქცია hello:

პროგრამული კოდი

```
>>> def hello():
...     print ("Hello World")
...
>>> hello()
Hello World
```

მაგალითი 2:

პროგრამული კოდი

```
>>> def multiply_nums(x, y):
...     return x * y
...
>>> multiply_nums(10, 12)
120
```

პითონში ფუნქცია შეიძლება გამოყენებულ იქნას, როგორც ცვლადი. მაგალითად ერთი ფუნქცია შეიძლება გადავცეთ მეორეს არგუმენტად.

მაგალითი:

პროგრამული კოდი

```
>>> def perform_math(oper):
```

```
...     return oper(5, 6)
...
>>> perform_math(multiply_nums)
30
```

6. კლასები

პითონი არის ობიექტზე ორიენტირებული ენა, რაც ნიშნავს, რომ პითონში ყველაფერი არის გარკვეული ტიპის ობიექტი. კლასები პითონში განისაზღვრება class გასაღები სიტყვის გამოყენებით. კლასები შეიძლება შეიცავდნენ ფუნქციებს, მეთოდებს და ცვლადებს. მეთოდი არის იგივე რაც ფუნქცია, იმ განსხვავებით, რომ მეთოდებს ავტომატურად გადაეცემათ self ცვლადი, რომელიც მიუთითებს იმ ობიექტს რომელსაც ეკუთვნის ეს მეთოდი. კლასებს გააჩნიათ აგრეთვე სპეციალური მეთოდი(კონსტრუქტორი), რომელიც გამოიძახება, ამ კლასის ობიექტის შექმნის დროს.

განვიხილოთ მაგალითი:

პროგრამული კოდი

```
>>> class my_object:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
```

```
...     def mult(self):
...         print (self.x*self.y)
...
...     def add(self):
...         print (self.x + self.y)
...
>>> obj1 = my_object(7, 8)
>>> obj1.mult()
56
>>> obj1.add()
15
```

როგორც ვხედავთ მოცემული გვაქვს my_object კლასი, რომელსაც გააჩნია __init__-მეთოდი,

რომელიც გამოიძახება ამ კლასის ობიექტის შექმნისას და გადაეცემა ორი არგუმენტი x და y.

კლასის შიგნით, შესაძლებელია, კონსტრუქტორში გამოცხადებული ცვლადების გამოყენება.

მაგალითში ჩანს რომ ჩვენ ვქმნით obj1-ობიექტს. x და y ცვლადებზე წვდომა შეიძლება

შემდეგნაირად obj1.x, obj1.y

obj1-იდენტიფიკატორის გამოყენებით უკვე შეიძლება კლასის მეთოდებსა და ფუნქციებზე წვდომა.

7. if-elif-else ოპერატორი

მოცემული ოპერატორი გამოთვლის მასში შემავალ გამოსახულებას და შეასრულებს მოქმედებას

იმის მიხედვით თუ რა არის გამოსახულების მნიშვნელობა True თუ False-ი.

ამ ოპერატორის ფსევდოკოდი გამოიყურება შემდეგნაირად:

პროგრამული კოდი

```
if <გამოსახულება>:
    შეასრულე მოქმედება
 else:
    შეასრულე სხვა მოქმედება
```

მაგალითი:

პროგრამული კოდი

```
>>> x = 3
 >>> y = 2
 >>> if x == y:
 ...     print ('x is equal to y')
 ... elif x > y:
 ...     print ('x is greater than y')
 ... else:
 ...     print ('x is less than y')
 ...
 x is greater than y
```

8. print ოპერატორი

print ოპერატორი გამოიყენება ტექსტის ეკრანზე გამოსატანად.

მაგალითი:

პროგრამული კოდი

```
>>> print("Hello World")
  Hello World
```

აგრეთვე ჩვენ შეგვიძლია გამოვიყენოთ + ოპერაცია ორი სტრიქონის ერთმანეთთან დასაკავშირებლად.

მაგალითი:

პროგრამული კოდი

```
>>> print("Hello " + "World")
  Hello World
```

მაგრამ თუ ვეცდებით შემდეგი კოდის შესრულებას მივიღებთ შეცდომას.

პროგრამული კოდი

```
>>> x = 2
  >>> print("2 bread costs " + x)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: Can't convert 'int' object to str implicitly
```

იმისათვის რომ int ტიპის მონაცემი მივაბათ სტრიქონს უნდა გამოვიყენოთ %d ფორმატი.

პროგრამული კოდი

```
>>> x = 2
 >>> print("2 bread costs %d$" % x)
 2 bread costs 2$
```

ინტერპრეტატორი %d-ს ადგილას ჩასვამს x-ის მნიშვნელობას, 2-იანს.

სხვა ფორმატები:

%s - სტრიქონი

%d - მთელი

%f - ნამდვილი(float ტიპი)

9. try-except-finally

შეცდომების დამუშავების არსი მდგომარეობს იმაში, რომ ჩვენ ვცდილობთ გავუშვათ პროგრამის გარკვეული ფრაგმენტი და შეცდომის მოხდენის შემთხვევაში მოვახდინოთ რეაგირება, მაგალითად დავბეჭდოთ შეტყობინება ეკრანზე შეცდომის შესახებ.

პროგრამული კოდი

```
>>> x = 8
 >>> y = 0
 >>> try:
 ...    print ('The rocket trajectory is: %f' % (x/y))
 ... except:
 ...    print ("Houston, we have a problem....")
 ...
 Houston, we have a problem....
```

ნულზე გაყოფის მცდელობისას გამოისროლება error-ი, რომელსაც ჩვენ ვიჭერთ try, except ბლოკის გამოყენებით და ვახდენთ რეაგირებას. ამ შემთხვევაში ვბეჭდავთ შეტყობინებას შეცდომის შესახებ.

10. raise ოპერატორი

raise ოპერატორი გამოიყენება იმისთვის, რომ გამოვისროლოთ exception-ი.

მაგალითი:

პროგრამული კოდი

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

თუ გვინდა, რომ მივუთითოთ საკუთარი შეტყობინება შეცდომის გამოსროლისას ვიქცევით შემდეგნაირად.

პროგრამული კოდი

```
>>> raise Exception('Custom Exception')
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  Exception: Custom Exception
```

11. import ოპერატორი

პროგრამის შესანახად, რომელიც გამოყენებული უნდა იქნას მოგვიანებით, ჩვენ ვიყენებთ ფაილებს, რომელთაც გააჩნიათ .py გაფართოება მაგალითად my_code.py

ასეთ ფაილებს პითონში უწოდებენ მოდულებს.

import ოპერატორი გამოიყენება იმისთვის, რომ ჩატვირთოს მოდული ან კოდი ოპერატიულ მეხსიერებაში, რათა გამოყენებულ იქნას ჩვენი პროგრამის მიერ.

იმისათვის, რომ მოვახდინოთ მოდული TipCalculator-ის იმპორტი ვასრულებთ კოდს:

პროგრამული კოდი

```
# მოდულის იმპორტი სახელად TipCalculator
  import TipCalculator
```

თუ გვინდა, რომ მოდულიდან მოვახდინოთ იმპორტირება მაგალითად რაიმე ფუნქციის მაშინ:

პროგრამული კოდი

```
# იმპორტი tipCalculator ფუნქციის მოდულიდან, სახელად ExternalModule.py
  from ExternalModule import tipCalculator
```

12. იტერაცია

სია(list) პითონში არის კონტეინერი, რომელიც ინახავს ობიექტებს.

მაგალითი:

პროგრამული კოდი

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

ამ მაგალითში ჩვენ განვსაზღვრეთ სია my_numbers და შემდეგ წავიკითხეთ ამ სიის მეორე ელემენტი.

მოცემული სიის ელემენტებზე იტერაცია, შეიძლება განხორციელდეს for ციკლის გამოყენებით.

პროგრამული კოდი

```
>>> for value in my_numbers:
...     print (value)
...
1
2
3
4
5
```

აგრეთვე სიაზე იტერაცია, შეიძლება განხორციელდეს while-ციკლის გამოყენებით.

პროგრამული კოდი

```
>>> x = 0
>>> while x < len(my_numbers):
...     print (my_numbers[x])
...     x = x + 1
...
1
2
3
```

4

5

სადაც len-ფუნქცია აბრუნებს სიაში შემავალი ელემენტების რაოდენობას.

13. while ციკლი

while ციკლი მუშაობს მანამ, სანამ მასში შემავალი გამოსახულების პირობა არის ჭეშმარიტი.

while ციკლის ფსევდოკოდს აქვს ფორმა.

პროგრამული კოდი

```
while True:
    perform operation
```

while ციკლის მაგალითი ჯავაზე:

პროგრამული კოდი

```
int x = 9;
 int y = 2;
 int z = x - y;

 while (y < x){
    System.out.println("y is " + z + " less than x");
```

```
    y = y++;
}
```

იგივე კოდი პითონზე ჩაიწერება შემდეგნაირად:

პროგრამული კოდი

```
>>> x = 9
>>> y = 2
>>> while y < x:
...     print ('y is %d less than x' % (x-y))
...     y += 1
...
y is 7 less than x
y is 6 less than x
y is 5 less than x
y is 4 less than x
y is 3 less than x
y is 2 less than x
y is 1 less than x
```

14. for ციკლი

for ციკლი გამოიყენება მონაცემთა სტრუქტურის ელემენტებზე იტერაციისათვის.

for ციკლის ფსევდოკოდს აქვს ფორმა

პროგრამული კოდი

```
for each value in this defined set:
    perform suite of operations
```

for ციკლის მაგალითი ჯავაზე:

პროგრამული კოდი

```
for (x = 0; x <= 10; x++){
    System.out.println(x);
 }
```

იგივე კოდი პითონზე ჩაიწერება შემდეგნაირად.

პროგრამული კოდი

```
>>> for x in range(10):
 ...     print x
 ...
 0
 1
 2
```

3

4

5

6

7

8

9

15. კლავიატურიდან მონაცემების წაკითხვა

კლავიატურიდან მონაცემების წასაკითხად გამოიყენება input - ფუნქცია

მაგალითი:

პროგრამული კოდი

```
>>> name = input("Enter Your Name:")
 Enter Your Name: josh
 >>> print name
 josh
```

16. კოდის დოკუმენტირება

კოდის დოკუმენტაციისათვის ჩვენ შეგვიძლია გამოვიყენოთ # კომენტარი.

მაგალითი:

პროგრამული კოდი

```
>>> # This is a line of documentation
  >>> x = 0 # This is also documentation
  >>> y = 20
  >>> print x + y
  20
```

პითონის პარსერი უგულებელყოფს მონაცემებს დაწყებული # ნიშნიდან.

პითონს აგრეთვე გააჩნია მრავალ სტრიქონიანი კომენტარი, რომელიც იწყება ''' ნიშნით და მთავრდება იგივე '''. ნიშნით

მაგალითი:

პროგრამული კოდი

```
def mult(x1, x2):
      '''mocemuli funqcia,
         amravlebs or ricxvs.'''
      return x1*x2
```

ამის შემდეგ ჩვენ შეგვიძლია გამოვიყენოთ ფუნქცია help, რომელიც მოგვაწვდის ინფორმაციას

ამ ფუნქციის შესახებ.

მაგალითი:

პროგრამული კოდი

```
>>> help(mult)
 Help on function mult in module __main__:

 mult(x1, x2)
     mocemuli funqcia,
     amravlebs or ricxvs
```

მონაცემთა ტიპები და მითითება

ჩვენ განვსაზღვრავთ პროგრამებს, რომლებიც მუშაობენ მონაცემებთან. აგრეთვე ჩვენ გვჭირდება

კონტეინერები, რომლებიც შეინახავენ ამ მონაცემებს. მონაცემებს ჩვენ შეიძლება ვკითხულობდეთ

კლავიატურიდან, მონაცემთა ბაზიდან და ა.შ.

ამ თავში ჩვენ განვიხილავთ მონაცემთა ტიპებს და მათ გამოყენებას.

შევადარებთ ერთმანეთს სხვადასხვა ტიპის კონტეინერებს და მოვიყვანთ მაგალითებს, თუ რომელი კონტეინერი გამოვიყენოთ სხვადასხვა ტიპის მონაცემებთან სამუშაოდ. არსებობს ამოცანები, რომლებიც შეიძლება ამოიხსნას სიების და ლექსიკონების გამოყენებით და ჩვენ შევეცდებით განვიხილოთ ისინი. ამ კონტეინერების გამოყენების შესწავლის შემდეგ ჩვენ ვისაუბრებთ იმის შესახებ, თუ რა ხდება, როცა ეს კონტეინერები აღარაა საჭირო პროგრამისთვის.

1. პითონის მონაცემთა ტიპები

როგორც ვთქვით პროგრამა იყენებს მონაცემებს.

ჩვენ ვქმნით კონტეინერს იმისათვის, რომ შევინახოთ ეს მონაცემები.

პითონის მონაცემთა ტიპებია: None, int, long, float, complex, Boolean, Sequence, Mapping, Set, File, Iterator და ა.შ.

ახლა განვიხილოთ როგორ შეიძლება, გამოვაცხადოთ ცვლადი პითონში.

პროგრამული კოდი

```
# გამოვაცხადოთ სტრიქონი(String)
 x = 'Hello World'
 x = "Hello World Two"

 #  გამოვაცხადოთ მთელი(integer).
 y = 10

 #  Float
```

```
z = 8.75


# Complex
i = 1 + 8.07j
```

ობიექტის ტიპის განსასაზღვრავად გამოიყენება type()-ფუნქცია.

პროგრამული კოდი

```
# ობიექტის ტიპის განსაზღვრა type ფუნქციის გამოყენებით.
>>> i = 1 + 8.07j
>>> type(i)
<type 'complex'>
>>> a = 'Hello'
>>> type(a)
<type 'str'>
```

ერთ-ერთი კარგი ფუნქცია, რომელიც გააჩნია პითონს არის მრავალჯერადი მინიჭება

პროგრამული კოდი

```
>>> x, y, z = 1, 2, 3
>>> print x
1
>>> print z
```

3

>>>

2. სტრიქონები

სტრიქონი პითონში არის უცვლელი ობიექტი, რაც იმას ნიშნავს, რომ სტრიქონის შექმნის შემდეგ, მისი მნიშვნელობა პროგრამის მუშაობის განმავლობაში რჩება უცვლელი. სტრიქონი არის სიმბოლოების მიმდევრობა.

პითონში სტრიქონი არის ორი სახის standart და unicode.
სტრიქონებზე განსაზღვრულია დიდი რაოდენობით ფუნქციები, რომლებიც მუშაობენ ორივე ტიპის სტრიქონზე.

მაგალითი:

პროგრამული კოდი

```
our_string='python is the best language ever'
 # სტრიქონის პირველი სიმბოლოს ზედა რეგისტრში გადაყვანა.
 >>> our_string.capitalize()
 'Python is the best language ever'

 # სტრიქონის ცენტრირება.
```

```
>>> our_string.center(50)
'         python is the best language ever          '

>>> our_string.center(50,'-')
'---------python is the best language ever---------'

# ვამოწმებთ რამდენ 'a' სტრიქონს შეიცავს our_string სტრიქონი.
>>> our_string.count('a')
2

# სტრიქონში შემავალი 'ss' ქვესტრიქონის რაოდენობა.
>>> state = 'Mississippi'
>>> state.count('ss')
2

# სტრიქონის დაყოფა 3 ნაწილად, პირველი ნაწილი შეიცავს,
# სტრიქონს გამყოფ სტრიქონამდე, მეორე ნაწილი შეიცავს გამყოფ სტრიქონს,
# ხოლო მესამე, გამყოფი სტრიქონის შემდგომ სტრიქონს.
>>> x = "Hello, my name is Josh"
>>> x.partition('n')
('Hello, my ', 'n', 'ame is Josh')

# სტრიქონის დაყოფა ნაწილებად, 'l' გამყოფი სტრიქონის მიხედვით.
>>> x.split('l')
```

['He', '', 'o, my name is Josh']

სტრიქონის split მეთოდით გაყოფისას, შეგვიძლია მივუთითოთ maxsplits

პარამეტრის მნიშვნელობა. თუ მისი მნიშვნელობაა 1, მაშინ სტრიქონი გაიყოფა

ორ ნაწილად.

>>> x.split('l',1)

['He', 'lo, my name is Josh']

>>> x.split('l',2)

['He', '', 'o, my name is Josh']

სტრიქონი შეიძლება დავბეჭდოთ სხვადასხვა გზის გამოყენებით:

პროგრამული კოდი

სტრიქონის დასაბეჭდად შეგვიძლია გამოვიყენოთ ქვემოთ მოცემული გზები.

>>> x = "Josh"

>>> print "My name is %s" % (x)

My name is Josh

>>> print "My name is %s" % x

My name is Josh

მაგალითი, როცა გვაქვს ერთ არგუმენტზე მეტი.

>>> name = 'Josh'

```
>>> language = 'Python'
>>> print "My name is %s and I speak %s" % (name, language)
My name is Josh and I speak Python


# და ბოლოს სხვადასხვა ტიპის მონაცემების დაბეჭდვის ილუსტრაცია.
>>> day1_temp = 65
>>> day2_temp = 68
>>> day3_temp = 84
>>> print "Given the temparatures %d, %d, and %d, the average would be %f"
%(day1_temp, day2_temp, day3_temp, (day1_temp + day2_temp + day3_temp)/3)
Given the temperatures 65, 68, and 83, the average would be 72.333333
```

3. სიები

მოცემული კონტეინერები განსხვავებით სტრიქონებისაგან არიან ცვლადები(მუტირებადი).

სიები

სიები არის ყველაზე ხშირად გამოყენებადი პითონის პროგრამებში.

სია შეიძლება შეიცავდეს ნებისმიერი ტიპის ობიექტს.

სხვა ენებში როგორც წესი სია ხშირად განისაზღვრება როგორც ერთი კონკრეტული ტიპის ობიექტების შემცველი ობიექტი.

ახლა ვნახოთ როგორ შეიძლება შევქმნათ სია.

პროგრამული კოდი

```
# ცარიელი სიის გამოცხადება

my_list = []

my_list = list()  # გამოიყენება იშვიათად


# ერთ ელემენტიანი სია

>>> my_list = [1]

>>> my_list  #არ არის print-ის გამოყენების საჭიროება.

[1]


# სტრიქონების შემცველი სიის გამოცხადება

my_string_list = ['Hello', 'Jython' ,'Lists']


# სიის გამოცხადება, რომელიც შეიცავს სხვადასხვა ტიპის ობიექტებს

multi_list = [1, 2, 'three', 4, 'five', 'six']


# სიის გამოცხადება, რომელიც შეიცავს სხვა სიას.

combo_list = [1, my_string_list, multi_list]

>>> my_new_list = ['new_item1', 'new_item2', [1, 2, 3, 4], 'new_item3']

>>> print my_new_list

['new_item1', 'new_item2', [1, 2, 3, 4], 'new_item3']
```

იმისათვის რომ მივწვდეთ სიის ელემენტს, საჭიროა ინდექსის გამოყენება.

list[index]

თუ გვინდა მივწვდეთ ელემენტების სიმრავლეს სიაში საჭიროა საწყისი და საბოლოო ინდექსების მითითება

(დაბრუნებულ სიას ქვია ნაჭერი(slice)).

აგრეთვე ჩვენ შეგვიძლია მივწვდეთ სიის ელემენტებს საწყისი და საბოლოო ინდექსების მითითებით და ბიჯის მითითებით.

ამ ოპერაციების შესრულებისას ყოველთვის ბრუნდება ახალი სია, რომელიც არის ზედაპირული ასლი ძველი სიის,

რაც ნიშნავს შემდეგს, რომ იქმნება ახალი სია, რომელიც მიუთითებს იგივე ელემენტებს რასაც მიუთითებდა

ძველი სია.

სიის ღრმა ასლი, არის სია, რომელიც მიუთითებს ძველი სიაში შემავალი ობიექტების ასლებს.

პროგრამული კოდი

```
# სიიდან ელემენტების წაკითხვა
>>> my_string_list[0]
'Hello'
>>> my_string_list[2]
'Lists'

# სიის ბოლო ელემენტი
>>> my_string_list[-1]
'Lists'
```

სიაში ბოლოდან მეორე ელემენტი

```
>>> my_string_list[-2]
'Jython'
```

#წავიკითხოთ სიის პირველი ორი ელემენტი

```
>>> my_string_list[0:2]
['Hello', 'Jython']
```

#ვქმნით სიის ზედაპირულ ასლს.

```
>>> my_string_list_copy = my_string_list[:]
>>> my_string_list_copy
['Hello', 'Jython', 'Lists']
```

```
>>> new_list=[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

ბიჯი არის 1. ვკითხულობთ სიის ყველა ელემენტს

```
>>> new_list[0:10:1]
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

ბიჯი არის 2

```
>>> new_list[0:10:2]
[2, 6, 10, 14, 18]
```

თუ პოზიციის ინდექსებს დავტოვებთ ცარიელს მაშინ გაჩუმებით დაყენდება

```
# საწყისი ინდექსი როგორც 0, და საბოლოო ინდექსი, როგორც სიაში შემავალი
# ელემენტების რაოდენობა.
>>> new_list[::2]
[2, 6, 10, 14, 18]
```

სიის მოდიფიცირება ხდება მსგავსი პრინციპით, ჩვენ ვიყენებთ ინდექსებს იმისთვის, რომ ჩავსვათ ან წავშალოთ ელემენტი სიიდან.

პროგრამული კოდი

```
# სიის ელემენტის მოდიფიცირება ამ შემთხვევაში ჩვენ ვცვლით სიის 10-ე ელემენტს.
>>> new_list[9] = 25
>>> new_list
[2, 4, 6, 8, 10, 12, 14, 16, 18, 25]
```

ჩვენ შეგვიძლია გამოვიყენოთ append - მეთოდი, სიის ბოლოში ელემენტის დასამატებლად. extend() მეთოდი გამოიყენება სიისათვის სხვა სიის ელემენტების დასამატებლად და insert() მეთოდი გამოიყენება სიაში ელემენტის ჩასასმელად.

del ოპერატორი გამოიყენება სიიდან ელემენტის წასაშლელად ან თვითონ სიის წასაშლელად.

pop() და remove() ფუნქციები გამოიყენება, იმისათვის, რომ წავშალოთ ელემენტი სიიდან.

pop() მეთოდი შლის ელემენტს სიის ბოლოდან და აბრუნებს მას. თუ pop ფუნქციას გადავცემთ არგუმენტად ინდექსს მაშინ სიიდან წაიშლება შესაბამისი ინდექსის ელემენტი.

remove() ფუნქცია შლის ელემენტს, მაგრამ არ აბრუნებს მას.

მაგალითი:

პროგრამული კოდი

```
# ვამატებთ სიის ბოლოში ახალ ელემენტს.
>>> new_list=['a','b','c','d','e','f','g']
>>> new_list.append('h')
>>> print new_list
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

# დავამატოთ სხვა სია უკვე არსებულ სიას.
>>> new_list2=['h','i','j','k','l','m','n','o','p']
>>> new_list.extend(new_list2)
>>> print new_list
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h','h', 'i', 'j',
'k', 'l', 'm', 'n', 'o', 'p']

# ჩავსვათ სიაში ახალი ელემენტი 'c', რიგით მესამე პოზიციაზე
#(ელემენტების ათვლა იწყება 0-იდან).
>>> new_list.insert(2,'c')
>>> print new_list
['a', 'b', 'c', 'c', 'd', 'e', 'f', 'g', 'h', 'h','i', 'j',
'k', 'l', 'm', 'n', 'o', 'p']

# ჩავსვათ სია სხვა სიაში
>>> another_list = ['a', 'b', 'c']
```

```
>>> another_list.insert(2, new_list)
>>> another_list
['a', 'b', [2, 4, 8, 10, 12, 14, 16, 18, 25], 'c']

# გადავაწეროთ ერთი სიის პირველ ორ ელემენტს მეორე სიის ელემენტები
>>> new_listA=[100,200,300,400]
>>> new_listB=[500,600,700,800]
>>> new_listA[0:2]=new_listB
>>> print new_listA
[500, 600, 700, 800, 300, 400]

# ერთი სიის მეორესთვის მინიჭების სხვა გზა
>>> one = ['a', 'b', 'c', 'd']
>>> two = ['e', 'f']
>>> one
['a', 'b', 'c', 'd']
>>> two
['e', 'f']
>>> one[4:4]
[]
>>> one[4:4] = two
>>> one
['a', 'b', 'c', 'd', 'e', 'f']
```

del ოპერატორის გამოყენება სიიდან ელემენტის წასაშლელად.

>>> new_list3=['a','b','c','d','e','f']

>>> del new_list3[2]

>>> new_list3

['a', 'b', 'd', 'e', 'f']

>>> del new_list3[1:3]

>>> new_list3

['a', 'e', 'f']

#del ოპერატორის გამოყენება სიის წასაშლელად

>>> new_list3=[1,2,3,4,5]

>>> print new_list3

[1, 2, 3, 4, 5]

>>> del new_list3

>>> print new_list3

Traceback (most recent call last):

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

NameError: name 'new_list3' is not defined

pop და remove ფუნქციების გამოყენება სიიდან ელემენტების წასაშლელად

>>> print new_list

['a', 'b', 'c', 'c', 'd', 'e', 'f', 'g', 'h', 'h', 'i',

'j', 'k', 'l', 'm', 'n', 'o', 'p']

```
>>> new_list.pop(2)
'c'
>>> print new_list
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h','h', 'i',
'j', 'k', 'l', 'm', 'n', 'o', 'p']

# წავშალოთ სიიდან პირველივე შემხვედრი ელემენტი 'h'.
>>> new_list.remove('h')
>>> print new_list
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
'l', 'm', 'n', 'o', 'p']

# pop ფუნქციის გამოყენების მაგალითი.
>>> x = 5
>>> times_list = [1,2,3,4,5]
>>> while times_list:
...     print x * times_list.pop(0)
...
5
10
15
20
25
```

პითონს გააჩნია ფუნქციები, რომლებიც მუშაობენ სიებზე.

ეს ფუნქციებია index, count, sort, reverse.

მაგალითი:

პროგრამული კოდი

```
# აბრუნებს ციფრი 4-ის ინდექსს
 >>> new_list=[1,2,3,4,5,6,7,8,9,10]
 >>> new_list.index(4)
 3


 # ინდექსით 4-ე ელემენტის შეცვლა
 >>> new_list[4] = 30
 >>> new_list
 [1, 2, 3, 4, 30, 6, 7, 8, 9, 10]


 >>> new_list[4] = 5
 >>> new_list
 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]


 # დავამატოთ სიას დუბლიკატი 6-იანი.
 # index ფუნქცია დააბრუნებს პირველი 6-იანის ინდექსს
 >>> new_list.append(6)
 >>> new_list
```

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 6]

>>> new_list.index(6)

5

ითვლის 2-იანების რაოდენობას სიაში

>>> new_list.count(2)

1

ითვლის 6-იანების რაოდენობას სიაში

>>> new_list.count(6)

2

დავალაგოთ ელემენტები სიაში.

>>> new_list.sort()

>>> new_list

[1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10]

შევაბრუნოთ მნიშვნელობების მიმდევრობა სიაში.

>>> new_list.reverse()

>>> new_list

[10, 9, 8, 7, 6, 6, 5, 4, 3, 2, 1]

3.2. სიებზე იტერაცია

სიებზე იტერაცია შეიძლება განხორციელდეს for ციკლის გამოყენებით.

მაგალითი 1:

პროგრამული კოდი

```
>>> ourList=[1,2,3,4,5,6,7,8,9,10]
>>> for elem in ourList:
...     print elem
...
1
2
3
4
5
6
7
8
9
10
```

მაგალითი 2:

ამ მაგალითში დავბეჭდავთ სიის პირველ 5 ელემენტს.

პროგრამული კოდი

```
>>> for elem in ourList[:5]:
...     print elem
...
1
2
3
4
5
```

3.3 List Comprehensions

განვიხილოთ შემდეგი მაგალითი:

პროგრამული კოდი

```
# მოცემული გამოსახულება სიაში შემავალ ობიექტებს
 # ამრავლებს 2-ზე და აბრუნებს შედეგად მიღებულ სიას,
 # რომელიც შეგვიძლია მივანიჭოთ რაიმე ცვლადს.
 >>> num_list = [1, 2, 3, 4]
 >>> [num * 2 for num in num_list]
 [2, 4, 6, 8]

 >>> num_list2 = [num * 2 for num in num_list]
 >>> num_list2
```

[2, 4, 6, 8]

მაგალითი2:

პროგრამული კოდი

```
# მოცემული მაგალითი აბრუნებს სიიდან იმ ელემენტებს, რომელთა მნიშვნელობაც 4-ზე
 # მეტია.
 >>> nums = [2, 4, 6, 8]
 >>> [num for num in nums if num > 4]
 [6, 8]
```

პროგრამული კოდი

```
# ვქმნით ages სიას და ვზრდით მის ელემენტებს ერთით
 >>> ages=[20,25,28,30]
 >>> [age+1 for age in ages]
 [21, 26, 29, 31]

 # ვქმნით სახელების სიას და მათ პირველ სიმბოლოს გარდავქმნით ზედა რეგისტრში.
 >>> names=['jim','frank','vic','leo','josh']
 >>> [name.title() for name in names]
 ['Jim', 'Frank', 'Vic', 'Leo', 'Josh']
```

```
# ვქმნით რიცხვების სიას და ვაბრუნებთ მათ აყვანილს 2-ე ხარისხში,
# თუ ეს რიცხვები ლუწია.
>>> numList=[1,2,3,4,5,6,7,8,9,10,11,12]
>>> [num*num for num in numList if num % 2 == 0]
[4, 16, 36, 64, 100, 144]

# range ფუნქციის გამოყენების მაგალითი
>>> [x*5 for x in range(1,20)]
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95]

# ჩადგმული ციკლების გამოყენების მაგალითი.
>>> list1 = [5, 10, 15]
>>> list2 = [2, 4, 6]
>>> [e1 + e2 for e1 in list1 for e2 in list2]
[7, 9, 11, 12, 14, 16, 17, 19, 21]
```

4. Tuples

tuple არის იგივე, რაც სია, იმ განსხვავებით, რომ მისი წაკითხვა შეიძლება, მაგრამ მოდიფიცირება არა.

მაგალითი:

პროგრამული კოდი

```
# ვქმნით ცარიელ tuple-ს
 >>> myTuple = ()

 # tuple-ის შექმნა და მათი გამოყენება
 >>> myTuple2 = (1, 'two',3, 'four')
 >>> myTuple2
 (1, 'two', 3, 'four')

 # ვქმნით ერთ-ელემენტიან tuple-ს
 >>> myteam = 'Bears',
 >>> myteam
 ('Bears',)
```

tuple, როგორც წესი გამოიყენება მონაცემების გადასაცემად ფუნქციისათვის,

მეთოდებისთვის, კლასისთვის და ა.შ.

5. ლექსიკონები

პითონში ლექსიკონი არის გასაღები-მნიშვნელობა ტიპის კონტეინერი.

ლექსიკონის თითოეული ჩანაწერი შედგება გასაღებისა და შესაბამისი მნიშვნელობისაგან

მაგალითი:

პროგრამული კოდი

```
# ვქმნით ცარიელ ლექსიკონს
 >>> myDict={}
 >>> myDict.values()
 []


 # გასაღები მნიშვნელობა წყვილის განსაზღვრა მოცემული ლექსიკონისათვის
 >>> myDict['one'] = 'first'
 >>> myDict['two'] = 'second'
 >>> myDict


{'two': 'second', 'one': 'first'}
```

ლექსიკონებთან სამუშაოდ განსაზღვრულია ფუნქციები და მეთოდები.

მაგალითი:

პროგრამული კოდი

```
# ვქმნით ცარიელ ლექსიკონს
 >>> mydict = {}


 # ვცდილობთ ვიპოვოთ გასაღები მოცემულ ლექსიკონში
 >>> 'firstkey' in mydict
 False
```

```
# გასაღები/მნიშვნელობა წყვილის დამატება ლექსიკონისათვის

>>> mydict['firstkey'] = 'firstval'


>>> 'firstkey' in mydict

True


# მნიშვნელობების სიმრავლე mydict ლექსიკონისათვის

>>> mydict.values()

['firstval']


# გასაღებების სიმრავლე mydict ლექსიკონისათვის

>>> mydict.keys()

['firstkey']


# ელემენტების რაოდენობის განსაზღვრა ლექსიკონში

>>> len(mydict)

1


# ლექსიკონში შემავალი ელემენტების დაბეჭდვა

>>> mydict

{'firstkey': 'firstval'}


# მოცემული mydict ლექსიკონი ჩავანაცვლოთ ახალი ლექსიკონით
```

```
>>> myDict=
{'r_wing':'Josh','l_wing':'Frank','center':'Jim',
'l_defense':'Leo','r_defense':'Vic'}

>>> myDict.values()
['Josh', 'Vic', 'Jim', 'Frank', 'Leo']

>>> myDict.get('r_wing')
'Josh'

>>> myDict['r_wing']
'Josh'


# მცდელობა ლექსიკონიდან ამოვიღოთ მნიშვნელობა გასაღების მიხედვით,
# რომელიც არ არსებობს
>>> myDict['goalie']
Traceback (most recent call last):

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

KeyError: 'goalie'

# მცდელობა ლექსიკონიდან ამოვიღოთ მნიშვნელობა გასაღების მიხედვით,
```

```
# რომელიც არ არსებობს get მეთოდის გამოყენებით.
>>> myDict.get('goalie')


# ახლა შევარჩიოთ გაჩუმებითი შეტყობინება, რომელიც დაიბეჭდება გასაღების
# არ არსებობის შემთხვევაში.
>>> myDict.get('goalie','Invalid Position')
'Invalid Position'

# იტერაცია ლექსიკონის ელემენტებზე
>>> for player in myDict.iterItems():
...     print player
...
('r_wing', 'Josh')

('r_defense', 'Vic')

('center', 'Jim')

('l_wing', 'Frank')

('l_defense', 'Leo')
```

```
# იტერაცია ლექსიკონის ელემენტებზე. მეორე გზა
>>> for key,value in myDict.iteritems():
...     print key, value
...
r_wing Josh
r_defense Vic
center Jim
l_wing Frank
l_defense Leo
```

6. სიმრავლეები (Sets)

სიმრავლეები შედგებიან უნიკალური, დაულაგებელი ელემენტებისაგან. სიმრავლეები არ შეიძლება შეიცავდნენ მუტირებად(ცვალებად) ელემენტებს, მაგრამ სიმრავლეები თვითონ არიან მუტირებადი.

მაგალითი:

პროგრამული კოდი

```
# იმისათვის რომ, გამოვიყენოთ სიმრავლე, საჭიროა მისი იმპორტი.
>>> from sets import Set

# სიმრავლეების შესაქმნელად ვიყენებთ შემდეგ სინტაქსს.
>>> myset = Set([1,2,3,4,5])
>>> myset
```

Set([5, 3, 2, 1, 4])

სიმრავლისთვის ელემენტის დამატება

>>> myset.add(6)

>>> myset

Set([6, 5, 3, 2, 1, 4])

დუბლიკატის დამატების მცდელობა

>>> myset.add(4)

>>> myset

Set([6, 5, 3, 2, 1, 4])

არსებობს სიმრავლის ორი ტიპი სახელობითი და გაყინული სიმრავლე. პირველი ტიპის

სიმრავლე არის მუტირებადი, ხოლო მეორე არამუტირებადი. სიმრავლეებზე განსაზღვრულია

მეთოდები და ფუნქციები. უმრავლესობა ამ ფუნქციებისა მოქმედებს ორივე ტიპის

სიმრავლეზე, მაგრამ არიან ფუნქციები, რომლებიც მოქმედებენ მხოლოდ მუტირებად

სიმრავლეებზე.

მაგალითი:

პროგრამული კოდი

შევქმნათ ორი სიმრავლე.

>>> s1 = Set(['jython','cpython','ironpython'])

>>> s2 = Set(['jython','ironpython','pypy'])

```
# სიმრავლის ასლის შექმნა
>>> s3 = s1.copy()
>>> s3
Set(['cpython', 'jython', 'ironpython'])

# აბრუნებს სიმრავლეს რომლის ელემენტებიც არის s1-ში მაგრამ არ არის s2-ში.
>>> s1.difference(s2)
Set(['cpython'])

# აბრუნებს s1 და s2 სიმრავლეების გაერთიანებას
>>> s1.union(s2)
Set(['cpython', 'pypy', 'jython', 'ironpython'])
```

მაგალითი 2:

პროგრამული კოდი

```
# ვქმნით 3 სიმრავლეს
>>> s1 = Set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>> s2 = Set([5, 10, 15, 20])
>>> s3 = Set([2, 4, 6, 8, 10])

# s2-დან ელემენტის წაშლა
>>> s2.pop()
```

```
20
>>> s2
Set([5, 15, 10])

# s1-დან ელემენტ 3-იანის წაშლა
>>> s1.discard(3)
>>> s1
Set([6, 5, 7, 8, 2, 9, 10, 1, 4])

# ვანახლებთ s1-ის მნიშვნელობას s1 და s2-ის თანაკვეთით.
>>> s1.intersection_update(s2)
>>> s1
Set([5, 10])
>>> s2
Set([5, 15, 10])

# s2-დან ყველა ელემენტის წაშლა
>>> s2.clear()
>>> s2
Set([])

# ანახლებს სიმრავლე s1-ს, რომ შეიცავდეს ყველა ელემენტს s3-დან.
>>> s1.update(s3)
>>> s1
```

Set([6, 5, 8, 2, 10, 4])

7. არეები(Ranges)

არეები ხშირად გამოიყენება იტერაციის დროს. არეების ფორმატს აქვს შემდეგ სახე.

პროგრამული კოდი

range([start], stop, [step])

ფუნქცია range-ის გამოყენების მაგალითი:

პროგრამული კოდი

მოცემული გვაქვს არე, რომელიც იწყება 0-დან და მთავრდება 9-ით

>>> range(0,10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> range(50, 65)

[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64]

>>> range(10)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

ბიჯის გამოყენების მაგალითი. ბიჯი არის 2.

>>> range(0, 10, 2)

[0, 2, 4, 6, 8]

ამ შემთხვევაში 100 მცირდება ბიჯით 10.

>>> range(100,0,-10)

[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]

for ციკლში range ფუნქციის გამოყენების მაგალითი.

პროგრამული კოდი

```
>>> for i in range(10):
...         print i
...
0
1
2
3
4
5
6
7
8
9
```

მაგალითი 2

```
>>> x = 1
>>> for i in range(2, 10, 2):
...         x = x + (i * x)
...         print x
...
3
15
105
945
```

range-ის გამოყენებით სიის შექმნა.

პროგრამული კოდი

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

8. ფაილები

ფაილ ობიექტები გამოიყენება, ფაილიდან მონაცემების წასაკითხად და ჩასაწერად.

open(filename[, mode]) - გამოიყენება ფაილის გასახსნელად(აბრუნებს ფაილ ობიექტს), თუ ფაილი არ არსებობს, მაშინ მოხდება მისი შექმნა.

filename - არის გასახსნელი ფაილის სახელი, ხოლო mode არის რეჟიმი რა რეჟიმშიც გვინდა ფაილის გახსნა.

ფაილთან მუშაობის რეჟიმები:

პროგრამული კოდი

r - კითხვა

w - ჩაწერა

a - დამატება

r+ - კითხვა-ჩაწერა

rb - ორობითი კითხვა

wb - ორობითი ჩაწერა

r+b - ორობითი კითხვა-ჩაწერა

მაგალითი:

პროგრამული კოდი

ფაილის გახსნა ჩაწერის რეჟიმში.

>>> f = open('newfile.txt','w')

არსებობს ფაილთან მუშაობის სხვადასხვა მეთოდები.

read([size]) - მეთოდი გამოიყენება მონაცემების ფაილიდან წასაკითხად, სადაც size არის წასაკითხი ბაიტების რაოდენობა, თუ მას არ მივუთითებთ მაშინ წაკითხულ იქნება მთლიანი ფაილი.

readline() - გამოიყენება ფაილიდან ერთი სტრიქონის წასაკითხად.

readlines([size]) - მეთოდი აბრუნებს ფაილის სტრიქონებისგან შემდგარ სიას, სადაც size არის წასაკითხი ბაიტების რაოდენობა.

write(string) მეთოდი გამოიყენება ფაილში სტრიქონის ჩასაწერად.

როცა მონაცემებს ვწერთ უნდა ვიცოდეთ პოზიცია ფაილში, საიდანაც მოხდება მონაცემების ჩაწერა.

tell() - მეთოდი გამოიყენება ფაილში მიმდინარე პოზიციის განსასაზღვრავად.

seek(offset, from) - მეთოდი გამოიყენება ფაილის მიმდინარე პოზიციის შესაცვლელად, სადაც offset არის ბაიტების რაოდენობა, რომელიც განსაზღვრავს, რამდენი ბაიტით უნდა მოხდეს წანაცვლება მოცემული from პოზიციიდან. თუ from პარამეტრის მნიშვნელობაა 0, მაშინ წანაცვლება მოხდება ფაილის დასაწყისიდან, თუ 1-ია, მაშინ მიმდინარე პოზიციიდან, ხოლო თუ ამ პარამეტრის მნიშვნელობაა 2, მაშინ წანაცვლება მოხდება ფაილის ბოლოდან.

პროგრამული კოდი

```
# ვქმნით ფაილს, ვწერთ მასში მონაცემებს, და შემდეგ ვკითხულობთ მას
 >>> f = open('newfile.txt','r+')
 >>> f.write('This is some new text for our file\n')
 >>> f.write('This should be another line in our file\n')

 #  რადგანაც ვიმყოფებით ფაილის ბოლოში, ამიტომ read მეთოდი დააბრუნებს
 #  ცარიელ სტრიქონს
 >>> f.read()
 ''
```

```
>>> f.readlines()
[]

>>> f.tell()
75L

# გადავიდეთ ფაილის დასაწყისში
>>> f.seek(0)
>>> f.read()
'This is some new text for our file\nThis should be another line in our file\n'

>>> f.seek(0)
>>> f.readlines()
['This is some new text for our file\n',
'This should be another line in our file\n']

# ვხურავთ ფაილს
>>> f.close()
```

9. Iterators

პითონის ყველა კონტეინერს გააჩნია იტერატორის მხარდაჭერა. თუ თქვენ შეეცდებით მოახდინოთ იტერაცია ისეთ ობიექტზე, რომელსაც იტერატორის მხარდაჭერა არ გააჩნია,

მიიღებთ შეცდომას. ობიექტზე იტერაციისათვის საჭიროა ამ ობიექტისათვის განვსაზღვროთ

სპეციალური __iter__ მეთოდი. თუ გვინდა, რომ მივიღოთ კონტეინერის იტერატორი,

მაშინ რაიმე ცვლადს უნდა მივანიჭოთ container.__iter__() და ეს ცვლადი გახდება ამ ობიექტის იტერატორი.

ამ ცვლადის ყოველი next()- გამოძახება აბრუნებს მომდევნო

ელემენტს და შლის მას საკუთარი სიიდან.

მაგალითი:

პროგრამული კოდი

```
>>> hockey_roster = ['Josh', 'Leo', 'Frank', 'Jim', 'Vic']
 >>> hockey_itr = hockey_roster.__iter__()
 >>> hockey_itr.next()
 'Josh'

 >>> for x in hockey_itr:
 ...     print x
 ...
Leo
Frank
Jim
Vic

# იტერატორის next მეთოდის გამოძახება, რომელსაც აღარ გააჩნია ელემენტი.
>>> hockey_itr.next()
```

Traceback (most recent call last):

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

StopIteration

მაგალითი 2:

პროგრამული კოდი

```
# სტრიქონზე და სიაზე იტერაცია.
>>> str_a = 'Hello'
>>> list_b = ['Hello','World']
>>> for x in str_a:
...     print x
...
H
e
l
l
o

>>> for y in list_b:
...     print y + '!'
...
```

Hello!

World!

10. მითითება და ასლები

ასლების შექმნა მუტირებადი და არამუტირებადი ობიექტებისთვის განსხვავდება ერთმანეთისგან. თუ ჩვენ გვინდა არამუტირებადი ობიექტის ასლის შექმნა, საჭიროა ის უბრალოდ მივანიჭოთ სხვა ცვლადს. მაგრამ მუტირებადი ობიექტებისთვის ეს ხერხი არ გამოდგება, რადგან ამ შემთხვევაში იქმნება მითითება ორიგინალ ობიექტზე და თუ ჩვენ ამ მითითების გამოყენებით შევცვლით შესაბამის ობიექტს, მაშინ შეიცვლება ორიგინალიც.

პროგრამული კოდი

```
# სტრიქონი არის არამუტირებადი, ამიტომ მისი მინიჭებისას სხვა ობიექტზე იქმნება
 # რეალური ასლი.
 >>> mystring = "I am a string, and I am an immutable object"
 >>> my_copy = mystring
 >>> my_copy
 'I am a string, and I am an immutable object'
 >>> mystring
 'I am a string, and I am an immutable object'
 >>> my_copy = "Changing the copy of mystring"
 >>> my_copy
```

'Changing the copy of mystring'

>>> mystring

'I am a string, and I am an immutable object'

სიები არიან მუტირებადი ობიექტები, ამიტომ მათი მინიჭება სხვა ცვლადზე,

ქმნის მითითებას ამ სიაზე. და სიის რომელიმე ცვლადის მოდიფიცირება

გამოიწვევს ორიგინალი სიის ელემენტების ცვლილებასაც.

>>> listA = [1,2,3,4,5,6]

>>> print listA

[1, 2, 3, 4, 5, 6]

>>> listB = listA

>>> print listB

[1, 2, 3, 4, 5, 6]

>>> del listB[2]

>>> print listA

[1, 2, 4, 5, 6]

თუ გვინდა სიის რეალური ასლის შექმნა,

ამისთვის უნდა გამოვიყენოთ copy მოდული.

>>> import copy

>>> a = [[]]

>>> b = copy.copy(a)

>>> b

```
[[]]


# b არ არის იგივე სია რაც a
>>> b is a
False



# მაგრამ სია b[0], იგივეა რაც სია a[0]
# ამას ქვია მეჩხერი ასლი
>>> b[0].append('test')
>>> a
[['test']]
>>> b
[['test']]
```

ღრმა ასლის(deep copy) შესაქმნელად საჭიროა მოვიქცეთ შემდეგნაირად:

პროგრამული კოდი

```
# მანიპულაციები integer ტიპის ცვლადებზე
 >>> a = 5
 >>> b = a
 >>> print b
```

```
5
>>> b = a * 5
>>> b
25
>>> a
5


# ვქმნით სიის ღრმა ასლს და ვახდენთ მის მოდიფიკაციას
>>> import copy
>>> listA = [[1,2],3,4,5,6]
>>> listB = copy.deepcopy(listA)
>>> print listB
[[1, 2], 3, 4, 5, 6]
>>> del listB[0][0]
>>> print listB
[[2], 4, 5, 6]
>>> print listA
[[1, 2], 3, 4, 5, 6]
```

ოპერატორები, გამოსახულებები და პროგრამის შესრულების მართვა

1. მათემატიკური ოპერაციები

პროგრამული კოდი

+	მიმატება
-	გამოკლება
*	გამრავლება
/	გაყოფა
//	Truncating Division
%	ნაშთი
**	ხარისხში აყვანის ოპერატორი
+var	უნარული მიმატება
-var	უნარული გამოკლება

მათემატიკური ოპერაციების გამოყენების მაგალითი

პროგრამული კოდი

```
# ძირითადი მათემატიკური გამოთვლები
>>> 10 - 6
4
>>> 9 * 7
63
```

Truncating Division მუშაობს შემდეგნაირად ხდება გაყოფის შესრულება და მიღებული შედეგი მრგვალდება ქვემოთ.

Truncating Division და ხარისხში აყვანის ოპერაციების მაგალითი.

პროგრამული კოდი

```
>>> 36 // 5
 7
 # ნაშთის ამოღების ოპერაცია
 >>> 36 % 5
 1
 # 5 ის მე-2 ხარისხი
 >>> 5**2
 25
 # 100 მე-2 ხარისხში
 >>> 100**2
 10000
```

გაყოფის ოპერაცია /. პითონში გაყოფის შესრულებისას ხდება მიღებული შედეგის ქვედა მიმართულებით დამრგვალება. ამის გამო პითონის დეველოპერებმა შემოიღეს ახალი გაყოფა,

რომელიც შედეგს არ ამრგვალებს. ახალი გაყოფა შედის __future__ მოდულში.

პროგრამული კოდი

```
# მუშაობს, როგორც საჭიროა
>>> 14/2
7
>>> 10/5
2
>>> 27/3
9
# ამ შემთხვევაში ჩვენ მოველით შედეგს: 1.5-ს
>>> 3/2
1
# აქ 1.4-ს
>>> 7/5
1
# აქ მოველით, რომ შედეგი იყოს 2.3333
>>> 14/6
2
```

ახლა ვნახოთ თუ როგორ მუშაობს ახალი გაყოფა.

პროგრამული კოდი

```
# ვახდენთ division-ის იმპორტს __future__-დან
from __future__ import division
# როგორც მოველოდით.
```

```
>>> 14/2
7.0
>>> 10/5
2.0
>>> 27/3
9.0
>>> 3/2
1.5
>>> 7/5
1.4
>>> 14/6
2.3333333333333335
```

უნარული მინუსის გამოყენების მაგალითი.

პროგრამული კოდი

```
>>> -10 + 5
 -5
>>> +5 - 5
0
>>> -(1 + 2)
-3
```

არსებობს სხვადასხვა მათემატიკური ფუნქციები.

პროგრამული კოდი

```
abs(var)         - აბსოლიტური მნიშვნელობა
 pow(x, y)        - x აყვანილი y ხარისხში
 pow(x,y,mod)     - (x**y) % mod
 round(var[, n])  - ამრგვალებს var ცვლადის მნიშვნელობას, n განსაზღვრავს
                    დამრგვალების სიზუსტეს.
 divmod(x, y)     - აბრუნებს გაყოფის შედეგად მიღებულ განაყოფსა და ნაშთს.
```

მაგალითი:

პროგრამული კოდი

```
# 9-ის აბსოლიტური მნიშვნელობა
 >>> abs(9)
 9
 # -9_ის აბსოლიტური მნიშვნელობა
 >>> abs(-9)
 9
 # შევასრულოთ გაყოფა და მივიღოთ, განაყოფი და ნაშთი.
 >>> divmod(8,4)
 (2, 0)
 >>> divmod(8,3)
 (2, 2)
```

ავიყვანოთ 8 მე-2 ხარისხსში.

>>> pow(8,2)

64

ავიყვანოთ 8 მე-2 ხარისხსში, მოდულით 3 ((8 **2) % 3)

>>> pow(8,2,3)

1

მოვახდინოთ დამრგვალება

>>> round(5.67,1)

5.7

>>> round(5.67)

6.00

2. შედარების ოპერატორები

შედარების ოპერატორებია

პროგრამული კოდი

> - მეტობა

< - ნაკლებობა

>= - მეტია ან ტოლია

<= - ნაკლებია ან ტოლია

!= - არ უდრის

== - უდრის

მაგალითი:

პროგრამული კოდი

```
>>> 8 > 10
False
>>> 256 < 725
True
>>> 10 == 10
True
# შედარების გამოყენება გამოსახულებაში
>>> x = 2*8
>>> y = 2
>>> while x != y:
...     print 'Doing some work...'
...     y = y + 2
...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
Doing some work...
```

შედარებების კომბინირება

>>> 3<2<3

False

>>> 3<4<8

True

3. ბიტური ოპერატორები

ბიტური ოპერატორები მუშაობენ ციფრებზე.(უფრო სწორად მათ ორობით წარმოდგენაზე)

პროგრამული კოდი

& - აბრუნებს 1-ს თუ ორივე ბიტი 1-ია.

| - აბრუნებს 0-ს თუ ორივე ბიტი 0-ია.

^ - აბრუნებს ერთს თუ ერთი რომელიღაც ბიტი არის 1 ხოლო მეორე 0

~ - ბიტი 1 გადაყავს 0-ში და 0 გადაყავს 1-ში.

მაგალითი:

პროგრამული კოდი

>>> 14 & 27

10

>>> 14 | 27

31

>>> 14 ^ 27

21

>>> ~14

-15

>>> ~27

-28

პროგრამული კოდი

14 & 27 = 00001110 and 00011011 = 00001010 (The integer 10)

14 | 27 = 00001110 or 000110011 = 00011111 (The integer 31)

14 ^ 27 = 00001110 xor 000110011 = 00010101 (The integer 21)

~14 = 00001110 = 11110001 (The integer -15)

წანაცვლების ოპერატორები >> და << ახდენენ ბიტების წანაცვლებას მარჯვნივ და მარცხნივ.

რაც ექვივალენტურია 2-ზე გაყოფის ან გამრავლების.

პროგრამული კოდი

წანაცვლება მარცხნივ, ამ შემთხვევაში 3*2

>>> 3<<1

6

ექვივალენტურია 3*2*2

>>> 3<<2

12

ექვივალენტურია 3*2*2*2*2*2

>>> 3<<5

96

წანაცვლება მარჯვნივ

ექვივალენტურია 3/2

>>> 3>>1

1

ექვივალენტურია 9/2

>>> 9>>1

4

ექვივალენტურია 10/2

>>> 10>>1

5

ექვივალენტურია 10/2/2

>>> 10>>2

2

4. სპეციალური მინიჭება

სპეციალური მინიჭება არის მინიჭებისგან შემდგარი გამოსახულების შემოკლებული ფორმა.

მგ. x = x + 5 შეიძლება ჩაიწეროს როგორც x += 5

მაგალითი:

პროგრამული კოდი

```
>>> x = 5
 >>> x
5
 # x = x + 1
 >>> x+=1
 >>> x
6
 # x = x*5
 >>> x*=5
 >>> x
30
```

5. boolean ტიპის გამოსახულებები

პითონში False არის მცდარი და True არის ჭეშმარიტი. 0-ითვლება მცდარად 0-ისაგან განსხვავებული მნიშვნელობა კი ჭეშმარიტად. გარდა ამისა არაცარიელი ობიექტი პითონში ითვლება ჭეშმარიტად.

მაგალითი:

პროგრამული კოდი

```
>>> mystr = ''
 >>> if mystr:
 ...     'Now I contain the following: %s' % (mystr)
 ... else:
 ...     'I do not contain anything'
 ...
'I do not contain anything'
 >>> mystr = 'Now I have a value'
 >>> if mystr:
 ...     'Now I contain the following: %s' % (mystr)
 ... else:
 ...     'I do not contain anything'
 ...
'Now I contain the following: Now I have a value'
```

პირობითი ოპერატორები:

პროგრამული კოდი

and - აბრუნებს True-ს თუ ორივე ოპერანდი არის ჭეშმარიტი

or - აბრუნებს True-ს თუ ერთ-ერთი ოპერანდი მაინც არის ჭეშმარიტი

not - აბრუნებს True-ს False-ის შემთხვევაში და პირიქით

პითონში, ისე, როგორც დაპროგრამების სხვა ენებში განისაზღვრება ოპერაციების

შესრულების პრიორიტეტი, მაგალითად 5*2+3 ამ გამოსახულებაში ჯერ შესრულდება გამრავლება, შემდეგ კი მიმატება.

მაგალითი:

პროგრამული კოდი

```
# გამოვაცხადოთ ცვლადები

>>> x = 10

>>> y = 12

>>> z = 14


# ჯერ შესრულდება y*z, ხოლო შემდეგ მიმატების ოპერაცია

>>> x + y * z

178


# ანალოგიურად

>>> x * y - z

106


# იგივეა რაც, x < y and y <= z and z > x

>>> x < y <= z > x

True


# ჯერ შესრულდება (2 * 0) და რადგანაც ის არის 0, ამიტომ მოხდება მისი დაბრუნება.

>>> 2 * 0 and 5 + 1
```

0

ჯერ გამოითვლება 2*1 და რადგანაც ის არის 2(True),

შემდეგ გამოითვლება 5 + 1 და მოხდება 6-იანის დაბრუნება.

>>> 2 * 1 and 5 + 1

6

დაბრუნდება x თუ ის არის ჭეშმარიტი წინააღმდეგ შემთხვევაში დაბრუნდება y

თუ ის არის მცდარი, თუ არადა დაბრუნდება z.

>>> x or (y and z)

10

ამ შემთხვევაში გამოითვლება და დაბრუნდება (7-2).

>>> 2 * 0 or ((6 + 8) and (7 - 2))

5

ამ შემთხვევაში ჯერ მოხდება ხარისხში აყვანის ოპერაციის შესრულება,

ხოლო შემდეგ მიმატების.

>>> 2 ** 2 + 8

12

6. ტიპების დაყვანა

პითონს გააჩნია ისეთი ტიპის ფუნქციები, რომელთა დანიშნულებაა ერთი ტიპის ობიექტის მეორე ტიპის ობიექტზე დაყვანა.

მაგალითი:

პროგრამული კოდი

```
# დავიყვანოთ integer ტიპი სიმბოლურ ტიპზე
>>> chr(4)
'\x04'
>>> chr(10)
'\n'

# დავიყვანოთ integer ტიპი float-ზე
>>> float(8)
8.0

# სიმბოლური ტიპის დაყვანა integer ტიპზე
>>> ord('A')
65
>>> ord('C')
67
>>> ord('z')
122

# დავიყვანოთ ნებისმიერი ობიექტი სტრიქონზე.
```

```
>>> repr(3.14)
'3.14'
>>> x = 40 * 5
>>> y = 2**8
>>> repr((x, y, ('one','two','three')))
"(200, 256, ('one', 'two', 'three'))"
```

eval - ფუნქციის დანიშნულებაა სტრიქონით წარმოდგენილი გამოსახულების გამოთვლა და შედეგის დაბრუნება.

მაგალითი:

პროგრამული კოდი

```
# დავუშვათ კლავიატურიდან ვკითხულობთ შემდეგ გამოსახულებას (x * y).
>>> x = 5
>>> y = 12
>>> keyboardInput = 'x * y'
>>> eval(keyboardInput)
60
```

7. პროგრამის შესრულების მართვა

7.1. if-elif-else ოპერატორი

if-elif-else ოპერატორი მუშაობს შემდეგნაირად, ჯერ გამოითვლება if ოპერატორის

შესაბამისი გამოსახულება, თუ მისი მნიშვნელობა მცდარია, მაშინ მართვა გადაეცემა

მომდევნო elif ოპერატორს და ა.შ. თუ ყველა გამოსახულება მცდარია მაშინ შესრულდება

else ოპერატორი. თუ გამოსახულება რომელიმე if ან elif-სთვის არის ჭეშმარიტი მაშინ შესრულდება შესაბამისი ოპერატორისათვის განკუთვნილი მოქმედება და მოხდება

if-elif-else ჯაჭვიდან გამოსვლა.

მაგალითი:

პროგრამული კოდი

```
pi =3.14
 x = 2.7 * 1.45
 if x == pi:
    print 'The number is pi'
 elif x > pi:
    print 'The number is greater than pi'
 else:
    print 'The number is less than pi'
```

ცარიელი სიები და სტრიქონები განისაზღვრებიან, როგორც False.

მაგალითი:

პროგრამული კოდი

```
>>> mylist = []
 >>> if mylist:
 ...     for person in mylist:
```

```
...         print person
... else:
...     print 'The list is empty'
...
The list is empty
```

7.2. while ციკლი

while ციკლი მუშაობს მანამდე, სანამ მასში მოთავსებული გამოსახულება არის ჭეშმარიტი.

პროგრამული კოდი

```
>>> x = 0
>>> y = 10
>>> while x <= y:
...     print 'The current value of x is: %d' % (x)
...     x += 1
... else:
...     print 'Processing Complete...'
...
The current value of x is: 0
The current value of x is: 1
The current value of x is: 2
```

```
The current value of x is: 3

The current value of x is: 4

The current value of x is: 5

The current value of x is: 6

The current value of x is: 7

The current value of x is: 8

The current value of x is: 9

The current value of x is: 10

Processing Complete...
```

7.3. continue ოპერატორი

continue ოპერატორი გამოიყენება ციკლის შიგნით. მისი შესრულებისას მართვა უბრუნდება ციკლს.

პროგრამული კოდი

```
# ბეჭდავს მხოლოდ ლუწ რიცხვებს

>>> x = 0

>>> while x < 10:

...     x += 1

...     if x % 2 != 0:

...         continue

...     print x

...
```

2

4

6

8

10

7.4. break ოპერატორი

break ოპერატორიც გამოიყენება ციკლის შიგნით და მისი შესრულებისას ხდება ციკლის მუშაობის შეწყვეტა და მართვის ციკლის შემდგომ ოპერატორზე გადაცემა.

პროგრამული კოდი

```
>>> x = 10
>>> while True:
...     if x == 0:
...         print 'x is now equal to zero!'
...         break
...     if x % 2 == 0:
...         print x
...     x -= 1
...
10
8
6
```

4

2

x is now equal to zero!

7.5. for ციკლი

for ციკლის გამოყენება შეიძლება ობიექტზე იტერაციისათვის.

ჯავაზე for ციკლით სიაზე იტერაცია, განისაზღვრება შემდეგნაირად.

პროგრამული კოდი

```
for(x = 0; x <= myList.size(); x++){
    System.out.println("The current index is: " + x);
}
```

იგივე კოდს პითონზე აქვს შემდეგი ფორმა.

პროგრამული კოდი

```
my_list = [1,2,3,4,5]
>>> for value in my_list:
...     print 'The current value is %s' % (value)
...
The current value is 1
The current value is 2
```

```
The current value is 3

The current value is 4

The current value is 5
```

თუ პარალელურად გვჭირდება ინდექსი მონაცემებზე იტერაციისას მაშინ გვექნება

პროგრამული კოდი

```
>>> myList = ['jython','java','python','jruby','groovy']
 >>> for index, value in enumerate(myList):
 ...     print index, value
 ...
0 jython
1 java
2 python
3 jruby
4 groovy
```

ლექსიკონში გასაღებებზე იტერაცია.

პროგრამული კოდი

```
>>> my_dict = {'Jython':'Java', 'CPython':'C',
 'IronPython':'.NET','PyPy':'Python'}
 >>> for key in my_dict:
 ...     print key
```

...

Jython

IronPython

CPython

PyPy

ფუნქციების გამოცხადება და მათი გამოყენება

1. ფუნქციების შექმნის სინტაქსი

ფუნქციები ჩვეულებრივ განისაზღვრებიან def სიტყვის გამოყენებით,

რომლის შემდეგაც მოდის ფუნქციის სახელი, პარამეტრების ჩამონათვალი და ფუნქციის ტანი.

პროგრამული კოდი

```
def times2(n): return n*2
```

ამ ფუნქციის სახელია times2, რომელსაც არგუმენტად გადაეცემა n.

ეს ფუნქცია უბრალოდ ამრავლებს n-ს 2-ზე და აბრუნებს მნიშვნელობას.

მოცემული ფუნქციის გამოძახებას აქვს ფორმა

პროგრამული კოდი

```
>>> times2(8)
 16
 >>> x = times2(5)
 >>> x
 10
```

1.2. def გასაღები სიტყვა

როგორც ვნახეთ def სიტყვით შეიძლება განვსაზღვროთ ფუნქცია.

ფუნქციის განსაზღვრება შეიძლება შეგვხვდეს პროგრამის ნებისმიერ დონეზე.

პითონში ფუნქცია არის შესასრულებელი მონაცემების ერთობლიობა.

დასაშვებია შემდეგი ოპერაციები.

პროგრამული კოდი

```
if variant:
    def f():
        print "One way"
else:
    def f():
        print "or another"
```

ფუნქციის შექმნის შემდეგ, შესაძლებელია მოცემული სახელის მქონე ახალი ფუნქციის

განსაზღვრა, რომელიც გადაფარავს წინა ფუნქციას. წინა ფუნქცია კი ავტომატურად წაიშლება.

მაგალითი:

პროგრამული კოდი

```
>>> def f(): print "Hello, world"
  ...
  >>> def f(): print "Hi, world"
  ...
  >>> f()
  Hi,world
```

ფუნქციის სახელი მიუთითებს ფუნქციას(ფუნქციის ობიექტს), ამიტომ შესაძლებელია ამ სახელის მინიჭება რაიმე ცვლადისათვის.

პროგრამული კოდი

```
>>> t2 = times2
  >>> t2(5)
  10
```

1.3. ფუნქციის პარამეტრები და ფუნქციის გამოძახება

ფუნქციის განსაზღვრისას ჩვენ ვუთითებთ პარამეტრებს, რომელსაც ეს ფუნქცია იყენებს.

პროგრამული კოდი

```
def tip_calc(amt, pct)
```

ფუნქციის პარამეტრები არ არის კონკრეტული ტიპის, რაც ნიშნავს იმას, რომ ფუნქციას შეიძლება გადავცეთ, ნებისმიერი ტიპის პარამეტრი.

მაგალითი:

პროგრამული კოდი

```
>>> times2(4)
 8


 #'abc'*2 = 'abcabc'
 >>> times2('abc')
 'abcabc'


 >>> times2([1,2,3])
 [1, 2, 3, 1, 2, 3]
```

ყველა ცვლადი პითონში არის მითითება ობიექტზე. როგორც ვიცით, არამუტირებადი ობიექტები არ იცვლებიან. სტრიქონი არის არამუტირებადი ობიექტი, ამიტომ თუ მას გადავცემთ ფუნქციას, რეალურად ის იმუშავებს ამ სტრიქონის ასლზე და არა ორიგინალ სტრიქონზე.

მაგალითი:

პროგრამული კოდი

```
>>> def changestr(mystr):
...     mystr = mystr + '_changed'
...     print 'The string inside the function: ', mystr
...     return

>>> mystr = 'hello'
>>> changestr(mystr)
The string inside the function:  hello_changed
>>> mystr
'hello'
```

ფუნქციები არიან ობიექტები ამიტომ შეიძლება მათი პარამეტრად გადაცემა სხვა ფუნქციისათვის.

მაგალითი:

პროგრამული კოდი

```
# განვსაზღვროთ ფუნქცია, რომელსაც არგუმენტებად გადაეცემა 2 მნიშვნელობა და
# 1 მათემატიკური ფუნქცია.
>>> def perform_calc(value1, value2, func):
...     return func(value1, value2)
...

# განვსაზღვროთ მათემატიკური ფუნქცია
```

```
>>> def mult_values(value1, value2):
...     return value1 * value2...
>>> perform_calc(2, 4, mult_values)
8

# განვსაზღვროთ სხვა მათემატიკური ფუნქცია
>>> def add_values(value1, value2):
...     return value1 + value2
...
>>> perform_calc(2, 4, add_values)
6
>>>
```

თუ ფუნქციას გადაეცემა ორი ან ორ არგუმენტზე მეტი მაგალითად draw_point(x, y), მაშინ უმჯობესია ფუნქციის შემდეგნაირი გამოძახება. draw_point(x = 5, y = 10) ფუნქციის პარამეტრებს შეიძლება გააჩნდეს გაჩუმებითი მნიშვნელობა.

მაგალითი

პროგრამული კოდი

```
def times_by(n, by=2):    return n * by
```

ფუნქციებს შეიძლება გადაეცეს ნებისმიერი რაოდენობა პარამეტრებისა.

მაგალითი

პროგრამული კოდი

```
def sum_args(*nums):
    return sum(nums)
```

```
>>> seq = [6,5,4,3]
>>> sum_args(*seq)
18
```

```
>>> sum_args(1,2,3,4)
10
```

2. ფუნქციის რეკურსიული გამოძახება

რეკურსია, არის მოვლენა, როდესაც ფუნქცია იძახებს თავის თავს.

მაგალითი ფაქტორიალის დათვლა:

პროგრამული კოდი

```
def fact(n):
    if n in (0, 1):
        return 1
    else:
        return n * fact(n - 1)
```

3. ფუნქციების დოკუმენტირება

ფუნქციის აღწერა, როგორც წესი განთავსდება ფუნქციის ტანის დასაწყისში.

მაგალითად

პროგრამული კოდი

```
def times2(n):
    """გადაეცემა n, აბრუნებს n * 2"""
    return n * 2
```

მრავალსტრიქონიანი ფუნქციის დოკუმენტირებისას, რეკომენდირებულია შემდეგი ფორმატის

გამოყენება

პროგრამული კოდი

```
def fact(n):
    """აბრუნებს n-ის ფაქტორიალს
      რეკურსიულად ითვლის n-ის ფაქტორიალს. არ ამოწმებს
      არგუმენტებს არაუარყოფითობაზე და არც სტეკის გადავსებას.
    """
    if n in (0, 1):
       return 1
    else:
       return n * fact(n - 1)
```

ამის შემდეგ შეგვიძლია გამოვიყენოთ help(fact), რომელიც დაგვიბრუნებს fact ფუნქციის აღწერას.

პროგრამული კოდი

```
>>> help(fact)
 Help on function fact in module __main__:

 fact(n)
     აბრუნებს n-ის ფაქტორიალს
     რეკურსიულად ითვლის n-ის ფაქტორიალს. არ ამოწმებს
     არგუმენტებს არაუარყოფითობაზე და არც სტეკის გადავსებას.
 >>>
```

4. მნიშვნელობების დაბრუნება

ყველა ფუნქცია აბრუნებს რაღაც მნიშვნელობას. ფუნქციებს შეუძლიათ დააბრუნონ ბევრი მნიშვნელობა tuple ან სხვა სტრუქტურის გამოყენებით.

პროგრამული კოდი

```
>>> def calc_tips(amount):
...     return (amount * .18), (amount * .20)
```

```
...
>>> calc_tips(25.25)
(4.545, 5.0500000000000001)
```

ფუნქციებს შეუძლიათ მართვა დააბრუნონ ნებისმიერ დროს და დააბრუნონ ნებისმიერი ობიექტი.

პროგრამული კოდი

```
>>> def check_pos_perform_calc(num1, num2, func):
...     if num1 > 0 and num2 > 0:
...         return func(num1, num2)
...     else:
...         return 'Only positive numbers can be used with this function!'
...
>>> def mult_values(value1, value2):
...     return value1 * value2
...
>>> check_pos_perform_calc(3, 4, mult_values)
12
>>> check_pos_perform_calc(3, -44, mult_values)
'Only positive numbers can be used with this function!'
```

თუ return ოპერატორს არ ვიყენებთ, მაშინ ფუნქცია აბრუნებს None მნიშვნელობას.

5.ცვლადები

ფუნქციაში განსაზღვრული ცვლადები ხედვადია, მხოლოდ ფუნქციის შიგნით.

პროგრამული კოდი

```
>>> def square_num(num):
...     """ ვაბრუნებთ რიცხვს, აყვანილს კვადრატში """
...     sq = num * num
...     return sq
...
>>> square_num(35)
1225
>>> sq
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError:  name 'sq' is not defined
```

ფუნქციის შიგნით გლობალურ ცვლადებზე წვდომა ხორციელდება შემდეგნაირად.

პროგრამული კოდი

```
>>> sq = 0
>>> def square_num(n):
...     global sq
...     sq = n * n
```

```
...     return sq
...
>>> square_num(10)
100
>>> sq
100
```

6. ფუნქციების განსაზღვრის ალტერნატიული გზა

def გასაღები სიტყვა არ არის ერთადერთი გზა ფუნქციების განსაზღვრისა.

მაგალითად

1. lambda ფუნქციები. lambda გასაღები სიტყვა ქმნის უსახელო ფუნქციას.

2. ბმული ფუნქციები. იმის მაგივრად, რომ მოვახდინოთ x.a() გამოძახება.

ჩვენ შეგვიძლია x.a მივანიჭოთ რაიმე სახელს და შემდეგ მოვახდინოთ ამ

სახელის გამოყენებით ფუნქციის გამოძახება.

6.1.lambda ფუნქციები

როგორც ვთქვით lambda ფუნქციები არიან ანონიმური ფუნქციები. ამ ტიპის ფუნქციების

შექმნა შეიძლება საჭირო გახდეს,

1.როცა გვინდა კომპაქტური კოდის შექმნა.

2.როცა არ არის საჭიროება სახელის მქონე ფუნქციის შექმნისა რადგან მას ვიყენებთ

ერთხელ.

lambda ფუნქცია განისაზღვრება შემდეგნაირად: lambda პარამეტრები: ფუნქციის ტანი.

პროგრამული კოდი

```
>>> name_combo = lambda first, last: first + ' ' + last
 >>> name_combo('Jim','Baker')
 'Jim Baker'
```

7. გენერატორი ფუნქციები

გენერატორი ფუნქციები შეიცავენ yield ოპერატორს, ყოველ ჯერზე, როდესაც yield ოპერატორი შესრულდება ფუნქცია აბრუნებს მნიშვნელობას და იმახსოვრებს შეჩერების

წერტილს, ფუნქციაზე მეორედ მიმართვისას ფუნქციის მუშაობა განახლდება შეჩერების

წერტილიდან და ა.შ. თუ ფუნქცია აღარ შეიცავს მეტ yield ოპერატორს, მაშინ გამოისროლება StopIteration Exception-ი.

7.1. გენერატორის განსაზღვრა

გენერატორი განისაზღვრება შემდეგნაირად

პროგრამული კოდი

```
def g():
    print "before yield point 1"
    # გენერატორი აბრუნებს მნიშვნელობას, როდესაც შეხვდება yield ოპერატორი
    yield 1
    print "after 1, before 2"
    yield 2
    yield 3
```

გენერატორის გამოყენების მაგალითი

პროგრამული კოდი

```
# ვქმნით გენერატორს
  >>> x = g()
  # ვიძახებთ next(), რომ მივიღოთ მნიშვნელობა yield-ის გამოყენებით
  >>> x.next()
  before the yield point 1
  1
  >>> x.next()
  after 1, before 2
  2
  >>> x.next()
  3
  >>> x.next()
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  StopIteration
```

გენერატორი მუშაობს შემდეგნაირად ყოველი x.next()-ის გამოძახებისას, ბრუნდება მომდევნო

მნიშვნელობა, yield ოპერატორის გამოყენებით g-ფუნქციაში.

ახლა განვიხილოთ უფრო უკეთესი მაგალითი გენერატორების გამოყენებისა.

პროგრამული კოდი

```
>>> def step_to(factor, stop):
...     step = factor
...     start = 0
...     while start <= stop:
...         yield start
...         start += step
...
>>> for x in step_to(1, 10):
...     print x
...
0
1
2
3
4
5
6
7
8
9
10
```

```
>>> for x in step_to(2, 10):
...     print x
...
0
2
4
6
8
10
>>>
```

ახლა დავამატოთ ამ ფუნქციას შემოწმება, რომ თუ factor >= stop-ზე, მაშინ დაუბრუნოს მართვა გამომძახებელს.

პროგრამული კოდი

```
>>> def step_return(factor, stop):
...     step = factor
...     start = 0
...     if factor >= stop:
...         return
...     while start <= stop:
...         yield start
...         start += step
...
>>> for x in step_return(1,10):
```

```
...     print x
...
0
1
2
3
4
5
6
7
8
9
10
>>> for x in step_return(3,10):
...     print x
...
0
3
6
9
>>> for x in step_return(3,3):
...     print x
...
>>>
```

გენერატორი არ უნდა აბრუნებდეს რაიმე მნიშვნელობას return-ის გამოყენებით.

პროგრამული კოდი

```
def g():
    yield 1
    yield 2
    return None


 for i in g():
    print i


 SyntaxError: 'return' with argument inside generator
```

7.2. გენერატორის შექმნა გამოსახულების გამოყენებით

გენერატორის ობიექტის შექმნა შეიძლება გამოსახულებითაც.

პროგრამული კოდი

```
>>> x = (2 * x for x in [1,2,3,4])
 >>> x
 <generator object at 0x1>
 >>> x()
 Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'generator' object is not callable


# ვნახოთ როგორ მუშაობს, ჩვენს მიერ შექმნილი გენერატორი
>>> for v in x:
...     print v
...
2
4
6
8
>>>
```

8. სახელების სივრცეები, ხედვის არე.

როგორც ვიცით პითონში მოდული აერთიანებს კლასებს, ფუნქციებს და ა.შ.

სადაც მოდული არის ფაილი, რომელშიც ჩაწერილია ეს მონაცემები.

მონაცემების იმპორტი ჩვენ შეგვიძლია განვახორციელოთ ფუნქციის შიგნითაც.

პროგრამული კოდი

```
def f():
    from NS import A, B
```

პითონში ყველა სახის მონაცემი მათ შორის ფუნქციებიც წარმოადგენენ ობიექტებს.

ამიტომ შეგვიძლია ერთ ფუნქციას არგუმენტად გადავცეთ მეორე ფუნქცია. ჩვენ შეგვიძლია აგრეთვე ერთი ფუნქციის შიგნით განვსაზღვროთ მეორე შვილი ფუნქცია.

პროგრამული კოდი

```
>>> def parent_function():
...     x = [0]
...     def child_function():
...         x[0] += 1
...         return x[0]
...     return child_function
...
>>> p = parent_function()
>>> p()
1
>>> p()
2
>>> p()
3
>>> p()
4
```

როგორც კოდიდან ჩანს მშობელი ფუნქცია აბრუნებს შვილ ფუნქციას, რომლის ყოველი გამოძახებისას იცვლება მშობელი ფუნქციის ცვლადი x-ი.

9. ფუნქციების დეკორატორი.

ფუნქციების დეკორატორის დანიშნულებაა, იმ ფუნქციის შესაძლებლობის გაფართოება, რომელსაც უკეთებს დეკორაციას.

მაგალითად

პროგრამული კოდი

```
def plus_five(func):
    x = func()
    return x + 5

@plus_five
def add_nums():
    return 1 + 2
```

ამ მაგალითში ფუნქცია add_nums-ს, დეკორაციას უკეთებს ფუნქცია plus_five.

ეს იგივეა, რომ ფუნქცია plus_five-ს არგუმენტად გადავცეთ ფუნქცია add_nums.

იგივეა რაც შემდეგი კოდი

პროგრამული კოდი

```
>>> add_nums = plus_five(add_nums)
 >>> add_nums
 ... 8
```

ამ კოდის შესრულების შემდეგ add_nums უკვე არის მთელი რიცხვი და არა ფუნქცია.

ახლა შევიტანოთ კოდში შემდეგი ცვლილებები

პროგრამული კოდი

```
def plus_five(func):
    def inner(*args, **kwargs):
        x = func(*args, **kwargs) + 5
        return x
    return inner

 @plus_five
 def add_nums(num1, num2):
    return num1 + num2
```

ამ შემთხვევაში add_nums ფუნქციის გამოძახებისას, რეალურად გამოიძახება inner ფუნქცია.

და გვექნება:

პროგრამული კოდი

```
>>> add_nums(2,3)
```

```
10
>>> add_nums(2,6)
13
```

ახლა განვიხილოთ რამე უფრო გამოსადეგი მაგალითი.

პროგრამული კოდი

```
def sales_tax(func):
    ''' Applies a sales tax to a given bill calculator '''
    def calc_tax(*args, **kwargs):
        f = func(*args, **kwargs)
        tax = f * .18
        print "Total before tax: $ %.2f" % (f)
        print "Tax Amount: $ %.2f" % (tax)
        print "Total bill: $ %.2f" % (f + tax)
    return calc_tax

@sales_tax
def calc_bill(amounts):
    ''' Takes a sequence of amounts and returns sum '''
    return sum(amounts)
```

როგორც მაგალითიდან ჩანს calc_bill ფუნქცია უბრალოდ ითვლის გადასახადების ჯამს. ამ ფუნქციას დეკორაციას უკეთებს sales_tax ფუნქცია, რომლის calc_tax ფუნქცია ითვლის დამატებით გადასახადს და ბეჭდავს მას, და საერთო გადასახადს.

პროგრამული კოდი

```
>>> amounts = [12.95,14.57,9.96]
 >>> calc_bill(amounts)
 Total before tax: $ 37.48
 Tax Amount: $ 6.75
 Total bill: $ 44.23
```

*args და **kvargs პარამეტრები გამოიყენება იმის გამო, რომ ჩვენ წინასწარ არ ვიცით რა და რამდენი არგუმენტი გადაეცემა დეკორირებულ ფუნქციას. *args ნიშნავს, რომ ფუნქციას შეიძლება გადაეცეს ნებისმიერი რაოდენობა არგუმენტებისა.

დეკორაციის დროს ჩვენ შეგვიძლია პარამეტრები გადავცეთ დეკორატორ ფუნქციებს. ამისათვის
საჭიროა კიდევ ერთი ჩადგმული ფუნქცია დეკორატორი ფუნქციის შიგნით.

პროგრამული კოდი

```
def tip_amount(tip_pct):
    def calc_tip_wrapper(func):
        def calc_tip_impl(*args, **kwargs):
            f = func(*args, **kwargs)
            print "Total bill before tip: $ %.2f" % (f)
```

```
            print "Tip amount: $ %.2f" % (f * tip_pct)
            print "Total with tip: $ %.2f" % (f + (f * tip_pct))
        return calc_tip_impl
    return calc_tip_wrapper
```

ახლა ვნახოთ როგორ შეიძლება დეკორაციის განხორციელება

პროგრამული კოდი

```
>>> @tip_amount(.18)
... def calc_bill(amounts):
...     ''' Takes a sequence of amounts and returns sum '''
...     return sum(amounts)
...
>>> amounts = [20.95, 3.25, 10.75]
>>> calc_bill(amounts)
Total bill before tip: $ 34.95
Tip amount: $ 6.29
Total with tip: $ 41.24
```

როგორც ვხედავთ შედეგი იგივეა, როგორიც იყო მანამდე იმ განსხვავებით, რომ ახლა ჩვენ შეგვიძლია ვცვალოთ დამატებითი გადასახადის პროცენტული მნიშვნელობა.

განხილული დეკორაცია იგივეა, რაც შემდეგი კოდის შესრულება

პროგრამული კოდი

```
calc_bill = tip_amount(.18)(calc_bill)
```

იმპორტის დროს სრულდება tip_amount() ფუნქცია, რომელსაც არგუმენტებად გადაეცემა პროცენტული მნიშვნელობა და ფუნქცია calc_bill. შედეგად ვღებულობთ ახალ calc_bill

ფუნქციას, რომლის გამოძახებაც გვაძლევს საჭირო შედეგს.

10.1. Coroutines

Coroutine-ში გამოიყენება yield ოპერატორი, ისევე როგორც გენერატორებში, მაგრამ ამ შემთხვევაში ხდება არა მონაცემების დაბრუნება, არამედ წაკითხვა.

განვიხილოთ მაგალითი

პროგრამული კოდი

```
def co_example(name):
    print 'Entering coroutine %s' % (name)
    my_text = []
    while True:
       txt = (yield)
       my_text.append(txt)
       print my_text
```

Coroutine-ის შექმნა ხდება შემდეგნაირად

პროგრამული კოდი

```
>>> ex = co_example("example1")
 >>> ex.next()
 Entering coroutine example1
```

next() მეთოდის გამოძახება აუცილებელია coroutine-ის ინიციალიზაციისათვის.

ახლა ვნახოთ როგორ შეიძლება გავუგზავნოთ მონაცემები coroutine-ს.

პროგრამული კოდი

```
>>> ex.send("test1")
 ['test1']
 >>> ex.send("test2")
 ['test1', 'test2']
 >>> ex.send("test3")
 ['test1', 'test2', 'test3']
```

როგორც ჩანს ჩვენ ვიყენებთ send()_მეთოდს მონაცემების გასაგზავნად.

თუ coroutine აღარ გვჭირდება, მაშინ ის უნდა დავხუროთ close()-მეთოდის გამოყენებით

პროგრამული კოდი

```
>>> ex.close()
 >>> ex.send("test1")
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
```

StopIteration

10.2. დეკორატორი coroutine-სათვის

coroutine-ის ინიციალიზაციისათვის, როგორც ვიცით საჭიროა next()-მეთოდის გამოძახება, ამ ზედმეტი ოპერაციის თავიდან ასაცილებლად საჭიროა, დეკორატორის გამოყენება.

პროგრამული კოდი

```
def coroutine_next(f):
    def initialize(*args,**kwargs):
        coroutine = f(*args,**kwargs)
        coroutine.next()
        return coroutine
    return initialize
```

ახლა, ჩვენ ზემოთ განხილულ ფუნქციას გავაფორმებთ დეკორატორით.

პროგრამული კოდი

```
>>> @coroutine_next
... def co_example(name):
...     print 'Entering coroutine %s' % (name)
...     my_text = []
...     while True:
...         txt = (yield)
```

```
...         my_text.append(txt)
...         print my_text
...
>>> ex2 = co_example("example2")
Entering coroutine example2
>>> ex2.send("one")
['one']
>>> ex2.send("two")
['one', 'two']
>>> ex2.close()
```

10.3. coroutine მაგალითი

ახლა ჩვენ ვნახოთ, თუ როგორ შეიძლება იქნას გამოყენებული coroutine რეალურ სიტუაციებში.

ქვემოთ ჩვენ განვსაზღვრავთ ფუნქციას რომელიც ეძებს იმ ტექსტს ფაილში, რომელიც

ეგზავნება მას send()-მეთოდით და აბრუნებს სიტყვის, ტექსტში დამთხვევების რაოდენობას.

პროგრამული კოდი

```
def search_file(filename):
    print 'Searching file %s' % (filename)
    my_file = open(filename, 'r')
    file_content = my_file.read()
    my_file.close()
    while True:
```

```
        search_text = (yield)
        search_result = file_content.count(search_text)
        print 'Number of matches: %d' % (search_result)
```

search_file ფუნქციის ილუსტრაცია

პროგრამული კოდი

```
>>> search = search_file("example4_3.txt")
 >>> search.next()
 Searching file example4_3.txt
 >>> search.send('python')
 Number of matches: 0
 >>> search.send('Jython')
 Number of matches: 1
 >>> search.send('the')
 Number of matches: 4
 >>> search.send('This')
 Number of matches: 2
 >>> search.close();
```

მონაცემების კითხვა და ჩაწერა

როგორც წესი, პროგრამა კითხულობს მონაცემებს კლავიატურიდან და გამოაქვს შედეგები ეკრანზე,

ან წერს მას ფაილში. ამ თავში ჩვენ განვიხილავთ მონაცემების კითხვას ტერმინალიდან, ფაილიდან კითხვასა და ფაილში ჩაწერას. აგრეთვე განვიხილავთ picle მოდულის გამოყენებით ფაილში ობიექტების შენახვასა და კითხვას.

1. ტერმინალიდან მონაცემების წაკითხვა.

ამ ამოცანის შესასრულებლად გამოიყენება sys.stdin და raw_input მეთოდები.

პირველი გზის გამოყენებით მონაცემების წაკითხვა ხდება შემდეგნაირად.

პროგრამული კოდი

```
# ვკითხულობთ ტექსტს ტერმინალიდან და ვანიჭებთ მას რაიმე ცვლადს.
  >>> import sys
  >>> fav_team = sys.stdin.readline()
  Cubs
  >>> sys.stdout.write("My favorite team is: %s" % fav_team)
  My favorite team is: Cubs
```

მეორე გზით მონაცემების წაკითხვა

პროგრამული კოდი

```
# ვკითხულობთ ტექსტს ტერმინალიდან და ვანიჭებთ მას რაიმე ცვლადს.
  >>> fav_team = raw_input("Enter your favorite team: ")
  Enter your favorite team: Cubs
```

2. გარემომცველ ცვლადებთან მუშაობა.(Environment variables).

გარემომცველი ცვლადების მნიშვნელობის წაკითხვისა და ცვლილებისათვის, საჭიროა os მოდულის გამოყენება, რომელსაც გააჩნია ლექსიკონი environ.

პროგრამული კოდი

```
>>> import os
 >>> os.environ["HOME"]
 '/Users/juneau'


 # home დირექტორიის ცვლილება
 >>> os.environ["HOME"] = "/newhome"
 >>> os.environ["HOME"]
 '/newhome'
```

პროგრამისთვის გადაცემული პარამეტრების წასაკითხად გამოიყენება sys მოდული.

განვიხილოთ მაგალითი

პროგრამული კოდი

```
# sysargv_print.py - command line-იდან გადაცემული მნიშვნელობების წაკითხვა.
 import sys
 for sysargs in sys.argv:
     print sysargs
```

```
# გამოყენება
>>> jython sysargv_print.py test test2 "test three"
sysargv_print.py
test
test2
test three
```

როგორც ვხედავთ sys.argv პარამეტრის პირველი მნიშვნელობაა სკრიპტის სახელი.

3. ფაილიდან კითხვა და ჩაწერა.

როგორც მე-2 თავში აღვნიშნეთ ფაილიდან წასაკითხად, ჯერ უნდა შეიქმნას file ობიექტი. რაც ხდება open - ფუნქციის საშუალებით. ფაილთან მუშაობის დამთავრების შემდეგ უნდა მოხდეს ფაილის დახურვა close - მეთოდით, write - მეთოდი გამოიყენება ფაილში ჩასაწერად.

მაგალითი:

პროგრამული კოდი

```
>>> my_file = open('mynewfile.txt','w')
>>> first_string = "This is the first line of text."
>>> my_file.write(first_string)
>>> my_file.close()
```

ახლა განვიხილოთ ფაილთან მუშაობის სხვადასხვა მეთოდები, მაგალითების საშუალებით

პროგრამული კოდი

მონაცემების ფაილში ჩაწერა

```
>>> my_file = open('mynewfile.txt','w')
>>> my_file.write('This is the first line of text.\n')
>>> my_file.write('This is the second line of text.\n')
>>> my_file.write('This is the last line of text.\n')
>>> my_file.flush()  # არა-აუცილებელი, წერს მონაცემებს ფაილში დროზე ადრე.
>>> my_file.close()
```

```
# ფაილის გახსნა კითხვის რეჟიმში
>>> my_file = open('mynewfile.txt','r')
>>> my_file.read()
'This is the first line of text.\nThis is the second line of text.\nThis is the last line of text.\n'
```

```
# თუ კიდევ წავიკითხავთ მივიღებთ '' სტრიქონს, რადგანაც უკვე გავედით ფაილის ბოლოში.
>>> my_file.read()
''
```

```
# გავიდეთ ფაილის დასაწყისში
>>> my_file.seek(0)
>>> my_file.read()
```

'This is the first line of text.This is the second line of text.This is the last line of text.'

გავიდეთ ფაილის დასაწყისში და გამოვიყენოთ readline - მეთოდი

>>> my_file.seek(0)

>>> my_file.readline()

'This is the first line of text.\n'

>>> my_file.readline()

'This is the second line of text.\n'

>>> my_file.readline()

'This is the last line of text.\n'

>>> my_file.readline()

''

tell() მეთოდის გამოყენება ფაილში, კურსორის(ადგილმდებარეობის)

მიმდინარე პოზიციის განსასაზღვრავად

>>> my_file.tell()

93L

>>> my_file.seek(0)

>>> my_file.tell()

0L

ციკლი რომელიც კითხულობს ფაილს სტრიქონ-სტრიქონ.

>>> for line in my_file:

...   print line

...

This is the first line of text.

This is the second line of text.

This is the last line of text.

file - ობიექტს გააჩნია ატრიბუტები, რომლებიც იძლევიან ინფორმაციას file ობიექტის შესახებ.

პროგრამული კოდი

closed - გააჩნია მნიშვნელობა True თუ ფაილი დახურულია, False წინააღმდეგ შემთხვევაში

mode - აბრუნებს რეჟიმს რა რეჟიმშიც არის ფაილი გახსნილი

name - აბრუნებს ფაილის სახელს

მაგალითი

პროგრამული კოდი

```
>>> my_file.closed
False
>>> my_file.mode
'r'
>>> my_file.name
```

'mynewfile.txt'

4. pickle მოდული

pickle მოდულის დანიშნულებაა პითონის ობიექტის შენახვა დისკზე ფაილის სახით, და შემდეგ მისი წაკითხვა ფაილიდან ობიექტის სახით.

ობიექტის დისკზე ჩასაწერად საჭიროა pickle მოდულის გამოყენება. ქვემოთ განხილულ მაგალითში ჩვენ ვქმნით player-ობიექტს და ვინახავთ მას ფაილში. შემდეგ ვკითხულობთ player ობიექტს და ვიყენებთ პროგრამაში.

პროგრამული კოდი

```
>>> import pickle
>>> class Player(object):
...     def __init__(self, first, last, position):
...         self.first = first
...         self.last = last
...         self.position = position
...
>>> player = Player('Josh','Juneau','Forward')
>>> pickle_file = open('myPlayer','wb')
>>> pickle.dump(player, pickle_file)
>>> pickle_file.close()
```

როგორც მაგალითიდან ჩანს ჩვენ შევინახეთ ობიექტი ფაილში pickle.dump()-ფუნქციის გამოყენებით. ახლა წავიკითხოთ ობიექტი ამ ფაილიდან picle.load()-ფუნქციის მეშვეობით

პროგრამული კოდი

```
>>> pickle_file = open('myPlayer','rb')
 >>> player1 = pickle.load(pickle_file)
 >>> pickle_file.close()
 >>> player1.first
 'Josh'
 >>> player1.last, player1.position
 ('Juneau', 'Forward')
```

ობიექტების ფაილში შესანახად ჩვენ შეგვიძლია გამოვიყენოთ shelve მოდული.

პროგრამული კოდი

```
# სხვადასხვა player ობიექტების შენახვა
 >>> import shelve
 >>> player1 = Player('Josh','Juneau','forward')
 >>> player2 = Player('Jim','Baker','defense')
 >>> player3 = Player('Frank','Wierzbicki','forward')
 >>> player4 = Player('Leo','Soto','defense')
 >>> player5 = Player('Vic','Ng','center')
```

```
>>> data = shelve.open("players")
>>> data['player1'] = player1
>>> data['player2'] = player2
>>> data['player3'] = player3
>>> data['player4'] = player4
>>> data['player5'] = player5
>>> player_temp = data['player3']
>>> player_temp.first, player_temp.last, player_temp.position
('Frank', 'Wierzbicki', 'forward')
>>> data.close()
```

როგორც მაგალითიდან ჩანს shelve მოდული ობიექტებს ინახავს, ლექსიკონის სახით.

მონაცემების წაკითხვა ხდება მსგავსი პრინციპით მაგალითად player_temp = data['player3'].

5. მონაცემების ტერმინალზე გამოტანა.

მონაცემების ტერმინალზე გამოსატანად გამოიყენება print ოპერატორი.

პროგრამული კოდი

```
# % სიმბოლოს გამოყენება
>>> x = 5
>>> y = 10
>>> print 'The sum of %d and %d is %d' % (x, y, (x + y))
The sum of 5 and 10 is 15
>>> adjective = "awesome"
```

```
>>> print 'Jython programming is %s' % (adjective)
Jython programming is awesome
```

float ტიპის მონაცემების გამოსატანად გამოიყენება %f ფორმატი.

პროგრამული კოდი

```
>>> pi = 3.14
>>> print 'Here is some formatted floating point arithmetic: %.2f' % (pi + y)
Here is some formatted floating point arithmetic: 13.14
>>> print 'Here is some formatted floating point arithmetic: %.3f' % (pi + y)
Here is some formatted floating point arithmetic: 13.140
```

ობიექტზე ორიენტირებული პითონი

1.ძირითადი სინტაქსი

პითონში კლასის დაწერა წარმოადგენს მარტივ საქმეს, ჯერ განისაზღვრება რაღაც მდგომარეობა,

რომელიც ასახავს ობიექტს და შემდეგ განისაზღვრება ფუნქციები, რომლებიც მოქმედებენ და ცვლიან ამ მდგომარეობას.

ახლა დავწეროთ მარტივი კლასი Car, და შევინახოთ ის car.py ფაილში.

პროგრამული კოდი

```
class Car(object):

    NORTH = 0
    EAST = 1
    SOUTH = 2
    WEST = 3

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
        self.direction = self.NORTH

    def turn_right(self):
        self.direction += 1
        self.direction = self.direction % 4

    def turn_left(self):
        self.direction -= 1
        self.direction = self.direction % 4

    def move(self, distance):
        if self.direction == self.NORTH:
            self.y += distance
```

```
        elif self.direction == self.SOUTH:
            self.y -= distance
        elif self.direction == self.EAST:
            self.x += distance
        else:
            self.x -= distance

    def position(self):
        return (self.x, self.y)
```

ეს კი დამტესტი ფუნქცია

პროგრამული კოდი

```
from car import Car

def test_car():
    c = Car()
    c.turn_right()
    c.move(5)
    assert (5, 0) ==  c.position()

    c.turn_left()
    c.move(3)
```

```
assert (5, 3) == c.position()
```

ახლა კი აღვწეროთ Car კლასი, Car კლასის პირველ ხაზზე წერია, რომ ჩვენ ვქმნით კლასს სახელად Car, რომელიც აფართოებს Object კლასს. Object კლასი არის საერთო კლასი ყველა სხვა კლასისა. Object კლასი შეიცავს ძირითად ფუნქციონალობას რაც ახასიათებს ნებისმიერ პითონში შექმნილ ობიექტს. 3-6 ხაზი პროგრამისა განსაზღვრავს კლასის ატრიბუტებს, კლასის ატრიბუტებზე წვდომა შეიძლება კლასის ობიექტის შექმნის გარეშე. 8-11 ხაზი პროგრამისა განსაზღვრავს ობიექტის ინიციალიზაციის __init__() მეთოდს, ამ მეთოდის გამოძახება ხდება მოცემული კლასის ობიექტის შექმნისას. მოცემული ინიციალიზაციის მეთოდი განსაზღვრავს მანქანის საწყის პოზიციას - (0, 0) და მანქანის მიმართულებას(ჩრდილოეთი). მოცემულ ფუნქციას პირველ არგუმენტად გადაეცემა self პარამეტრი, რომელიც მიუთითებს მიმდინარე ობიექტს. ამ პარამეტრის საშუალებით, შესაძლებელია კონკრეტული ობიექტის პარამეტრებზე წვდომა.
13-19 სტრიქონები განსაზღვრავენ მეთოდებს, რომელთა დანიშნულებაა მანქანის მარჯვნივ და მარცხნივ მობრუნება.
სტრიქონები 21-29 აღწერენ მეთოდს, რომელიც ახდენს მანქანის გადაადგილებას პირდაპირ, იმ მიმართულებით, რა მიმართულებითაც იმყოფება მანქანა მოცემულ მომენტში.

1.2. pickle მოდული

ახლა დავამატოთ ჩვენ კლასს ახალი მეთოდები, რომელთა დანიშნულებაცაა მიმდინარე ობიექტის მდგომარეობის შენახვა ფაილში და ჩატვირთვა ფაილიდან.

პროგრამული კოდი

```
 def save(self, filename):
     state = (self.direction, self.x, self.y)
     pickle.dump(state, open(filename,'wb'))


  def load(self, filename):
     state = pickle.load(open(filename,'rb'))
     (self.direction, self.x, self.y) = state
```

იმის გამო, რომ ყოველი ახალი ობიექტის მდგომარეობა უნდა იქნას შენახული ცალკე ფაილში, ამიტომ ჩვენ დაგვჭირდება ობიექტის ახალი ატრიბუტი name.

პროგრამული კოდი

```
 def __init__(self, name, x=0, y=0):
     self.name = name
     self.x = x
     self.y = y
     self.direction = self.NORTH
```

ამის შემდეგ ყოველი მობრუნების და გადაადგილების ფუნქციებს, ბოლოში ვამატებთ save()-მეთოდის გამოძახებას, რათა შენახულ იქნას ობიექტის ახალი მდგომარეობა ფაილში.

პროგრამული კოდი

```
 def turn_right(self):
     self.direction += 1
```

```
        self.direction = self.direction % 4
        self.save(self.name)

    def turn_left(self):
        self.direction -= 1
        self.direction = self.direction % 4
        self.save(self.name)

    def move(self, distance):
        if self.direction == self.NORTH:
            self.y += distance
        elif self.direction == self.SOUTH:
            self.y -= distance
        elif self.direction == self.EAST:
            self.x += distance
        else:
            self.x -= distance
        self.save(self.name)
```

2. ობიექტის ატრიბუტების ძებნა

თქვენ შეიძლება გაგიჩნდეთ შემდეგი კითხვა, როგორ არის დაკავშირებული კლასის ცვლადები NORTH, SOUTH, EAST და WEST-ი self მითითებასთან.

როცა ჩვენ ვახდენთ წვდომას ობიექტის ცვლადზე, ჯერ ხდება self მითითებასთან

დაკავშირებული ცვლადების შემოწმება, ხოლო თუ არ იქნა ნაპოვნი საძიებო ცვლადი, ხდება ზემოთ ასვლა და იწყება კლასის ატრიბუტებს შორის ძიება.

მაგალითი

პროგრამული კოდი

```
>>> class Foobar(object):
...     def __init__(self):
...         self.somevar = 42
...     class_attr = 99
...
>>>
>>> obj = Foobar()
>>> obj.somevar
42
>>> obj.class_attr
99
>>> obj.not_there
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foobar' object has no attribute 'not_there'
>>>
```

თუ ცვლადებს დავაკავშირებთ self მითითებასთან, მაშინ ეს ცვლადები სხვადასხვა

ობიექტებისთვის იქნება სხვადასხვა, ხოლო თუ ცვლადებს დავაკავშირებთ კლასთან მაშინ ეს

ცვლადები საერთო იქნება მოცემული კლასის ობიექტებისათვის.

ობიექტიდან კლასის ცვლადზე წვდომა ხდება შემდეგი სინტაქსით

object.__class__.class_attr.

პროგრამული კოდი

```
>>> other = Foobar()
>>> other.somevar
42
>>> other.class_attr
99

>>> # obj და other გააჩნიათ somevar ატრიბუტის სხვადასხვა მნიშვნელობები.
>>> obj.somevar = 77
>>> obj.somevar
77
>>> other.somevar
42

>>> # თუ მინიჭებას მოვახდენთ other.class_attr-სათვის, შეიქმნება
>>> # ობიექტის ცვლადი class_attr, რომელიც გადაფარავს კლასის ცვლადს
>>> other.class_attr = 66
>>> other.class_attr
66
```

```
>>> # და არ შეიცვლება class_attr-ის მნიშვნელობა სხვა ობიექტებისათვის
>>> obj.class_attr
99

>>> # ჩვენ კვლავ შეგვიძლია მოვახდინოთ წვდომა კლასის,
>>> # class_attr ატრიბუტზე შემდეგნაირად
>>> other.__class__.class_attr
99

>>> # და თუ წავშლით ობიექტის ატრიბუტს სახელად class_attr,
>>> # მაშინ class_attr სიტყვით კვლავ შეგვეძლება კლასის ცვლადზე წვდომა
>>> del other.class_attr
>>> other.class_attr
99

>>> # თუ კლასის ცვლადს, რომელიც არის მუტირებადი შეცვლით, მაშინ ის შეიცვლება
>>> # ყველა ობიექტისათვის.
>>> Foobar.class_list = []
>>> obj.class_list
[]
>>> other.class_list
[]
>>> obj.class_list.append(1)
>>> obj.class_list
```

```
[1]
>>> other.class_list
[1]
```

ობიექტის ატრიბუტების დათვალიერება შეიძლება __dict__ ლექსიკონის გამოყენებით.

პროგრამული კოდი

```
>>> obj = Foobar()
>>> obj.__dict__
{'somevar': 42}
```

კლასის ატრიბუტების დასათვალიერებლად ვიქცევით იგივენაირად

პროგრამული კოდი

```
>>> Foobar.__dict__
{'__module__': '__main__',
    'class_attr': 99,
    '__dict__': <attribute '__dict__' of 'Foobar' objects>,
    '__init__': <function __init__ at 1>}
```

3. მემკვიდრეობა და გადატვირთვა

თუ ჩვენ ვაპროგრამებთ კლასებს, რომელთაც გააჩნიათ, რაიმე საერთო თვისება, ჩვენ შეგვიძლია ეს თვისება გავიტანოთ მშობელ კლასში და ჩვენი კლასები გამოვაცხადოთ ამ მშობელი კლასის მემკვიდრეებად.

პროგრამული კოდი

```
class Animal(object):
    def sound(self):
        return "I don't make any sounds"
class Goat(Animal):
    def sound(self):
        return "Bleeattt!"
class Rabbit(Animal):
    def jump(self):
        return "hippity hop hippity hop"
class Jackalope(Goat, Rabbit):
    pass
```

მოცემული კლასები ავტომატურად მემკვიდრეობით იღებენ მშობელი Animal კლასის sound მეთოდს და საჭიროების შემთხვევაში ახდენენ მის გადაფარვას.

პროგრამული კოდი

```
>>> from animals import *
>>> animal = Animal()
>>> goat = Goat()
```

```
>>> rabbit = Rabbit()

>>> jack = Jackalope()

>>> animal.sound()

"I don't make any sounds"

>>> animal.jump()

Traceback (most recent call last):

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

AttributeError: 'Animal' object has no attribute 'jump'

>>> rabbit.sound()

"I don't make any sounds"

>>> rabbit.jump()

'hippity hop hippity hop'

>>> goat.sound()

'Bleeattt!'

>>> goat.jump()

Traceback (most recent call last):

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

AttributeError: 'Goat' object has no attribute 'jump'

>>> jack.jump()

'hippity hop hippity hop'

>>> jack.sound()

'Bleeattt!'
```

თუ მოვახდენთ sound მეთოდის გადაფარვას, მაშინ მოხდება ამ გადაფარული მეთოდის

გამოძახება თუ არა და გამოიძახება მშობელი Animal კლასის მეთოდი.

jackalope-ის შემთხვევაში გვაქვს მრავლობითი მემკვიდრეობა და ის მემკვიდრეობით იღებს Goat და Rabbit კლასების მეთოდებს.

isinstance - მეთოდი გამოიყენება იმის დასადგენად არის თუ არა რაიმე ობიექტი, კონკრეტული კლასის წარმომადგენელი.

პროგრამული კოდი

```
>>> isinstance(bunny, Rabbit)
True
>>> isinstance(bunny, Animal)
True
>>> isinstance(bunny, Goat)
False
```

ჩვენ შეიძლება მშობელი კლასის მეთოდის გაფართოება გვინდოდეს, ამისთვის ვიყენებთ

super() ფუნქციას.

პროგრამული კოდი

```
class EasterBunny(Rabbit):
    def sound(self):
        orig = super(EasterBunny, self).sound()
        return "%s - but I have eggs!" % orig
```

ამ თავში ჩვენ განვიხილეთ კლასის მიერ მეთოდების მემკვიდრეობით მიღების საკითხი,

ანალოგიური სიტუაცია გვაქვს ატრიბუტებთან მიმართებაშიც.

4. სპეციალური მეთოდები

პითონში სპეციალური მეთოდები იწყებიან და მთავრდებიან __ ქვედა ხაზის სიმბოლოებით.

განვიხილოთ მეთოდი __str__. ეს მეთოდი, გამოიყენება ობიექტის სტრიქონის სახით

წარმოსადგენად.

მაგალითი

პროგრამული კოდი

```
 from __future__ import with_statement
  from contextlib import closing
  from pickle import dumps, loads


  def write_object(fout, obj):
      data = dumps(obj)
      fout.write("%020d" % len(data))
      fout.write(data)


  def read_object(fin):
      length = int(fin.read(20))
      obj = loads(fin.read(length))
      return obj


  class Simple(object):
```

```
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return "Simple[%s]" % self.value

with closing(open('simplefile','wb')) as fout:
    for i in range(10):
        obj = Simple(i)
        write_object(fout, obj)

print "Loading objects from disk!"
print '=' * 20

with closing(open('simplefile','rb')) as fin:
    for i in range(10):
        print read_object(fin)
```

ამ მაგალითში write_object-მეთოდი ინახავს ობიექტს ფაილში, რომელიც მუშაობს შემდეგნაირად დაჰყავს ობიექტი სტრიქონზე dumps() ფუნქციის გამოყენებით და წერს ამ სტრიქონის სიგრძესა და სტრიქონს ფაილში. read_object-კი კითხულობს მონაცემებს ფაილიდან, დაჰყავს ის ობიექტზე და აბრუნებს მას.
ობიექტის print ოპერატორით ბეჭდვისას გამოიძახება __str__ მეთოდი, რის შედეგადაც ვღებულობთ.

პროგრამული კოდი

```
Loading objects from disk!
====================
Simple[0]
Simple[1]
Simple[2]
Simple[3]
Simple[4]
Simple[5]
Simple[6]
Simple[7]
Simple[8]
Simple[9]
```

5. პროტოკოლები

იტერატორები არიან ობიექტები, რომლებიც საშუალებას იძლევიან დააბრუნონ მომდევნო ელემენტი, ყოველ გამოძახებაზე. იმისათვის, რომ ობიექტი იყოს იტერატორი, საჭიროა მას გააჩნდეს next() და __iter__ მეთოდები. (აკმაყოფილებდეს იტერაციის პროტოკოლს)

პროგრამული კოდი

```
class PickleStream(object):
    """
    This stream can be used to stream objects off of a raw file stream
```

```
"""
def __init__(self, file):
    self.file = file

def write(self, obj):
    data = dumps(obj)
    length = len(data)
    self.file.write("%020d" % length)
    self.file.write(data)

def __iter__(self):
    return self

def next(self):
    data = self.file.read(20)
    if len(data) == 0:
        raise StopIteration
    length = int(data)
    return loads(self.file.read(length))

def close(self):
    self.file.close()
```

ამის შემდეგ PickleStream კლასის ობიექტის ელემენტების კითხვა ბევრად უფრო მარტივდება.

პროგრამული კოდი

```
 with closing(PickleStream(open('simplefile','wb'))) as stream:
     for i in range(10):
         obj = Simple(i)
         stream.write(obj)


  with closing(PickleStream(open('simplefile','rb'))) as stream:
     for obj in stream:
         print obj
```

სპეციალური მეთოდების ორი ყველაზე ხშირი გამოყენებაა პროქსის შექმნა და საკუთარი კონტეინერის შექმნა. პროქსი კლასი დგას გამომძახებელ კლასსა და გამოძახებულს შორის.

როგორც ვიცით როდესაც ობიექტის მეთოდის გამოძახებისას, ეს მეთოდი კლასში არ იმყოფება,

ხდება __getattr__ მეთოდის გამოძახება, რომელიც უზრუნველყოფს მეთოდის ძიებას მშობელ კლასში. ჩვენ შეგვიძლია გადავფაროთ ეს მეთოდი, საჭიროებისამებრ.

მარტივი პროქსის მაგალითი.

პროგრამული კოდი

```
 class SimpleProxy(object):
     def  __init__(self, parent):
         self._parent = parent
```

```
    def __getattr__(self, key):
        return getattr(self._parent, key)
```

ახლა განვიხილოთ შემდეგი კოდი

პროგრამული კოდი

```
>>> class TownCrier(object):
...     def __init__(self, parent):
...         self._parent = parent
...     def __getattr__(self, key):
...         print "Accessing : [%s]" % key
...         return getattr(self._parent, key)
...
>>> class Calc(object):
...     def add(self, x, y):
...         return x + y
...     def sub(self, x, y):
...         return x - y
...
>>> calc = Calc()
>>> crier = TownCrier(calc)
>>> crier.add(5,6)
Accessing : [add]
```

11

>>> crier.sub(3,6)

Accessing : [sub]

-3

ამ მაგალითში, როდესაც ჩვენ ვცდილობთ crier.add(5,6) მეთოდის შესრულებას, ჯერ ხდება მეთოდის ძიება TownCrier-კლასში, და რადგანაც ეს მეთოდი არ გააჩნია მოცემულ კლასს, გამოიძახება __getattr__ მეთოდი, რომელიც ჩვენ შემთხვევაში ძებნას გააგრძელებს Calc კლასში იპოვის მას და შეასრულებს. TownCrier-ამ შემთხვევაში არის პროქსი გამომძახებელ კლასსა და გამოძახებულს შორის.

ახლა განვიხილოთ საკუთარი ლექსიკონის შექმნის საკითხი. ამ შემთხვევაში, ჩვენ კლასში უნდა განვსაზღვროთ შემდეგი მეთოდები.

პროგრამული კოდი

__getitem__(self, key)

__setitem__(self, key, value)

__delitem__(self, key)

__contains__(self, item)

__len__(self)

შემდეგნაირად

პროგრამული კოდი

```
class  SimpleDict(object):
    def __init__(self):
        self._items = []

    def __getitem__(self, key):
       # do a brute force key lookup and return the value
        for k, v in self._items:
            if k == key:
                return v
        raise LookupError, "can't find key: [%s]" % key

    def __setitem__(self, key, value):
        # do a brute force search and replace
        # for the key if it exists. Otherwise append
        # a new key/value pair.
        for i, (k , v) in enumerate(self._items):
            if k == key:
                self._items[i][1] = v
                return
        self._items.append((key, value))

    def __delitem__(self, key):
        # do a brute force search and delete
        for i, (k , v) in enumerate(self._items):
```

```
            if k == key:
                del self._items[i]
                return
        raise LookupError, "Can't find [%s] to delete" % key
```

მოცემულ კოდი შევინახოთ foo.py ფაილში

პროგრამული კოდი

```
 >>> from foo import *
  >>> x = SimpleDict()
  >>> x[0] = 5
  >>> x[15]= 32
  >>> print x[0]
  5
  >>> print x[15]
  32
```

იმისათვის, რომ მივიღოთ ლექსიკონის ზომის და გასაღების არსებობის შესახებ ინფორმაცია. საჭიროა შემდეგი მეთოდების განსაზღვრა.

პროგრამული კოდი

```
 def __contains__(self, key):
      return key in [k for (k, v) in self._items]
```

```
def __len__(self):
    return len(self._items)
```

6. გაჩუმებითი არგუმენტები

პითონის კლასის შექმნისას, ჩვენ შეგვიძლია მეთოდის სიგნატურაში განვსაზღვროთ ფორმალური პარამეტრების გაჩუმებითი მნიშვნელობები.

პროგრამული კოდი

```
>>> class Tricky(object):
...     def mutate(self, x=[]):
...         x.append(1)
...         return x
...
>>> obj = Tricky()
>>> obj.mutate()
[1]
>>> obj.mutate()
[1, 1]
>>> obj.mutate()
[1, 1, 1]
```

იქიდან გამომდინარე, რომ პითონში მეთოდებიც წარმოადგენენ ობიექტებს,

მოცემულ მაგალითში mutate-ფუნქცის ყოველი გამოძახება შეცვლის მეთოდის ცვლადს,

რომლის ინიციალიზაცია ხდება ერთხელ x=[]. იმის გამო რომ კლასის ობიექტები მიუთითებენ

ერთი და იგივე მეთოდს ეს ცვლილება გავრცელდება კლასის ყველა ობიექტზე.

7. მეთოდების მიბმა კლასზე, პროგრამის მუშაობისას.

ჩვენ შეგვიძლია კონკრეტულ კლასზე მივაბათ მეთოდები, პროგრამის მუშაობის პროცესში.

პროგრამული კოდი

```
>>> def some_func(self, x, y):
...     print "I'm in object: %s" % self
...     return x * y
...
>>> import new
>>> class Foo(object): pass
...
>>> f = Foo()
>>> f
<__main__.Foo object at 0x1>
>>> Foo.mymethod = new.instancemethod(some_func, f, Foo)
>>> f.mymethod(6,3)
I'm in object: <__main__.Foo object at 0x1>
```

ეს საჭიროა იმ შემთხვევებში, როცა ჩვენ გვინდა უკვე დაწერილი კლასის ფუნქციონალური გაფართოება.

8. ატრიბუტებზე წვდომის კეშირება.

დავუშვათ მოვცემული გვაქვს რაღაც კლასი, რომლის მეთოდის შესრულება მოითხოვს დიდ დროით დანახარჯებს. ამ შემთხვევაში კარგი აზრი იქნებოდა, თუ დავაკეშირებდით უკვე გამოთვლილ მონაცემებს, დროის დაზოგვის მიზნით.

ჩვენს მიერ განხილული მეთოდი, რეალურად არაფერს არ აკეთებს ის უბრალოდ აპაუზებს 1 წამით და შემდეგ აბრუნებს რაღაც მონაცემებს.

პროგრამული კოდი

```
import time
class Foobar(object):
    def slow_compute(self, *args, **kwargs):
        time.sleep(1)
        return args, kwargs, 42
```

ახლა დავწეროთ დეკორატორი ფუნქცია, რომელიც ასრულებს შემდეგ ოპერაციებს.

პროგრამული კოდი

```
import hashlib
def cache(func):
    # ფუნქციის სახელის განსაზღვრა
    func_name = func.func_name
```

```
    def inner(self, *args, **kwargs):
        # ვითვლით უნიკალურ მნიშვნელობას ფუნქციის არგუმენტებისათვის.
        arghash = hashlib.sha1(str(args) + str(kwargs)).hexdigest()
        cache_name = '_cache_%s_%s' % (func_name, arghash)

        if hasattr(self, cache_name):
            # თუ მონაცემები კეშირებულია დავაბრუნოთ ისინი.
            print "Fetching cached value from : %s" % cache_name
            return getattr(self, cache_name)
        result = func(self, *args, **kwargs)
        setattr(self, cache_name, result)
        return result
    return inner
```

getattr, setattr, და hasattr მეთოდები აბრუნებენ, აყენებენ და ამოწმებენ ობიექტის ატრიბუტებს.

მოცემული დეკორატორი ფუნქცია შეიძლება გამოყენებულ იქნას ნებისმიერი კლასის ნებისმიერი მეთოდისათვის.

პროგრამული კოდი

```
 @cache
  def slow_compute(self, *args, **kwargs):
      time.sleep(1)
```

return args, kwargs, 42

შეცდომების დამუშავება

1. შეცდომების დამუშავება

შეცდომების დამუშავება პითონში, ხდება იგივენაირად როგორც მაგალითად ჯავაში.

ჯავაში შეცდომების დამუშავება ხდება შემდეგი სინტაქსის გამოყენებით.

პროგრამული კოდი

```
try {
 // გარკვეული ოპერაციები, რიმელმაც შეიძლება გამოიწვიოს შეცდომა.
 } catch (ExceptionType messageVariable) {
 // ვამუშავებთ შეცდომას.
 } finally {
 // კოდი, რომელიც ყველა შემთხვევაში სრულდება.
 }
```

1.1. შეცდომების დაჭერა

შეცდომები პროგრამაში შეიძლება გამოიწვიოს სხვადასხვა ოპერაციებმა, შეცდომები შეიძლება

იქნას დაჭერილი და დამუშავებული. დამუშავებაში იგულისხმება პროგრამის მუშაობის

შეწყვეტა და შეცდომის შესახებ ინფორმაციის გამოტანა ან შეცდომის გასწორება და პროგრამის

მუშაობის აღდგენა. შეცდომის დასაჭერად გამოიყენება try-except ბლოკი.

მაგალითი

პროგრამული კოდი

```
# ეს ფუნქცია იყენებს try-except ბლოკს, რომ დაბეჭდოს ინფორმაცია შეცდომის შესახებ,
 # თუ მომხმარებელი გადასცემს გამყოფად 0-ს.
 >>> from __future__ import division
 >>> def divide_numbers(x, y):
 ...     try:
 ...         return x/y
 ...     except ZeroDivisionError:
 ...         return 'You cannot divide by zero, try again'
 ...
 # მცდელობა 8-ის 3-ზე გაყოფისას
 >>> divide_numbers(8,3)
 2.6666666666666665
 # მცდელობა 8-ის 0-ზე გაყოფისას
 >>> divide_numbers(8, 0)
 'You cannot divide by zero, try again'
```

პითონში შეცდომების დამუშავება არ არის სავალდებულო განსხვავებით ჯავასგან, სადაც არსებობს სპეციალური ტიპი შეცდომებისა, რომელთა დამუშავებასაც გვაიძულებს კომპილატორი.

შეცდომები პითონში არის სპეციალური კლასის ობიექტი.

ქვემოთ მოცემულია შეცდომების კლასები, რომელიც გააჩნია პითონს.

პროგრამული კოდი

BaseException	This is the root exception for all others
GeneratorExit	Raised by close() method of generators for terminating iteration
KeyboardInterrupt	Raised by the interrupt key
SystemExit	Program exit
Exception	Root for all non-exiting exceptions
StopIteration	Raised to stop an iteration action
StandardError	Base class for all built-in exceptions
ArithmeticError	Base for all arithmetic exceptions
FloatingPointError	Raised when a floating-point operation fails
OverflowError	Arithmetic operations that are too large
ZeroDivisionError	Division or modulo operation with zero as divisor
AssertionError	Raised when an assert statement fails
AttributeError	Attribute reference or assignment failure
EnvironmentError	An error occurred outside of Python
IOError	Error in Input/Output operation
OSError	An error occurred in the os module
EOFError	input() or raw_input() tried to read past the end of a file
ImportError	Import failed to find module or name
LookupError	Base class for IndexError and KeyError
IndexError	A sequence index goes out of range
KeyError	Referenced a non-existent mapping (dict) key
MemoryError	Memory exhausted
NameError	Failure to find a local or global name

UnboundLocalError Unassigned local variable is referenced

ReferenceError Attempt to access a garbage-collected object

RuntimeError Obsolete catch-all error

NotImplementedError Raised when a feature is not implemented

SyntaxError Parser encountered a syntax error

IndentationError Parser encountered an indentation issue

TabError Incorrect mixture of tabs and spaces

SystemError Non-fatal interpreter error

TypeError Inappropriate type was passed to an

ValueError Argument error not covered by TypeError or a more precise error

Warning Base for all warnings

try-except-finally ბლოკს პითონში აქვს შემდეგი სახე.

პროგრამული კოდი

```
try:
    # ოპერაციები, რომელმაც შეიძლება გამოიწვიოს შეცდომა
except Exception, value:
    # ვამუშავებთ შეცდომებს
finally:
    # ოპერაციები, რომელიც სრულდება ნებისმიერ შემთხვევაში
```

try ბლოკში იწერება პროგრამული კოდი, რომელმაც შესაძლოა გამოიწვიოს შეცდომა, except ბლოკში იწერება შეცდომების დამმუშავებელი კოდი, finally ბლოკში იწერება კოდი, რომელიც სრულდება ნებისმიერ შემთხვევაში, ხდება პროგრამაში შეცდომა თუ არა. finally ბლოკში, როგორც წესი იწერება ისეთი კოდი, როგორიცაა ფაილის დახურვის ოპერაცია და ა.შ.

შეიძლება გვქონდეს try-finally ბლოკი.

პროგრამული კოდი

```
try:
    # ოპერაციები, რომელმაც შეიძლება გამოიწვიოს შეცდომა
finally:
    # ოპერაციები, რომელიც სრულდება ნებისმიერ შემთხვევაში
```

try-except-else - ბლოკს აქვს ფორმა

პროგრამული კოდი

```
try:
    # ოპერაციების შესრულება, რომელმაც შეიძლება გამოიწვიოს შეცდომები
except:
    # შეცდომების დამუშავება
else:
    # ოპერაციები, რომელიც სრულდება იმ შემთხვევაში, როდესაც
    # არ ხდება შეცდომა try ბლოკში.
```

თუ ჩვენ გვინდა, რომელიღაც ტიპის შეცდომის დაჭერა, რომელიც ხდება პროგრამის შესრულების პროცესში, უნდა მოხდეს მისი მითითება try-except - ბლოკში.

პროგრამული კოდი

```
# კოდი შეცდომის დამმუშავებლის გარეშე.
>>> x = 10
>>> z = x / y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'y' is not defined


# იგივე პროგრამის ფრაგმენტი try-except ბლოკით.
>>> x = 10
>>> try:
...     z = x / y
... except NameError, err:
...     print "One of the variables was undefined: ", err
...
One of the variables was undefined:  name 'y' is not defined
```

ამ შემთხვევაში try-except ბლოკს აქვს ფორმა

პროგრამული კოდი

```
try:
    # კოდი
 except ExceptionType, messageVar:
    # კოდი
```

იგივე try-except ბლოკი შეიძლება ჩაიწეროს შემდეგნაირად.

პროგრამული კოდი

```
try:
    # კოდი
 except ExceptionType as messageVar:
    # კოდი
```

შეცდომის დაჭერის შემდეგ ჩვენ შეგვიძლია გამოვიყენოთ type()-ფუნქცია, რათა დავადგინოთ შეცდომის ტიპი.

პროგრამული კოდი

```
# ამ მაგალითში ჩვენ ვიჭერთ შეცდომას და შემდეგ ვადგენთ შეცდომის ტიპს.
 >>> try:
 ...     8/0
 ... except Exception, ex1:
 ...     'An error has occurred'
 ...
```

```
'An error has occurred'
>>> ex1
ZeroDivisionError('integer division or modulo by zero',)
>>> type(ex1)
<type 'exceptions.ZeroDivisionError'>
>>>
```

არსებობს კიდევ ერთი ფუნქცია sys.exc_info(), რომელიც განსაზღვრავს შეცდომის ტიპსა და შეცდომის შეტყობინებას.

პროგრამული კოდი

```
# try-except-ით ვიჭერთ ნებისმიერი ტიპის შეტყობინებას.
>>> x = 10
>>> try:
...     z = x / y
... except:
...     print "Unexpected error: ", sys.exc_info()[0], sys.exc_info()[1]
...
Unexpected error:  <type 'exceptions.NameError'> name 'y' is not defined
```

პროგრამის შიგნით შეიძლება მოხდეს ერთზე მეტი შეცდომა, პითონში სხვადასხვა ტიპის შეცდომების დამუშავების ორი გზა არსებობს. პირველი თითოეული ტიპის შეცდომისათვის

ცალკე except ბლოკის განსაზღვრა, მეორე except ბლოკში დასამუშავებელი შეცდომების ტიპების ერთმანეთისგან მძიმეებით გამოყოფა.

პროგრამული კოდი

```
# იჭერს NameError და ZeroDivisionError-ს
 equation
>>> try:
...     z = x/y
... except(NameError, ZeroDivisionError), err:
...     "An error has occurred, please check your values and try again"
...
'An error has occurred, please check your values and try again'


# რამოდენიმე except ბლოკის გამოყენება
>>> x = 10
>>> y = 0
>>> try:
...     z = x / y
... except NameError, err1:
...     print err1
... except ZeroDivisionError, err2:
...     print 'You cannot divide a number by zero!'
...
You cannot divide a number by zero!
```

არსებობს შეცდომების მშობელი კლასი და მისი ქვეკლასები. მაგალითად Lookup კლასს გააჩნია მემკვიდრეები KeyError და IndexError.

განვიხილოთ მშობელი Lookup კლასის დაჭერის მაგალითი.

პროგრამული კოდი

```
# ამ მაგალითში find_value ფუნქცია აბრუნებს კონტეინერის ელემენტს
 # ფუნქცია იღებს არგუმენტად სიას ან ლექსიკონს და ახდენს Lookup შეცდომის
 # დამუშავებას
 >>> def find_value(obj, value):
 ...     try:
 ...         return obj[value]
 ...     except LookupError, ex:
 ...         return 'An exception has been raised, check your values and try again'
 ...


# ვქმნით ლექსიკონის და სიის ობიექტს და ვცდილობთ წავიკითხოთ ისეთი ელემენტი
 # რომელსაც ისინი არ შეიცავენ
 >>> mydict = {'test1':1,'test2':2}
 >>> mylist = [1,2,3]
 >>> find_value(mydict, 'test3')
 'An exception has been raised, check your values and try again'
 >>> find_value(mylist, 2)
```

```
3
>>> find_value(mylist, 3)
'An exception has been raised, check your values and try again'
>>>
```

თუ რამოდენიმე შეცდომის დამუშავებისთვის ცალ-ცალკე განვსაზღვრავთ ბლოკებს, მაშინ ნაპოვნი შეცდომა იქნება დამუშავებული შესაბამისი პირველივე except ბლოკის მიერ.

პროგრამული კოდი

```
# ყოველ შეცდომას ვამუშავებთ ცალ-ცალკე, ჩვენ შემთხვევაში
 # შეცდომა პირველივე ნაპოვნი except ბლოკის მიერ იქნება დამუშავებული.
 # აქედან გამომდინარე "except LookupError" ბლოკი არასდროს არ შესრულდება.
 >>> def find_value(obj, value):
 ...     try:
 ...         return obj[value]
 ...     except KeyError:
 ...         return 'The specified key was not in the dict, please try again'
 ...     except IndexError:
 ...         return 'The specified index was out of range, please try again'
 ...     except LookupError:
 ...         return 'The specified key was not found, please try again'
 ...
 >>> find_value(mydict, 'test3')
 'The specified key was not in the dict, please try again'
 >>> find_value(mylist, 3)
```

```
'The specified index was out of range, please try again'
>>>
```

try-except ბლოკები შეიძლება ჩადგმული იყოს ერთმანეთში. ამ შემთხვევაში შიდა try-except შეცდომის დამმუშავებელი ბლოკის შესრულების შემდეგ პროგრამის მუშაობა გაგრძელდება და ახალი შეცდომის მოხდენის შემთხვევაში ამუშავდება უკვე გარე try-except ბლოკი.

პროგრამული კოდი

```
# შევასრულოთ გაყოფის ოპერაციები რიცხვებზე
try:
    # do some work
    try:
        x = raw_input ('Enter a number for the dividend:  ')
        y = raw_input('Enter a number to divisor: ')
        x = int(x)
        y = int(y)
    except ValueError:
        # ვიჭერთ შეცდომას და გადავდივართ გარეთა try-except ბლოკზე
        print 'You must enter a numeric value!'
    z = x / y
except ZeroDivisionError:
    # ვიჭერთ შეცდომას
```

```
    print 'You cannot divide by zero!'
except TypeError:
    print 'Retry and only use numeric values this time!'
else:
    print 'Your quotient is: %d' % (z)
```

შიდა try-except ბლოკის შიგნით, პროგრამაში, კლავიატურიდან ვკითხულობთ x და y რიცხვებს, ამ დროს შეიძლება მოხდეს ValueError ტიპის შეცდომა. შეცდომის დამუშავების(ან დაუმუშავებლობის) შემდეგ მართვა გადადის გარე try-except ბლოკზე, რომლის შიგნით ხდება უკვე გაყოფის ოპერაციის შესრულება, გაყოფისას შეიძლება მოხდეს ZeroDivisionError და TypeError შეცდომები, რომელსაც ამუშავებს გარე try-except ბლოკი.

1.2. შეცდომების გამოსროლა

შეცდომების გამოსასროლად პითონში გამოიყენება raise ოპერატორი.

შეცდომის გამოსროლა შეიძლება დაგვჭირდეს სხვადასხვა შემთხვევებში, მაგალითად როცა გვინდა საკუთარი შეცდომის კლასის დაწერა და მისი გამოსროლა.

raise ოპერატორს გააჩნია შემდეგი ფორმატი.

პროგრამული კოდი

```
raise ExceptionType or String[, message[, traceback]]
```

შეცდომების გამოსროლის მაგალითები

პროგრამული კოდი

```
>>> raise Exception("An exception is being raised")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: An exception is being raised


>>> raise TypeError("You've specified an incorrect type")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: You've specified an incorrect type
```

შეცდომა შეიძლება გამოსროლილ იქნას შემდეგი ფორმატის გამოყენებითაც

პროგრამული კოდი

```
>>> raise TypeError, "This is a special message"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: This is a special message
```

2. საკუთარი Exception კლასის განსაზღვრა.

საკუთარი Exception კლასის განსასაზღვრავად, ყველაზე მარტივი გზაა,

განვსაზღვროთ Exception კლასის მემკვიდრე კლასი, რომლის კოდი არაფერს არ შეიცავს.

პროგრამული კოდი

```
class MyNewError(Exception):
    pass
```

მის გამოსასროლად საჭიროა შემდეგი კოდი.

პროგრამული კოდი

```
raise MyNewError("Something happened in my program")
```

3. (გამაფრთხილებელი შეტყობინებები) Warnings

Warnings შეიძლება გამოყენებულ იქნას პროგრამაში ნებისმიერ დროს და გამოიყენება, რომ გამოიტანოს ეკრანზე გამაფრთხილებელი შეტყობინება, მაგრამ Warning-ები არ იწვევენ პროგრამის მუშაობის შეწყვეტას. Warning-ების გამოსაყენებლად საჭიროა Warnings მოდულის იმპორტი და შემდეგ warnings.warn() ფუნქციის გამოძახება, რომელსაც გადაეცემა არგუმენტებად გამაფრთხილებელი შეტყობინება და ამ შეტყობინების კლასი.

warning-ები შეიძლება გამოყენებულ იქნას მაგალითად მაშინ, როდესაც ფუნქცია არ აკმაყოფილებს საჭირო მოთხოვნებს(deprecated) და უმჯობესია სხვა ფუნქცია იქნას გამოყენებული მის მაგივრად.

პროგრამული კოდი

```
# warnings მოდულის იმპორტი
 import warnings
```

```
# გამაფრთხილებელი შეტყობინების მაგალითი
warnings.warn("this feature will be deprecated")
warnings.warn("this is a more involved warning", RuntimeWarning)

# ფუნქციაში გამაფრთხილებელი შეტყობინების გამოყენების მაგალითი
# მოცემული შეტყობინება გვამცნობს, რომ მოცემული
# ფუნქციის გამოყენება არ არის მიზანშეწონილი.
>>> def add_days(current_year, days):
...     warnings.warn("This function has been deprecated as of version x.x",
                    DeprecationWarning)
...     num_years = 0
...     if days > 365:
...         num_years = days/365
...     return current_year + num_years
...

# ამ ფუნქციის გამოძახება გამოიწვევს გამაფრთხილებელი შეტყობინების დაბრუნებას
# მაგრამ არ გამოიწვევს პროგრამის მუშაობის შეწყვეტას.
>>> add_days(2009, 450)
__main__:2: DeprecationWarning: This function has been deprecated as of version x.x
2010
```

გამაფრთხილებელი შეტყობინებების კატეგორიები

პროგრამული კოდი

Warning Root warning class

UserWarning A user-defined warning

DeprecationWarning Warns about use of a deprecated feature

SyntaxWarning Syntax issues

RuntimeWarning Runtime issues

FutureWarning Warns that a particular feature will be changing in a future release

გამაფრთხილებელი შეტყობინებების ფილტრები

პროგრამული კოდი

'always' Always print warning message

'default' Print warning once for each location where warning occurs

'error' Converts a warning into an exception

'ignore' Ignores the warning

'module' Print warning once for each module in which warning occurs

'once' Print warning only one time

ფილტრების გამოყენების მაგალითი

პროგრამული კოდი

```
# მოცემული ფილტრით ხდება warning-ის დაყვანა exception-ზე.
>>> warnings.simplefilter('error', UserWarning)
>>> warnings.warn('This will be raised as an exception')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 63, in warn
    warn_explicit(message, category, filename, lineno, module, registry,
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 104, in warn_explicit
    raise message
UserWarning: This will be raised as an exception

# გამოვრთოთ ყველა აქტიური ფილტრი resetwarnings()-ის გამოყენებით
>>> warnings.resetwarnings()
>>> warnings.warn('This will not be raised as an exception')
__main__:1: UserWarning: This will not be raised as an exception

# რეგულარული გამოსახულების გამოყენებით warning-ების გაფილტვრა
# ამ შემთხვევაში იგნორირდება ყველა შეტყობინება,
# რომელიც შეიცავს სიტყვა "one"-ს.
>>> warnings.filterwarnings('ignore', '.*one*.',)
>>> warnings.warn('This is warning number zero')
__main__:1: UserWarning: This is warning number zero
>>> warnings.warn('This is warning number one')
>>> warnings.warn('This is warning number two')
__main__:1: UserWarning: This is warning number two
>>>
```

ჩვენ შეგვიძლია გამაფრთხილებელი შეტყობინების ფილტრის დაყენება, პროგრამის ტერმინალიდან შესრულების დროს. ამისთვის ვიყენებთ -W პარამეტრს.

პროგრამული კოდი

```
# მოცემული გვაქვს test_warnings.py სკრიპტი
 # რომელსაც ვუშვებთ ტერმინალიდან
 import warnings
 def test_warnings():
    print "The function has started"
    warnings.warn("This function has been deprecated", DeprecationWarning)
    print "The function has been completed"

 if __name__ == "__main__":
    test_warnings()



# გავუშვათ პროგრამა -W ოფციის გამოყენების გარეშე
jython test_warnings.py
The function has started
test_warnings.py:4: DeprecationWarning: This function has been deprecated
  warnings.warn("This function has been deprecated", DeprecationWarning)
The function has been completed

# გავუშვათ სკრიპტი და იგნორირება გავუკეთოთ ყველა deprecation
# ტიპის გამაფრთხილებელ შეტყობინებას
```

```
jython -W "ignore::DeprecationWarning::0" test_warnings.py
The function has started
The function has been completed

# გავუშვათ სკრიპტი და ყველა DeprecationWarning-ი დავიყვანოთ exception-ზე.
jython -W "error::DeprecationWarning::0" test_warnings.py
The function has started
Traceback (most recent call last):
  File "test_warnings.py", line 8, in <module>
    test_warnings()
  File "test_warnings.py", line 4, in test_warnings
    warnings.warn("This function has been deprecated", DeprecationWarning)
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 63, in warn
    warn_explicit(message, category, filename, lineno, module, registry,
  File "/Applications/Jython/jython2.5.1rc2/Lib/warnings.py", line 104, in warn_explicit
    raise message
DeprecationWarning: This function has been deprecated
```

4. assertion(მტკიცება)

assertions - გამოიყენება გამოსახულების ჭეშმარიტობის დასადგენად

მისი ფორმატია

assert expression [, message]

მაგალითი

პროგრამული კოდი

```
#  ეს მაგალითი გვიჩვენებს, თუ როგორ მუშაობს მტკიცება(assertion).
 >>> x = 5
 >>> y = 10
 >>> assert x < y, "The assertion is ignored"
 >>> assert x > y, "The assertion raises an exception"
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
 AssertionError: The assertion raises an exception



# მტკიცებების გამოყენება პარამეტრების ვალიდაციისათვის
# აქ ჩვენ ვამოწმებთ, რომ ფუნქციის პარამეტრები იყოს integer ტიპის
>>> def add_numbers(x, y):
...     assert type(x) is int, "The arguments must be integers,
                        please check the first argument"
...     assert type(y) is int, "The arguments must be integers,
                        please check the second argument"
...     return x + y
...


# შევამოწმოთ add_numbers ფუნქცია
>>> add_numbers(3, 4)
7
```

```
# აქ კი გამოისვრის Exception-ს.
>>> add_numbers('hello','goodbye')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in add_numbers
AssertionError: The arguments must be integers, please check the first argument
```

5. context managers

with ოპერატორის გამოსაყენებლად საჭიროა მისი იმპორტი __future__ მოდულიდან.

ფაილებთან ან ბაზებთან მუშაობის დამთავრების შემდეგ, როგორც წესი რესურსების გამონთავისუფლების მიზნით, საჭიროა close() მეთოდის გამოძახება.

with ოპერატორის გამოყენებისას ამის საჭიროება აღარ არის.

მაგალითი

პროგრამული კოდი

```
# მონაცემების წაკითხვა players.txt ფაილიდან
>>> from __future__ import with_statement
>>> with open('players.txt','r') as file:
...     x = file.read()
...
>>> print x
Sports Team Management
---------------------------------
```

Josh - forward

Jim - defense

იმისათვის რომ შესაძლებელი იყოს, with ოპერატორის გამოყენებისა ობიექტთან მიმართებაში,

საჭიროა ობიექტმა განსაზღვროს ორი მეთოდი. __enter__ და __exit__ მეთოდები.

__enter__ მეთოდი with ოპერატორის მიერ გამოიძახება მუშაობის დასაწყისში,

ხოლო __exit__ მეთოდი გამოიძახება with ოპერატორის მიერ მუშაობის დამთავრების

შემდეგ.

მოდულები და პაკეტები

აქამდე ჩვენ განვიხილავდით, მცირე ზომის სკრიპტებს, მაგრამ როდესაც პროგრამა იზრდება

ზომაში, საჭიროა მისი დაყოფა მოდულებად. მოდულები გამოიყენება ბიბლიოთეკების

შესაქმნელად, რომელიც შეიძლება იმპორტირებულ და გამოყენებულ იქნას პროგრამის მიერ.

პითონს მოყვება სტანდარტული ბიბლიოთეკა, რომელიც შეიცავს დიდი რაოდენობით მოდულებს.

1. მოდულის იმპორტი

განვიხილოთ შემდეგი მაგალითი

breakfast.py

პროგრამული კოდი

```
import search.scanner as scanner
 import sys
 class Spam(object):


    def order(self, number):
        print "spam " * number


    def order_eggs():
        print " and eggs!"


    s = Spam()
    s.order(3)
    order_eggs()
```

შემოვიტანოთ რამოდენიმე განმარტება. სახელების სივრცე,(namespace) არის ლოგიკური დაჯგუფება უნიკალური იდენტიფიკატორების. სხვანაირად, რომ ვთქვათ namespace არის სახელების სიმრავლე. მაგალითად თუ ჩვენ პითონის ინტერპრეტატორში ავკრიფავთ dir()-ს ეკრანზე დაიბეჭდება ინტერპრეტატორის namespace-ის სახელები.

პროგრამული კოდი

```
>>> dir()
 ['__doc__', '__name__']
```

ჯერ ვნახოთ რა არის პითონში მოდული. მოდული პითონში ეს არის ფაილი, რომელიც შეიცავს ფუნქციებს, კლასებს და ა.შ. მოდულის სახელი პითონში იგივეა რაც ფაილის სახელი, ჩამოშორებული .py გაფართოება. მაგალითად breakfast.py ფაილი წარმოადგენს მოდულს სახელად breakfast.

ახლა ვნახოთ რა არის __name__ თვისება, თუ ჩვენ მოდულს გავუშვებთ ტერმინალიდან

მაგალითად ასე

პროგრამული კოდი

```
python breakfast.py
```

მაშინ __name__ ატრიბუტის მნიშვნელობა იქნება __main__.

თუ ხდება მოდულის იმპორტი მაშინ მოდულის შიგნით __name__ ატრიბუტის მნიშვნელობა იქნება ამ მოდულის სახელი.

პროგრამული კოდი

```
>>> dir()
 ['__doc__', '__name__']
 >>> __name__
 '__main__'
```

ახლა ვნახოთ რა ხდება, როდესაც ჩვენ ვახდენთ breakfast მოდულის იმპორტს.

პროგრამული კოდი

```
>>> import breakfast
 spam spam spam
```

```
and eggs!
>>> dir()
['__doc__', '__name__', 'breakfast']
>>> import breakfast
>>>
```

პირველ რიგში breakfast მოდულის იმპორტირების შემდეგ, ინტერპრეტატორის სახელების სივრცეში გაჩნდა ერთი ახალი სტრიქონი 'breakfast'.

breakfast - მოდულის იმპორტირებისას შესრულდება breakfast მოდულის შიგნით არსებული კოდი. გარდა ამისა უნდა აღვნიშნოთ ის, რომ breakfast მოდულის მეორედ იმპორტირებისას მოდულის შიგნით არსებული კოდი არ სრულდება.

დავუშვათ ჩვენ გვინდა, რომ მოდულის იმპორტირებისას არ შესრულდეს მისი კოდი, ამისათვის breakfast.py გადავაკეთოთ შემდეგნაირად.

პროგრამული კოდი

```
class Spam(object):
    def order(self, number):
        print "spam " * number
    def order_eggs():
        print " and eggs!"
if __name__ == '__main__':
    s = Spam()
    s.order(3)
    order_eggs()
```

ახლა უკვე შეგვიძლია მოვახდინოთ მოდულის იმპორტი მის შიგნით არსებული კოდის შეუსრულებლად.

პროგრამული კოდი

```
>>> import breakfast
```

მაგრამ თუ მოდულის კოდს გავუშვებთ ტერმინალიდან, მაშინ.

პროგრამული კოდი

```
$ python breakfast.py
 spam spam spam
 and eggs!
```

1.2. import ოპერატორი

import ოპერატორის გამოყენებისას შეიძლება მოხდეს error-ი, თუ მოდული, რომლის იმპორტირებასაც ვახდენთ არ არსებობს.

პროგრამული კოდი

```
>>> try:
 ...     from blah import foo
 ...     print "imported normally"
 ... except ImportError:
 ...     print "defining foo in except block"
 ...     def foo():
 ...         return "hello from backup foo"
```

...

defining foo in except block

>>> foo()

'hello from backup foo'

>>>

2. მარტივი პროგრამა

პროგრამის სტრუქტურას აქვს შემდეგი ფორმა.

პროგრამული კოდი

```
chapter10/
    greetings.py
    greet/
        __init__.py
        hello.py
        people.py
```

მოცემული პროგრამა შედგება ერთი პაკეტისაგან სახელად greet. greet არის პაკეტი რადგან

იგი შეიცავს __init__.py ფაილს. შევნიშნოთ, რომ chapter10 დირექტორია არ არის პაკეტი, რადგან ის არ შეიცავს __init__.py ფაილს.

მოცემული პროგრამა შედგება სამი მოდულისგან greetings, greet.hello და greet.people.

პროგრამული კოდი

greetings.py

```
print "in greetings.py"
import greet.hello
g = greet.hello.Greeter()
g.hello_all()
```

greet/__init__.py

```
print "in greet/__init__.py"
```

greet/hello.py

```
print "in greet/hello.py"
import greet.people as people

class Greeter(object):
   def hello_all(self):
      for name in people.names:
         print "hello %s" % name
```

greet/people.py

```
print "in greet/people.py"
names = ["Josh", "Jim", "Victor", "Leo", "Frank"]
```

გავუშვათ ეს პროგრამა

პროგრამული კოდი

$ python greetings.py

in greetings.py

in greet/__init__.py

in greet/hello.py

in greet/people.py

hello Josh

hello Jim

hello Victor

hello Leo

hello Frank

თავიდან მოცემულ კოდში greetings.py-ის შესრულებისას დაიბეჭდება "in greetings.py"

სტრიქონი, შემდეგ greetings.py პროგრამა ახდენს greet.hello მოდულის იმპორტირებას.

რადგან მოდულის იმპორტირება ხდება პირველად ამიტომ ჯერ შესრულდება greet/__init__.py ფაილში არსებული კოდი, რომელიც დაბეჭდავს "in greet/__init__.py"

სტრიქონს, შემდეგ შესრულდება greet.hello მოდულში არსებული კოდი, რომელიც სტრიქონის

დაბეჭდვის შემდეგ ახდენს greet.people მოდულის იმპორტირებას, რომელიც თავის მხრივ ბეჭდავს "in greet/people.py" სტრიქონს.

3. import ოპერატორის ტიპები

import ოპერატორის ტიპების

პროგრამული კოდი

import module

from module import submodule

from . import submodule

3.1. დავიწყოთ import module, ოპერატორის განხილვით.

ეს ოპერატორი ჩვენს მიერ განხილული ზემოთ მოყვანილი მაგალითისთვის გამოიყურება

შემდეგნაირად.

პროგრამული კოდი

import greet.hello

ამის შემდეგ ამ მოდულზე წვდომა ხდება 'greet.hello' სახელით და არა 'hello'-თი,

როგორც ეს არის ჯავაში.

პროგრამული კოდი

import greet.hello as foo

ასეთი იმპორტის შემდეგ მოდულის ელემენტებზე წვდომა მოხდება 'foo' სახელით.

3.2. from import ოპერატორი

from import ოპერატორს აქვს ფორმა

პროგრამული კოდი

from module import name

იმპორტის ეს ფორმა საშუალებას იძლევა მოდულიდან იმპორტირებულ იქნას მოდულები, კლასები და ფუნქციები, რომლებიც ჩადგმულია სხვა მოდულში.

ჩვენი მაგალითისთვის გვექნება

პროგრამული კოდი

from greet import hello

ამ შემთხვევაში ჩვენ ვახდენთ ქვემოდულის იმპორტირებას, სადაც hello არის ქვემოდული.

თუ გვინდა მოდულში შემავალი ყველა სახელის იმპორტი ვიყენებთ შემდეგ ფორმატს

პროგრამული კოდი

from module import *

მაგრამ ამ ფორმატის გამოყენება არ ითვლება პროგრამირების კარგ სტილად.

3.3. რელაციური import ოპერატორები.

პროგრამული კოდი

from . import module

from .. import module

from .module import submodule

from ..module import submodule

რელაციური import ოპერატორები განსაზღვრავენ მიმდინარე მოდულიდან, რამდენით უნდა ავიდეთ ზედა დირექტორიებში, რომ მოვახდინოთ წვდომა საჭირო მოდულზე. ასეთი ფორმით

მოდულზე წვდომა პროგრამირებაში არ ითვლება კარგ სტილად.

თუ ჩვენ გვინდა მოდულიდან კლასის იმპორტი შეგვიძლია მოვიქცეთ შემდეგნაირად.

პროგრამული კოდი

from greet.hello import Greeter

თუ გვინდა Greeter კლასის იმპორტი greet.people მოდულიდან მაშინ შეგვიძლია

გამოვიყენოთ რელაციური იმპორტი შემდეგნაირად.

პროგრამული კოდი

from hello import Greeter

უმჯობესია რელაციური იმპორტები არ გამოვიყენოთ პროგრამაში.

3.4. მეტსახელების გამოყენება import ოპერატორებში

მეტსახელების გამოყენება import ოპერატორებში, ხდება შემდეგი ფორმატით as სიტყვის

გამოყენებით.

პროგრამული კოდი

import module as alias

from module import submodule as alias

from . import submodule as alias

ჩვენი მაგალითისთვის

პროგრამული კოდი

```
import greet.hello as foo
```

ამის შემდეგ შეგვიძლია გამოვიყენოთ სიტყვა foo, greet.hello-ს მაგივრად.

3.5. მოდულის სახელების დამალვა

ჩვეულებრივ, როცა ხდება მოდულის იმპორტი, მოდულში შემავალი ყველა სახელი არის წვდომადი პროგრამისათვის. (სახელში აქ იგულისხმება ქვემოდულის, კლასის და ფუნქციის სახელები).

თუ გვინდა სახელი არ ჩანდეს მოდულის იმპორტირებისას საჭიროა ეს სახელი იწყებოდეს _ ქვედახაზის სიმბოლოთი. რეალურად ეს სახელები კვლავ არის წვდომადი, მაგრამ მათი იმპორტირება არ ხდება შემდეგი ფორმატის გამოყენებისას 'from module import *'.

მეორე გზა მოდულში შემავალი სახელების დამალვისა არის __all__ სიის შევსება იმ

სახელებით, რომლებიც გვინდა, რომ ჩანდეს იმპორტირებისას. მაგალითად Os

მოდულისათვის __all__ სიას აქვს შემდეგი ფორმა.

პროგრამული კოდი

```
__all__ = ["altsep", "curdir", "pardir", "sep", "pathsep",
        "linesep", "defpath", "name", "path",
        "SEEK_SET", "SEEK_CUR", "SEEK_END"]
```

4. მოდულების ძებნა და ჩატვირთვა

პითონში გამოიყენება sys.path სია, რომლის მიხედვითაც პითონის ინტერპრეტატორი ეძებს და ტვირთავს მოდულებს.

პროგრამული კოდი

```
>>> import sys
 >>> sys.path
 ['', 'C:\\Windows\\system32\\python27.zip',
     'C:\\python27\\DLLs', 'C:\\python27\\lib',
     'C:\\python27\\lib\\plat-win', 'C:\\python27\\lib\\lib-tk',
     'C:\\python27', 'C:\\python27\\lib\\site-packages']
```

მოდულის ჩატვირთვის მცდელობისას, ხდება მოდულის ძებნა ზემოთ(sys.path) მოცემულ დირექტორიებში, თუ მოდული იქნება ნაპოვნი მოხდება ძებნის შეწყვეტა და მისი ჩატვირთვა.

ჩვენ შეგვიძლია დავამატოთ ამ სიას საკუთარი დირექტორია, სადაც შენახული გვექნება

საკუთარი მოდულები. ეს შეიძლება განხორციელდეს append მეთოდის საშუალებით.