თავი 5
ფუნქციები

4.1 Function calls
4.1. ფუნქციის გამოძახება

პროგრამირების ამ კონტექსტში, ფუნქცია არის სახელდარქმეული ბრძანებათა რიგი, რომელიც ასრულებს გამოთვლას. როცა ფუნქციას განსაზღვრავ, მიუთითებ სახელს და ბრძანებათა რიგს. მოგვიანებით შეგიძლია "დაუძახო" ფუნქციას სახელით. ფუნქციის ერთი მაგალითი უკვე ვნახეთ:

```
>>> type(32)
<type 'int'>
```

ფუნქციის სახელია **type**. გამოსახულებას ფრჩხილებში ჰქვია ფუნქციის არგუმენტი. არგუმენტი არის მნიშვნელობა ან ცვლადი, რომელიც შეგვაქვს ფუნქციაში . **type**ფუნქციის შედეგია არგუმენტის ტიპი. შეიძლება ვთქვათ, რომ ფუნქცია "იღებს" არგუმენტს და "აბრუნებს" შედეგს. ამ შედეგს ეძახიან დაბრუნებულ მნიშვნელობას(return value).

4.2 ჩაშენებული ფუნქციები

პითონი უზრუნველყოფს გარკვეული რაოდენობის მნიშვნელოვან ჩაშენებულ ფუნქციებს,რომელიც შეგვიძლია გამოვიყენოთ ისე, რომ არ უზრუნველვყოთ ფუნქციის განსაზღვრება. პითონის შემქმნელებმა საერთო პრობლემების გადასაჭრელად დაწერეს ფუნქციების კომპლექტი და ჩააშენეს პითონში .

max და **min** ფუნქციები გვაძლევს უდიდეს და უმცირეს მნიშვნელობებს სიაში. შესაბამისად :

```
>>> max('Hello world')
'w'
>>> min('Hello world')
' '
>>>
```

max ფუნქცია გვეუბნება რომელია "უდიდესი ასო" სტრინგში (რომელიც თურმე არის ასო `'w'`) და **min** ფუნქცია გვეუბნება რომელია 'უმცირესი ასო' - რომელიც თურმე არის ცარიელი ადგილი (' ').

სხვა ყველაზე ჩვეულებრივი ჩაშენებული ფუნქციაა **len** ფუნქცია, რომელიც გვეუბნება რამდენი ელემენტია არგუმენტში. არგუმენტი თუ სტრინგია, მაშინ გვაჩვენებს რამდენი ასოა სტრინგში.

```
>>> len('Hello world')
11
>>>
```

ფუნქციები არაა ლიმიტირებული და შეუძლიათ იმუშაონ ნებისმიერი ტიპის მნიშვნელობასთან - რასაც ვნახავთ შემდეგ თავებში.

ჩაშენებული ფუნქციის სახელები განიხილე როგორც განსაზღვრული სიტყვები(ე.ი. ცვლადს არ დაარქვა სახლი "`max`")

4.3 Type conversion functions
4.3. ტიპის გარდაქმნის ფუნქციები

პითონი ასევე უზრუნველყოფილია ჩაშენებული ფუნქციებით, რომელიც აკონვერტებს მნიშვნელობის ერთ ტიპს მეორე ტიპად. **int** ფუნქცია იღებს ნებისმიერ მნიშვნელობას და აკონვერტებს მთელ რიცხვად - თუ შეუძლია. თუ არა და წუწუნებს:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int -ს შეუძლია გადაიყვანოს წილადი მთელ რიცხვში, თუმცა რიცხვს არ ამრგვალებს, უბრალოდ წილადის ნიშანს აცლის:

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float აკონვერტებს მთელ რიცხვს და სტრინგს წილადად:

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

str თავის არგუმენტს აკონვერტებს სტრინგად:

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

4.6 ახალი ფუნქციების დამატება

ამდენი ხნის განმავლობაში ვიყენებდით ფუნქციებს რომელიც ჩაშენებულია პითონში, თუმცა შესაძლებელია ახალი ფუნქციების დამატებაც. ფუნქციის განსაზღვრება (function definition) მოითხოვს სპეციალურ სახელს ახალი ფუნქციისთვის და ბრძანებათა რიგს,რომელიც გამოძახებისას განახორციელებს ამ ფუნქციას. რაკი ერთხელ გავაკეთებთ ფუნქციას, შეგვეძლება გამოვიყენოთ მთელ ჩვენ პროგრამაში.
აქ არის მაგალითი :

```
def print_lyrics():
      print ("I'm a lumberjack, and I'm okay.")
      print ('I sleep all night and I work all day.')
```

def არის განსაზღვრული სიტყვა რომელიც მიანიშნებს რომ ესაა ფუნქციის განსაზღვრება (function definition). ფუნქციის სახელია `print_lyrics`. წესები ფუნქციის სახელისთვის არის იგივე, რაც ცვლადების სახელისთვის: ასოები, ციფრები და ზოგიერთი სასვენი ნიშანი დაშვებულია, მაგრამ სახელის პირველი სიმბოლო არ შეიძლება იყოს ციფრი. არ შეგიძლია გამოიყენო განსაზღვრული სიტყვა როგორც ფუნქციის სახელი. არ უნდა გქონდეს ცვლადი და ფუნქცია ერთი და იგივე სახელით.
სახელის შემდეგ ცარიელი ფრჩხილები მიანიშნებს, რომ ფუნქცია არ იღებს არგუმენტებს. მოგვიანებით გავაკეთებთ ფუნქციას, რომელიც იღებს არგუმენტს, როგორც input - ს.
ფუნქციის განსაზღვრების პირველ ხაზს ეწოდება სათაური (`header`); დანარჩენს - სხეული (`body`). სათაური უნდა დამთავრდეს ორწერტილით და სხეული უნდა იყოს დაკბილული. დაკბილვა ხდება 4 ჰარით (space). სხეული შეიძლება შეიცავდეს განუსაზღვრელი რაოდენობის ბრძანებებს. ამობეჭდვის ბრძანებაში (**`print`**) სტრინგი ჩასმულია ბრჭყალებში. ერთმაგი და ორმაგი ბრჭყალები ერთი და იგივე დანიშნულებისაა. უმეტესობა იყენებს ერთმაგ ბრჭყალებს, გარდა შემთხვევებისა, სადაც ერთმაგი ბრჭყალი გამოიყენება სტრინგში შიგნით. ფუნქციის განსაზღვრებას დიალოგურ რეჟიმში თუ დაწერ, მთარგმნელი დაბეჭდავს ელიფსებს (**...**) რათა გაგაგებინოს რომ განსაზღვრება არაა დასრულებული.

```
>>> def print_lyrics():
...   print ("I'm a lumberjack, and I'm okay.")
...   print ('I sleep all night and I work all day.')
...
```

ფუნქციის დასამთავრებლად უნდა დაბეჭდო ერთი ცარიელი ხაზი. (სკრიპტში ამის გაკეთება არაა აუცილებელი).
ფუნქციის გაკეთება ასევე აკეთებს ცვლადს იგივე სახელით.

```
>>> print print_lyrics
<function print_lyrics at 0xb7e99e9c>
>>> print type(print_lyrics)
<type 'function'>
```

`print_lyrics` - ის მნიშვნელობა არის ფუნქციის ობიექტი რასაც აქვს ტიპი `'function'`.
ფუნქციის გამოძახების სინტაქსი არის იგივე რაც ჩაშენებული ფუნქციის :

```
>>> print_lyrics()
      I'm a lumberjack, and I'm okay.
      I sleep all night and I work all day.
```

როცა გააკეთებ ფუნქციას, შეგიძლია გამოიყენო სხვა ფუნქციებში. მაგალითად, წინა ფუნქცია რომ გამოვიყენოთ, უნდა დავწეროთ ახალი ფუნქცია სახელად `repeat_lyrics`:

```
def repeat_lyrics():
      print_lyrics()
      print_lyrics()
```

და შემდეგ გამოვიძახოთ repeat_lyrics:

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

4.7 განსაზღვრა და გამოყენება

წინა ორივე კოდის ფრაგმენტების გაერთიანებით მთლიანი პროგრამა გამოიყურება ასე :

```
def print_lyrics():
      print ("I'm a lumberjack, and I'm okay.")
      print ('I sleep all night and I work all day.')

def repeat_lyrics():
      print_lyrics()
      print_lyrics()

repeat_lyrics()
```

პროგრამა შეიცავს ორ ფუნქციას: print_lyrics და repeat_lyrics. ფუნქციები ხორციელდება ისე, როგორც სხვა ბრძანებები, მაგრამ ეფექტი არის რომ შექმნას ფუნქციური ობიექტები. ფუნქციის შიგნით ბრძანებები არ ხორციელდება, სანამ ფუნქცია არის გამოძახებული და ფუნქციის განსაზღვრებას არ აქვს output.
როგორც ელოდებოდი, ჯერ უნდა გააკეთო ფუნქცია სანამ შეგეძლება მისი განხორციელება. სხვა სიტყვებით: **function definition has to be executed before the first time it is called.**

სავარჯიშო 4.2 პროგრამის ბოლო ხაზი აიტანე სულ ზემოთ, ასე რომ ფუნქციას გამოიძახებ განსაზღვრებამდე. გაუშვი პროგრამა და ნახე რა შეცდომა მოხდება.

სავარჯიშო 4.3 ფუნქციის გამოძახება დააბრუნე თავის ადგილზე და print_lyricsგანსაზღვრება გადაიტანე repeat_lyrics - ის შემდეგ. რა მოხდება როცაპროგრამას გაუშვებ?

4.9 პარამეტრები და არგუმენტები

ზოგიერთ ჩაშენებულ ფუნქციას (რომელიც ვნახეთ) ესაჭიროება არგუმენტები. მაგალითად; როცა იძახებ **math.sin**, შეგყავს ციფრი როგორც არგუმენტი. ზოგი ფუნქცია იღებს ერთზე მეტ არგუმენტს. **math.pow** იღებს ორს - ფუძეს და ხარისხის მაჩვენებელს. ფუნქციის შიგნით ეს არგუმენტები გატოლებულია ცვლადებს, რომელსაც ეძახიან პარამეტრებს. აქაა მაგალითი მომხმარებლის გაკეთებული ფუნქციის რომელსაც აქვს არგუმენტი:

```
def print_twice(bruce):
      print (bruce)
      print (bruce)
```

ეს ფუნქცია არგუმენტს უტოლებს ცვლადს სახელად `bruce`. როცა ფუნქცია გამოძახებულია, ბეჭდავს პარამეტრის (რაც არ უნდა იყოს)მნიშვნელობას ორჯერ.
ეს ფუნქცია მუშაობს ყველა მნიშვნელობასთან რაც შეიძლება რომ დაიბეჭდოს.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(17)
17
17
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

ჩაშენებულ და მომხმარებლის გაკეთებულ ფუნქციებში იგივე წესებია, ასე რომ შეგიძლია გამოვიყენოთ გამოსახულება როგორც არგუმენტი:

```
print_twice:
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

არგუმენტი შეფასებულია ფუნქციის გამოძახებამდე, ამ მაგალითში `'Spam '* 4` და`math.cos(math.pi)` გამოანგარიშებულია მხოლოდ ერთხელ.

ასევე შეგიძლია გამოიყენო ცვლადი როგორც არგუმენტი

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

ცვლადის სახელს, რომელიც არგუმენტის მაგივრად შევიტანეთ `(michael)` არაფერი აქვს საერთო პარამეტრის სახელთან (`bruce`). არ აქვს აზრი თუ რა იყო ადრე მნიშვნელობა; აქ `print_twice` ფუნქციაში ყველაფერს ვეძახით `bruce` -ს .

4.10 ნაყოფიერი ფუნქციები და გაბათილებული ფუნქციები

ზოგიერთი ფუნქცია რომელსაც ვიყენებთ, როგორიცაა მათ. ფუნქცია აწარმოებს შედეგებს. ასეთ ფუნქციებს ვეძახით ნაყოფიერ ფუნქციას (**fruitful functions**). სხვა ფუნქციები, მაგალითად `print_twice`, რაღაცას აკეთებს, მაგრამ არ აბრუნებს მნიშვნელობას. ამათ ეძახიან გაბათილებულ ფუნქციებს (**void functions**.)
როცა იძახებ ნაყოფიერ ფუნქციას, შენ თითქმის ყოველთვის გინდა, რომ შედეგით რამე გააკეთო; მაგალითად შეიძლება გაუტოლო ცვლადს ან გამოიყენო როგორც გამოსახულების ნაწილი:

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

როცა ფუნქციას დიალოგურ რეჟიმში იძახებ, პითონს შედეგი ეკრანზე გამოაქვს

```
>>> math.sqrt(5)
2.2360679774997898
```

მაგრამ სკრიპტში, ნაყოფიერ ფუნქციას თუ გამოიძახებ და ფუნქციის შედეგს (return value,result)ცვლადში თუ არ შეინახავ, მნიშვნელობა გაქრება ნისლში!

```
math.sqrt(5)
```

ეს სკრიპტი ანგარიშობს კვადრატულ ფესვს ხუთიდან, მაგრამ სანამ შედეგს არ შეინახავს ცვლადში ან ეკრანზე არ გამოიტანს, მანამდე არც ისე გამოსადეგია.
გაბათილებულმა ფუნქციამ ეკრანზე შეიძლება რამე გამოიტანოს ან ქონდეს სხვა რამე ეფექტი, მაგრამ არ ექნება დაბრუნებული მნიშვნელობა (return value). თუ ცდი რომ შედეგი ცვლადს გაუტოლო, მიიღებ სპეციალურ მნიშვნელობას სახელად **`None`.**

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print (result)
None
```

მნიშვნელობა `None` არაა იგივე რაც სტრინგი "`None`", ამას აქვს სპეციალური მნიშვნელობა რომელსაც აქვს თავისი ტიპი:

```
>>> print type(None)
<type 'NoneType'>
```

ფუნქციაში შედეგი რომ დააბრუნო , ფუნქციაში ვიყენებთ **`return`** ბრძანებას. მაგალითად, შეგვიძლია გავაკეთოთ მარტივი ფუნქცია სახელად `addtwo` , რომელიც ორ ციფრს ერთმანეთს მიუმატებს და დააბრუნებს შედეგს:

```
def addtwo(a, b):
      added = a + b
      return added

x = addtwo(3, 5)
print (x)
```

ეს სკრიპტი როცა განხორციელდება, **`print`** ბრძანება ამობეჭდავს 8 - ს, იმიტომ რომ`addtwo` ფუნქცია გამოძახებული იყო არგუმენტებით 3 და 5. ფუნქციის შიგნით

პარამეტრები `a` და `b` შესაბამისად იყო 3 და 5 . ფუნქციამ გამოთვალა ჯამი ორი ციფრის და მოათავსა ლოკალურ ფუნქციის ცვლადში სახელად `added` და გამოიყენა ბრძანება `return`რომ დათვლილი მნიშვნელობა გაეგზავნა **უკან გამომახების კოდში (back to the calling code)** როგორც ფუნქციის შედეგი რომელიც მიენიჭა ცვლადს `x` და ამოიბეჭდა.

4.11 რატომ ფუნქციები?

ეს შეიძლება არ იყოს ნათელი, თუ რატომაა ღირებული პროგრამის დაყოფა ფუნქციებში. აქაა რამდენიმე მიზეზი:

- ახალი ფუნქციის გაკეთება გაძლევს შესაძლებლობას რომ სახელი დაარქვა ბრძანებათა ჯგუფს, რომელიც შენს პროგრამას გახდის უფრო ადვილად წასაკითხს, გასაგებს და გამართულს.
- ფუნქციებს შეუძლია განმეორებადი კოდის აღკვეთით პროგრამა გახადოს უფრო პატარა. მოგვიანებით თუ ცვლილებას გააკეთებ, ამის გაკეთება დაგჭირდება ერთ ადგილას.
- დიდი პროგრამის ფუნქციებად დაყოფა გაძლევთ საშუალებას რომ გამართოთ ნაწილები სათითაოდ და შემდეგ შეკრათ ერთ მთლიანად.
- კარგად გაკეთებული ფუნქციები ხშირად სასარგებლოა ბევრი პროგრამისთვის. რაკი ერთხელ დაწერ და გამართავ, შეგიძლია ბევრჯერ გამოიყენო.

წიგნის დანარჩენ ნაწილში ხშირად გამოვიყენებთ ფუნქციის განსაზღვრებას როგორც მაგალითს. ცოდნის ნაწილი ფუნქციის შექმნის და გამოყენების ისაა, რომ ფუნქცია ზუსტად გამოსახავდეს აზრს; როგორიცაა "იპოვნე ყველაზე პატარა მნიშვნელობა მნიშვნელობათა სიაში". მოგვიანებით გაჩვენებთ კოდს, რომელიც პოულობს ყველაზე პატარას მნიშვნელობათა სიაში და წარმოგიდგენთ მას, როგორც ფუნქციას სახელად **`min`**,რომელიც იღებს მნიშვნელობათა სიას როგორც არგუმენტს და აბრუნებს უმცირეს მნიშვნელობას ამ სიიდან.

4.13 ლექსიკონი

algorithm: პრობლემების კატეგორიის გადასაჭრელი ზოგადი პროცესი.
argument: მნიშვნელობა, რომელიც უზრუნველყოფს ფუნქციას, როცა ფუნქცია გამოძახებულია. ეს მნიშვნელობა ენიჭება შესაბამის პარამეტრს ფუნქციაში
body: ბრძანებათა თანრიგი ფუნქციის განსაზღვრებაში.
composition: გამოსახულების გამოყენება როგორც უფრო დიდი გამოსახულების ნაწილის, ან ანგარიშის (ბრძანების - statement.) გამოყენება როგორც უფრო დიდი ანგარიშის (ბრძანების - statement.) ნაწილის.
deterministic: ეკუთვნის პროგრამას, რომელიც ერთი და იგივე შენატანის (input) შემთხვევაში აკეთებს იგივეს ყოველ გაშვებაზე.
dot notation: მოდულში ფუნქციის გამოსაძახებელი სინტაქსი ზუსტად განსაზღვრული მოდულის სახელით, რომელსაც მოყვება წერტილი და ფუნქციის სახელი.
flow of execution: თანრიგი, რომლის მიხედვითაც ანგარიშები ხორციელდება პროგრამის მუშაობის დროს.
fruitful function: ფუნქცია, რომელიც აბრუნებს მნიშვნელობას.
function: სახელდარქმეული ბრძანებათა რიგი, რომელიც ასრულებს სასარგებლო ოპერაციას. ფუნქციებს შეიძლება ჰქონდეს ან არ ჰქონდეს არგუმენტები და შეიძლება აწარმოებდეს ან არ აწარმოებდეს შედეგს.
function call: ბრძანება, რომლითაც ფუნქცია ხორციელდება. ის შედგება ფუნქციის სახელისა და არგუმენტების სიისგან.
function definition: ბრძანება, რომელიც აკეთებს ახალ ფუნქციას, უსაზღვრავს სახელს, პარამეტრებს და ბრძანებას რომელსაც ახორციელებს.
function object: მნიშვნელობა, რომელიც გააკეთა ფუნქციის განსაზღვრებამ. ფუნქციის სახელი არის ცვლადი რომელიც გზავნის ფუნქციის ობიექტს .
header: ფუნქციის განსაზღვრების პირველი ხაზი.

import statement: ბრძანება, რომელიც კითხულობს მოდულის ფაილს და აკეთებს მოდულის ობიექტს,
module object: მნიშვნელობა, რომელიც გააკეთა `import` ბრძანებამ, რაც უზრუნველყოფს წვდომას მონაცემებთან და კოდთან, რაც განსაზღვრულია მოდულში.
parameter: სახელი, რომელიც გამოიყენება ფუნქციის შიგნით და გზავნის მნიშვნელობას როგორც არგუმენტს.
pseudorandom: ეკუთვნის ციფრების თანრიგს, რომელიც ჩანს როგორც შემთხვევითი, მაგრამ არის გენერირებული დეტერმინირებული პროგრამით.
return value: ფუნქციის შედეგი. გამოძახებული ფუნქცია თუ გამოყენებულია როგორც გამოსახულება, დაბრუნებული მნიშვნელობა არის გამოსახულების მნიშვნელობა.
void function: ფუნქცია რომელიც არ აბრუნებს შედეგს.