

# **Lekcija 7 - Osnove Python programiranja**

# Contents

<b>LearningObject.....</b>	<b>3</b>
L7: Osnove Python programiranja.....	3
Python funkcije.....	3
Definisanje funkcije.....	4
Poziv funkcije.....	5
Prenos po referenci.....	7
Argumenti funkcije.....	9
Argumenti ključnih reci.....	11
Zadati argumenti.....	13
Argumenti promjenjive dužine.....	15
Anonimne funkcije.....	16
Izjava return.....	17
Opseg varijabli.....	19
Raspakivanje argumenata liste.....	21
Rekurzivne funkcije.....	23
Izuzetci.....	26
Upravljanje izuzetcima.....	31
Aktiviranje izuzetka.....	37
Izuzetci programera.....	38
L7: Osnove Python programiranja.....	41
 <b>LearningObject.....</b>	 <b>42</b>
Python - kontrolne strukture, funkcije, strukture podataka.....	42
Python - kontrolne strukture, greške, rekurzija.....	48

# LearningObject

---

## L7: Osnove Python programiranja

---

### Uvod \*

Ovo poglavlje predstavlja uvod u osnove Python programiranja.

## Python funkcije

---

### Cilj

- Upoznavanje sa Python funkcijama

### Python funkcije

Python funkcije

Python ima mnoge ugrađene funkcije (engl. built-in functions) kao što su `print()` itd. i koje programer može koristiti, ali mu se daje mogućnost da kreira svoje funkcije. Ove funkcije se zovu korisnički definisane funkcije (engl. user-defined functions).

#### Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like `print()`, etc. but you can also create your own functions. These functions are called *user-defined functions*.

Function blocks begin with the keyword `def` followed by the function name and parentheses `( )`.

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.

The code block within every function starts with a colon `:` and is indented.

The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

#### Syntax

```
def functionname( parameters ):
```

```
"function_docstring"
```

```
function_suite
return [expression]
```

## Definisanje funkcije

---

### Cilj

- Upoznavanje sa Python funkcijama

### Definisanje Python funkcije

Definisanje Python funkcije

Programer može da definiše funkcije koje pružaju određenu funkcionalnost. Evo nekih pravila za definisanje Python funkcija:

Blok funkcije počinje s ključnom riječju `def`, slijedi ime funkcije i zagrade `(( ))`.

Bilo koji ulazni parametri ili argumenti trebaju biti smješteni u okviru tih zagrada. Takođe se mogu definisati parametri u okviru tih zagrada.

Blok koda u okviru svake funkcije počinje s dvotačkom `(:)`.

Zahtjev za izvršavanje funkcije se zove poziv funkcije (engl. function call). Kada se funkcije pozove, može primiti argumente koji definišu tip podataka koje funkcija koristi. Python funkcija uvijek vraća vrijednost koja može biti `None` ili vrijednost koja predstavlja rezultat nekog proračuna izvršenog u samoj funkciji.

Python funkcije su objekti i na njih se primjenjuju ista pravila kao i za sve druge objekte. Neka Python funkcija se može proslijediti kao argument u pozivu neke druge funkcije. Takođe, Python funkcija može da kao kao izlaz vrati drugu Python funkciju

Slika 1 daje izjave i izrazi vezani za funkcije.

Statement	Examples
Calls	<code>myfunc('spam', 'eggs', meat=ham)</code>
<code>def</code> , <code>return</code>	<code>def adder(a, b=1, *c):     return a + b + c[0]</code>
<code>global</code>	<code>def changer():     global x; x = 'new'</code>
<code>nonlocal</code>	<code>def changer():     nonlocal x; x = 'new'</code>
<code>yield</code>	<code>def squares(x):     for i in range(x): yield i ** 2</code>
<code>lambda</code>	<code>funcs = [lambda x: x**2, lambda x: x*3]</code>

Slika 1 Izjave i izrazi vezani za funkcije

`def` izjava

Def izjava definiše object funkcije i daje mu ime. Opšti format:

```
def <name>(arg1, arg2,... argN):
```

```
<statements>
```

Funkcija često sadrži i povratnu izjavu:

```
def <name>(arg1, arg2,... argN):
```

```
...
```

```
return <value>
```

Primer

```
def printme( str ):
```

```
"This prints a passed string into this function"
```

```
print (str)
```

```
return
```

Dodatak

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

Function blocks begin with the keyword `def` followed by the function name and parentheses `( )`.

Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.

The first statement of a function can be an optional statement - the documentation string of the function or docstring.

Docstring

*Unlike conventional source code comments, or even specifically formatted comments like Javadoc documentation, docstrings are not stripped from the source tree when it is parsed, but are retained throughout the runtime of the program. This allows the program to inspect these comments at run time, for instance as an interactive help system, or as metadata.*

The code block within every function starts with a colon `(:)` and is indented.

The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

## Poziv funkcije

---

### Cilj

- Osnove Python programiranja

**Poziv funkcije**

Poziv funkcije

```
#!/usr/bin/python

# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print (str);
    return;

# Now you can call printme function
printme("I'm first call to user defined function!");
printme("Again second call to the same function");
```

Rezultat

I'm first call to user defined function!  
Again second call to the same function

Primer

```
>>> def times(x, y): # Create and assign function
... return x * y # Body executed when called

>>> times(2, 4) # Arguments in parentheses
8
```

Primer

```
>>> x = times(3.14, 4) # Save the result object
>>> x
12.56
```

Primer

```
>>> times('Ni', 4) # Functions are "typeless"
'NiNiNiNi'
```

Primer

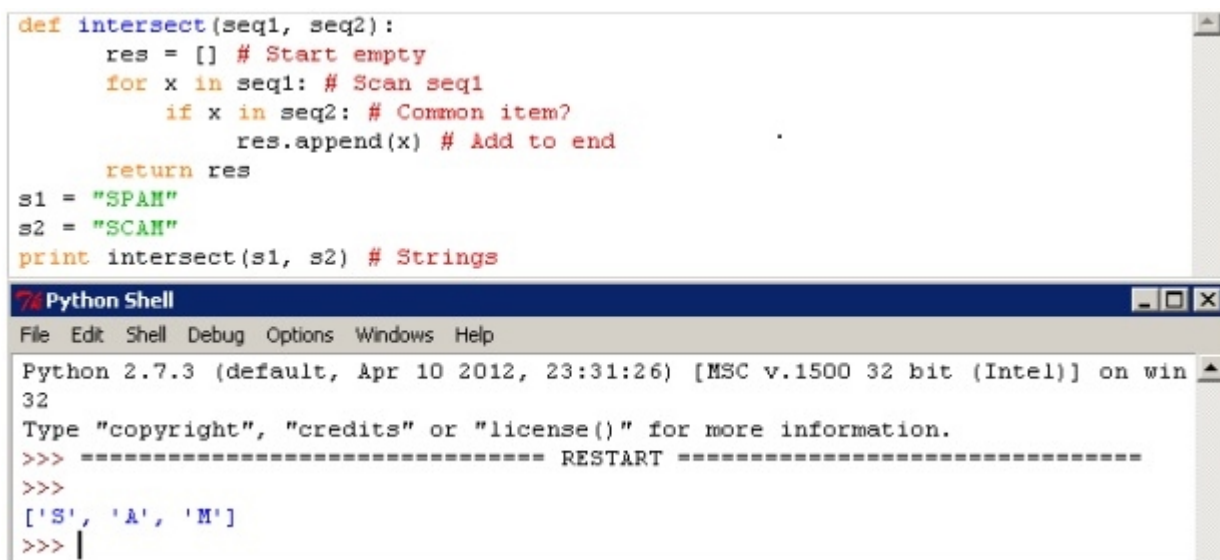
```
def f(a, L=[]):
    L.append(a)
    return L

print f(1)
print f(2)
print f(3)
```

Primer

```
def intersect(seq1, seq2):
    res = [] # Start empty
    for x in seq1: # Scan seq1
        if x in seq2: # Common item?
            res.append(x) # Add to end
    return res
s1 = "SPAM"
s2 = "SCAM"
print intersect(s1, s2) # Strings
```

Slika 1 daje rezultat izvršavanja predhodnog programa.



```
def intersect(seq1, seq2):
    res = [] # Start empty
    for x in seq1: # Scan seq1
        if x in seq2: # Common item?
            res.append(x) # Add to end
    return res
s1 = "SPAM"
s2 = "SCAM"
print intersect(s1, s2) # Strings
```

Python Shell

File Edit Shell Debug Options Windows Help

Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more information.

>>> ===== RESTART =====

>>>

['S', 'A', 'M']

>>> |

Slika 1 Primjer izvršavanja Python programa

## Prenos po referenci

---

### Cilj

- Upoznavanje metoda prenosa po referenci

### Prenos po referenci

Prenos po referenci (engl. pass by reference)

Svi parametri (argumenti) u Python jeziku se prenose po reference. To znači da ako se promijeni ono na šta se parametar odnosi u okviru funkcije, promjena se odražava nazad u pozivnu funkciju.

Primer

```
# Function definition is here
def changeme( mylist ):
```

"This changes a passed list into this function"

```
mylist.append([1,2,3,4]);
```

```
print "Values inside the function: ", mylist
```

```
return
```

# Now you can call changeme function

```
mylist = [10,20,30];
```

```
changeme( mylist );
```

```
print "Values outside the function: ", mylist
```

Ovdje se održava referenca prenešenog objekta i vrši dodavanje vrijednosti istom objektu. Rezultat je slijedeći:

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

U sledećem primeru argument se prenosi po referenci, ali u okviru funkcije tako da je referenca uništena.

Primer

# Function definition is here

```
def changeme( mylist ):
```

"This changes a passed list into this function"

```
mylist = [1,2,3,4]; # This would assign new reference in mylist
```

```
print "Values inside the function: ", mylist
```

```
return
```

# Now you can call changeme function

```
mylist = [10,20,30];
```

```
changeme( mylist );
```

```
print "Values outside the function: ", mylist
```

Parameter mylist je local za funkciju changeme. Promjenom mylist u okviru funkcije ne mijenja.

Funkcija ne izvršava ništa i rezultat je slijedeći:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

Dodatak

Pass by reference vs value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
#!/usr/bin/python
```

# Function definition is here

```
def changeme( mylist ):
```



"This changes a passed list into this function"

```
mylist.append([1,2,3,4]);
print "Values inside the function: ", mylist
return
```

# Now you can call changeme function

```
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result:

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

```
#!/usr/bin/python
```

# Function definition is here

```
def changeme( mylist ):
    "This changes a passed list into this function"
    mylist = [1,2,3,4]; # This would assign new reference in mylist
    print "Values inside the function: ", mylist
    return
```

# Now you can call changeme function

```
mylist = [10,20,30];
changeme( mylist );
print "Values outside the function: ", mylist
```

The parameter mylist is local to the function changeme. Changing mylist within the function does not affect mylist. The function accomplishes nothing and finally this would produce the following result:

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

## Argumenti funkcije

---

### Cilj

- Upoznavanje sa argumentima Python funkcije

## Argumenti funkcije

Argumenti funkcije

Funkcija se može pozvati korišćenjem slijedećih tipova formalnih argumenata:

Potrebni argumenti (engl. required arguments)

Argumenti ključnih reči (engl. keyword arguments)

Zadani argumenti (engl. default arguments)

Argumenti promjenjive dužine (engl. variable-length arguments)

Potrebni argumenti

Potrebni argumenti su argumenti koji se prenose funkciji u pravilnom pozicionom redu. Ovdje bi broj argumenata u pozivu funkciji trebao da bude jednak broju argumenata u definiciji funkcije

Za pozivanje funkcije `printme()` svakako je potrebno da se prenese jedan argument inače bi to dovelo do sintaktičke greške:

Primjer

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str;
    return;
# Now you can call printme function
printme();
```

Kada se gornji kod izvrši, dobija se slijedeći rezultat:

Traceback (most recent call last):

File "test.py", line 11, in <module>

`printme();`

`TypeError: printme() takes exactly 1 argument (0 given)`

Dodatak

Function Arguments

You can call a function by using the following types of formal arguments:

Required arguments

Keyword arguments

Default arguments

Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it would give a syntax error as follows:

```
#!/usr/bin/python

# Function definition is here

def printme( str ):
    "This prints a passed string into this function"
    print str;
    return;

# Now you can call printme function
printme();
```

When the above code is executed, it produces the following result:

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

## Argumenti ključnih reci

---

### Cilj

- Upoznavanje sa argumentima ključnih reci

### Argumenti ključnih reci

Argumenti ključnih reci

Funkcije se takođe mogu pozvati korišćenjem argumenata ključnih riječi kao kwarg=value. Na primjer, slijedeća funkcija:

```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

prihvata jedan traženi argument (voltage) i tri opciona argumenta. Ova se funkcija poziva na jednad od slijedecih nacina:

```
parrot(1000) # 1 positional argument
```

```
parrot(voltage=1000) # 1 keyword argument
```

```
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
```

```
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments
```

```
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
```

```
parrot('a thousand',state='pushing up the daisies')# 1 positional, 1 keyword
```

Argumenti ključnih riječi se odnose na pozive funkcija. Kada se koriste argumenti ključnih riječi u pozivu funkcije, pozivač identifikuje argumente po imenu parametra.

To omogućuje preskakanje argumenata ili njihovo stavljanje izvan reda, jer je Python interpreter u mogućnosti da koristi ključne riječi pod uslovom da odgovaraju vrijednostima parametara. Takođe, možete napraviti pozive ključnih riječi za `printme()` funkciju u slijedeći način:

Primer

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str;
    return;
# Now you can call printme function
printme( str = "My string");
```

Kada se gornji kod izvrši, dobija se slijedeći rezultat:

My string

Slijedeći primjer daje jasniju sliku. Zapaziti da ovdje redoslijed parametara nije važan:

Primer

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age ):
    "This prints a passed info into this function"
    print "Name: ", name;
    print "Age ", age;
    return;
# Now you can call printinfo function
printinfo( age=50, name="miki" );
```

Kada se gornji kod izvrši, dobija se slijedeći rezultat:

Name: miki

Age 50

Dodatak

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the `printme()` function in the following ways:

```
#!/usr/bin/python
# Function definition is here
def printme( str ):
    "This prints a passed string into this function"
    print str;
    return;

# Now you can call printme function
printme( str = "My string");
```

When the above code is executed, it produces the following result:

My string

Following example gives more clear picture. Note, here order of the parameter does not matter.

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age ):
    "This prints a passed info into this function"
    print "Name: ", name;
    print "Age ", age;
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
```

When the above code is executed, it produces the following result:

Name: miki

Age 50

## Zadati argumenti

---

### Cilj

- Zadati argumenti

### Zadati argumenti

Zadani argument (engl. default argument) je argument koji pretpostavlja zadanu vrijednost ako vrijednost za taj argument nije obezbjeđena u pozivu funkcije. Naredni primjer daje ideju o zadanim argumentima, program bi napisao zadanu vrijednost godina, ako vrijednost nije proslijeđena:

### Primjer

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print "Name: ", name;
    print "Age ", age;
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

Kada se gornji kod izvrši, dobija se slijedeći rezultat:

```
Name: miki
Age 50
Name: miki
Age 35
```

## **Dodatak**

### **Default arguments**

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. Following example gives an idea on default arguments, it would print default age if it is not passed:

```
#!/usr/bin/python
# Function definition is here
def printinfo( name, age = 35 ):
    "This prints a passed info into this function"
    print "Name: ", name;
    print "Age ", age;
    return;

# Now you can call printinfo function
printinfo( age=50, name="miki" );
printinfo( name="miki" );
```

When the above code is executed, it produces the following result:

```
Name: miki
Age 50
Name: miki
Age 35
```

## Argumenti promjenjive dužine

---

### Cilj

- Osnove Python programiranja

### Argumenti promjenjive dužine

Možda će postojati potreba za više argumenata koje funkciju treba, ali koji nisu navedeni pri definisanju funkcije. Ovi argumenti se nazivaju argumenti promjenljive dužine i oni nisu imenovani u definiciji funkciji, za razliku od potrebnih i zadanih argumenata.

Opšta sintaksa za funkciju sa argumenatima sa ne-ključnim varijablama:

```
def functionname([formal_args,] *var_args_tuple):
```

```
...
```

```
...
```

```
return [expression]
```

Zvezdica (\*) se stavlja ispred imena varijable koja će se držati vrijednosti svih ne-ključnih varijabli argumenata. Ovo torka ostaje prazna, ako se nikakvi dodatni argumenti nisu navedeni u funkcijskom pozivu. Slijedi jednostavan primjer:

### Primer

```
#!/usr/bin/python
```

```
# Function definition is here
```

```
def printinfo( arg1, *vartuple ):
```

```
"This prints a variable passed arguments"
```

```
print "Output is: "
```

```
print arg1
```

```
for var in vartuple:
```

```
print var
```

```
return;
```

```
# Now you can call printinfo function
```

```
printinfo( 10 );
```

```
printinfo( 70, 60, 50 );
```

Kada se gornji kod izvrši, dobija se sledeći rezultat:

Output is:

10

Output is:

70

60

50

**Dodatak****Variable-length arguments**

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

The general syntax for a function with non-keyword variable arguments is this:

```
def functionname([formal_args,] *var_args_tuple ):
    "function_docstring"
    function_suite
    return [expression]
```

An asterisk (\*) is placed before the variable name that will hold the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example:

```
#!/usr/bin/python
# Function definition is here
def printinfo( arg1, *vartuple ):
    "This prints a variable passed arguments"
    print "Output is: "
    print arg1
    for var in vartuple:
        print var
    return;

# Now you can call printinfo function
printinfo( 10 );
printinfo( 70, 60, 50 );
```

When the above code is executed, it produces the following result:

Output is:

10

Output is:

70

60

50

## Anonimne funkcije

---



**Cilj**

- Upoznavanje sa anonimnim Python funkcijama

**Anonimne funkcije**

Možete koristiti lambda ključnu riječ za stvaranje malih anonimnih funkcija. Ove funkcije se nazivaju anonimne jer nisu definisane na standardan način pomoću **def** ključnu riječi.

Lambda forme mogu uzeti bilo koji broj argumenata, ali vraćaju samo jednu vrijednost u obliku izraza. Oni ne mogu sadržavati naredbu ili više izraza.

Anonimna funkcija ne može biti direktno pozvana za štampanje, jer lambda zahtijeva izraz.

Lambda funkcije imaju svoj lokalni prostor imena (engl. namespace) i mogu samo pristupiti varijablama koje su definisane u njihovim listama parametara i onim u globalnom prostoru imena.

**Sintaksa**

Sintaksa lambda funkcija sadrži samo jednu izjavu, koja glasi:

lambda [ARG1 [, ARG2, ..... argn]]: izraz

Slijedeći primjer pokazuje kako radi lambda oblik funkcije:

**Primer**

```
#!/usr/bin/python
```

```
# Function definition is here
```

```
sum = lambda arg1, arg2: arg1 + arg2;
```

```
# Now you can call sum as a function
```

```
print "Value of total : ", sum( 10, 20 )
```

```
print "Value of total : ", sum( 20, 20 )
```

Kad se izvrši predhodni kod, dobija se sledeći rezultat:

```
Value of total : 30
```

```
Value of total : 40
```

## Izjava return

---

**Cilj**

- Izjava return

**Izjava return**

Izjava **return** [izraz] omogućava izlazak iz funkcije, opcionalno slanje natrag izraza pozivaču. Povratak izjava bez argumenata je isti kao i povratak None.

Svi navedeni primjeri ne vraćaju nikakve vrijednosti, ali ako se želi, možete vratiti vrijednost iz funkcije kako sledi:

**Primer**

```
#!/usr/bin/python
```

```
# Function definition is here
```

```
def sum( arg1, arg2 ):
# Add both the parameters and return them."
total = arg1 + arg2
print "Inside the function : ", total
return total;
# Now you can call sum function
total = sum( 10, 20 );
print "Outside the function : ", total
```

Kad se izvrši predhodni kod, dobija se slijedeći rezultat:

Inside the function : 30

Outside the function : 30

Jednostavna funkcija koja vraća listu brojeva Fibonacci serije:

### **Primer**

```
>>> def fib2(n): # return Fibonacci series up to n
... """Return a list containing the Fibonacci series up to n."""
... result = []
... a, b = 0, 1
... while a < n:
... result.append(a) # see below
... a, b = b, a+b
... return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

### **Primer**

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
while True:
ok = raw_input(prompt)
if ok in ('y', 'ye', 'yes'):
return True
if ok in ('n', 'no', 'nop', 'nope'):
return False
retries = retries - 1
if retries < 0:
raise IOError('refusenik user')
print complaint
```

```
ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')
```

## **Dodatak**

### **The return statement**

The statement `return [expression]` exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as `return None`.

All the above examples are not returning any value, but if you like you can return a value from a function as follows:

```
#!/usr/bin/python

# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2
    print "Inside the function : ", total
    return total;

# Now you can call sum function

total = sum( 10, 20 );
print "Outside the function : ", total
```

When the above code is executed, it produces the following result:

```
Inside the function : 30
```

```
Outside the function : 30
```

## **Opseg varijabli**

---

### **Cilj**

- Upoznavanje sa Perl opsegom varijabli

### **Opseg varijabli**

Sve varijable u programu ne mogu biti dostupne na svim lokacijama u tom programu. To ovisi o tome gdje je varijabla definisana.

Opseg varijablu određuje dio programa u kojem možete pristupiti određenom identifikatoru. Postoje dva osnovna opsega varijabli u Pythonu:

globalne varijable

lokalne varijable

### **Globalne i lokalne varijable**

Varijable koje su definisane u tijelu funkcije imaju lokalni opseg, a one definisane van funkcije imaju globalni opseg (engl. global scope)..

To znači da se lokalnim varijablama može se pristupiti samo u funkciji u kojoj su definisane dok se sve funkcije mogu pristupiti globalnim varijablama u cijelom programu.

Kada se poziva funkcija, varijable definisane u unutra se su u opsegu. Slijedi jednostavan primjer:

### Primer

```
#!/usr/bin/python
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
    # Add both the parameters and return them."
    total = arg1 + arg2; # Here total is local variable.
    print "Inside the function local total : ", total
    return total;
# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

Kad se izvrši predhodni kod, dobija se slijedeći rezultat:

Inside the function local total : 30

Outside the function global total : 0

### Dodatak

#### Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

#### Global variables

#### Local variables

#### Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example:

```
#!/usr/bin/python
total = 0; # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
```

```
# Add both the parameters and return them."
total = arg1 + arg2; # Here total is local variable.
print "Inside the function local total : ", total
return total;
```

```
# Now you can call sum function
sum( 10, 20 );
print "Outside the function global total : ", total
```

When the above code is executed, it produces the following result:

```
Inside the function local total : 30
Outside the function global total : 0
```

## Raspakivanje argumenata liste

---

### Cilj

- Raspakivanje argumenata liste

### Raspakivanje argumenata liste

Obrnuta situacija nastaje kada su argumenti već na listi ili u torci (engl. tuple), ali trebaju biti raspakovani za poziv funkcije koja zahtijeva argumente koji su na različitim pozicijama.

Na primjer, Python ugrađena funkcija (eng. built-in) **range ()** funkcija očekuje start i stop argumente. Ako oni nisu dostupni odvojeno, napisati poziv funkcije sa \* operatorom raspakovati argumente iz liste ili torke:

```
>>>
>>> range(3, 6) # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> range (*args) # call with arguments unpacked from a list
[3, 4, 5]
```

Na sličan način, rječnici mogu dostaviti argumente ključne riječi \*\*operatorom:

```
>>>
>>> def parrot(voltage, state='a stiff', action='voom'):
... print "-- This parrot wouldn't", action,
... print "if you put", voltage, "volts through it.",
... print "E's", state, "!"
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

## Dodatak

The most basic data structure in Python is the **sequence**. Each element of a sequence is assigned a number - its position or index. The first index is zero, the second index is one, and so forth.

Python has six built-in types of sequences, but the most common ones are lists and tuples, which we would see in this tutorial.

There are certain things you can do with all sequence types. These operations include indexing, slicing, adding, multiplying, and checking for membership. In addition, Python has built-in functions for finding the length of a sequence and for finding its largest and smallest elements.

## Python Lists

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Good thing about a list is that items in a list need not all have the same type.

Creating a list is as simple as putting different comma-separated values between square brackets.

For example:

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5 ];
```

```
list3 = ["a", "b", "c", "d"];
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

## Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in range() function expects separate start and stop arguments. If they are not available separately, write the function call with the \*-operator to unpack the arguments out of a list or tuple:

```
>>>
```

```
>>> range(3, 6) # normal call with separate arguments
```

```
[3, 4, 5]
```

```
>>> args = [3, 6]
```

```
>>> range(*args) # call with arguments unpacked from a list
```

```
[3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the \*\*-operator:

```
>>>
```

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
```

```
... print "-- This parrot wouldn't", action,
```

```
... print "if you put", voltage, "volts through it.",
```

```
... print "E's", state, "!"
```

```
...
```

```
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

## Rekurzivne funkcije

---

### Cilj

- Upoznavanja sa Python rekurzivnim funkcijama

### Rekurzivne funkcije

Funkcije koje zovu sami sebe direktno ili indirektno obično u petlji.

### Sumiranje sa rekurzivnom funkcijom

Zadatak je sumirati niz brojeva u listi ili sekvenci. Može se koristiti ili ugrađena funkcija ili napisati vlastitu rekurzivnu funkciju.

### Primer

```
>>> def mysum(L):
... if not L:
... return 0
... else:
... return L[0] + mysum(L[1:]) # Call myself
>>> mysum([1, 2, 3, 4, 5])
15
```

Na svakom nivou, ova funkcija rekurzivno poziva samu sebe za računanje sume preostalih brojeva u listi koji će kasnije biti dodati predhodnoj elementu liste. Može se dodati print of L funkciji i izvršiti progma još jednom

### Primer

```
def mysum(L):
print(L) # Trace recursive levels
if not L: # L shorter at each level
return 0
else:
return L[0] + mysum(L[1:])

>>> mysum([1, 2, 3, 4, 5])
[1, 2, 3, 4, 5]
[2, 3, 4, 5]
[3, 4, 5]
[4, 5]
[5]
```

```
[]
```

```
15
```

### **Petlje i rekurzije**

Petlje izvršavaju posao automatski čineći rekurzije irelevantnim u većini slučajeva, a takođe manje efikasnim u pogledu memorijskog prostora i vremena izvršavanja.

#### **Primer: petlja**

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

```
>>> sum = 0
```

```
>>> while L:
```

```
... sum += L[0]
```

```
... L = L[1:]
```

```
>>> sum
```

```
15
```

#### **Primer: rekurzija**

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

```
>>> sum = 0
```

```
>>> for x in L: sum += x
```

```
...
```

```
>>> sum
```

```
15
```

### **Dodatak**

#### **Definition of Recursion**

Recursion is a way of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition fulfills the condition of recursion, we call this function a recursive function.

#### **Termination condition**

A recursive function has to terminate to be used in a program. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can lead to an infinite loop, if the base case is not met in the calls.

**Example**  $4! = 4 * 3! \ 3! = 3 * 2! \ 2! = 2 * 1$

Replacing the calculated values gives us the following expression  $4! = 4 * 3 * 2 * 1$  Generally we can say: Recursion in computer science is a method where the solution to a problem is based on solving smaller instances of the same problem.

#### **Recursive Functions in Python**

Now we come to implement the factorial in Python. It's as easy and elegant as the mathematical definition.

```
def factorial(n):
```

```
if n == 1:
```

```
    return 1
```



```
else:
    return n * factorial(n-1)
```

We can track how the function works by adding two print() function to the previous function definition:

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for ", n, " * factorial(", n-1, "): ", res)
    return res

print(factorial(5))
```

This Python script outputs the following results:

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for 2 * factorial( 1 ): 2
intermediate result for 3 * factorial( 2 ): 6
intermediate result for 4 * factorial( 3 ): 24
intermediate result for 5 * factorial( 4 ): 120
120
```

Let's have a look at an iterative version of the factorial function.

```
def iterative_factorial(n):
    result = 1
    for i in range(2,n+1):
        result *= i
    return result
```

### **The Pitfalls of Recursion**

This subchapter of our tutorial on recursion deals with the Fibonacci numbers. What do have sunflowers, the Golden ratio, fur tree cones, The Da Vinci Code and the song "Lateralus" by Tool in common. Right, the Fibonacci numbers.

The Fibonacci numbers are the numbers of the following sequence of integer values: 0,1,1,2,3,5,8,13,21,34,55,89, ... The Fibonacci numbers are defined by:  $F_n = F_{n-1} + F_{n-2}$  with  $F_0 = 0$  and  $F_1 = 1$  The Fibonacci sequence is named after the mathematician Leonardo of Pisa,

who is better known as Fibonacci. In his book "Liber Abaci" (publishes 1202) he introduced the sequence as an exercise dealing with bunnies. His sequence of the Fibonacci numbers begins with  $F_1 = 1$ , while in modern mathematics the sequence starts with  $F_0 = 0$ . But this has no effect on the other members of the sequence.

The Fibonacci numbers are easy to write as a Python function. It's more or less a one to one mapping from the mathematical definition:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

An iterative solution for the problem is also easy to write, though the recursive solution looks more like the mathematical definition:

```
def fibi(n):
    a, b = 0, 1
    for i in range(n):
        a, b = b, a + b
    return a
```

## Izuzetci

---

### Cilj

- Python izuzetci

### Izuzetci

Izuzetci (engl. exceptions)

Čak i ako su izjave ili izrazi sintaktički ispravni, može doći do greške kada se pokušaju izvršiti. Greške otkrivene u toku izvršenja programa se nazivaju izuzetci i nisu bezuslovno uvjetno fatalne.. Većina izuzetaka se ne obrađuju od strane programa

Primer

```
>>> 10 * (1/0)
```

Traceback (most recent call last):

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

ZeroDivisionError: integer division or modulo by zero

```
>>> 4 + spam*3
```

Traceback (most recent call last):

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

```
NameError: name 'spam' is not defined
```

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

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

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

```
TypeError: cannot concatenate 'str' and 'int' objects
```

Zadnja linija poruke greške označava ono što se dogodilo. Izuzetci dolaze u različitim vrstama, a tip greške je dio poruke: tip greške u primjeru je `ZeroDivisionError`, `NameError` i `TypeError`.

String odštampan kao tip izuzetka je ime ugrađenog izuzetka (engl. built-in exception) koji se dogodio. To vrijedi za sve ugrađene izuzetke, ali ne i za korisnički definirane izuzetke iznimke

Ostatak linije pruža detalja zavisno od vrste izuzetka i šta je uzrok izuzetka.

Dodatak

What is Exception?

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a `try`: block. After the `try`: block, include an `except`: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of `try....except...else` blocks:

```
try:
```

```
You do your operations here;
```

```
.....
```

```
except ExceptionI:
```

```
If there is ExceptionI, then execute this block.
```

```
except ExceptionII:
```

```
If there is ExceptionII, then execute this block.
```

```
.....
```

```
else:
```

```
If there is no exception then execute this block.
```

Here are few important points about the above-mentioned syntax:

A single `try` statement can have multiple `except` statements. This is useful when the `try` block contains statements that may throw different types of exceptions.

You can also provide a generic `except` clause, which handles any exception.

After the `except` clause(s), you can include an `else`-clause. The code in the `else`-block executes if the code in the `try`: block does not raise an exception.

The `else`-block is a good place for code that does not need the `try`: block's protection.

#### Example

Here is simple example, which opens a file and writes the content in the file and comes out gracefully because there is no problem at all:

```
#!/usr/bin/python

try:
    fh = open("testfile", "w")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
fh.close()
```

This will produce the following result:

Written content in the file successfully

#### Example

Here is one more simple example, which tries to open a file where you do not have permission to write in the file, so it raises an exception:

```
#!/usr/bin/python

try:
    fh = open("testfile", "r")
    fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
```

This will produce the following result:

Error: can't find file or read data

The *except* clause with no exceptions:

You can also use the `except` statement with no exceptions defined as follows:

```
try:
```

You do your operations here;

```
.....
```

except:

If there is any exception, then execute this block.

.....

else:

If there is no exception then execute this block.

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* clause with multiple exceptions

You can also use the same *except* statement to handle multiple exceptions as follows:

try:

You do your operations here;

.....

except(Exception1[, Exception2[,...ExceptionN]]):

If there is any exception from the given exception list,  
then execute this block.

.....

else:

If there is no exception then execute this block.

The try-finally clause

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

try:

You do your operations here;

.....

Due to any exception, this may be skipped.

finally:

This would always be executed.

.....

Note that you can provide except clause(s), or a finally clause, but not both. You can not use *else* clause as well along with a finally clause.

Example

```
#!/usr/bin/python
```

try:

```
fh = open("testfile", "w")
fh.write("This is my test file for exception handling!!")
finally:
    print "Error: can't find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result:

Error: can't find file or read data

Same example can be written more cleanly as follows:

```
#!/usr/bin/python

try:
    fh = open("testfile", "w")
    try:
        fh.write("This is my test file for exception handling!!")
    finally:
        print "Going to close the file"
    fh.close()
except IOError:
    print "Error: can't find file or read data"
```

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block. After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

### Argument of an Exception

An exception can have an *argument*, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the *except* clause as follows:

```
try:
    You do your operations here;
    .....
except ExceptionType,Argument:
    You can print value of Argument here...
```

If you are writing the code to handle a single exception, you can have a variable follow the name of the exception in the *except* statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable will receive the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

### Example

Following is an example for a single exception:

```
#!/usr/bin/python

# Define a function here.
def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "The argument does not contain numbers\n", Argument

# Call above function here.
temp_convert("xyz");
```

This would produce the following result:

```
The argument does not contain numbers
invalid literal for int() with base 10: 'xyz'
```

## Upravljanje izuzetcima

---

### Cilj

- Upravljanje izuzetcima

### Upravljanje izuzetcima

Upravljanje izuzetcima

Moguće pisati programe koji obrađuju odabrane izuzetke. Pogledajte sljedeći primjer, koji traži od korisnika za ulaz sve dok se unose cjelobrojne vrijednosti vrijedi, ali omogućuje korisniku da prekine program (koristeći Control-C ili što god operativni system podržava). Korisnički generisan prekid (engl. interrupt), se signalizira aktiviranjem KeyboardInterrupt izuzetkom.

Najjednostavniji primjer obrade izuzetja sa “try-except” blokom.

Primer

```
(x,y) = (5,0)
try:
    z = x/y
except ZeroDivisionError:
    print "divide by zero"
```

Za ispitivanje izuzetaka iz koda, koristiti slijedeći primjer:

Primer

```
(x,y) = (5,0)
try:
    z = x/y
```

```
except ZeroDivisionError, e:
```

```
z = e # representation: "<exceptions.ZeroDivisionError instance at 0x817426c>"
```

```
print z # output: "integer division or modulo by zero"
```

Kada treba nešto uraditi ako se desi izuzetak:

```
try;
```

```
do_some_stuff()
```

```
except:
```

```
rollback()
```

```
raise
```

```
else:
```

```
commit()
```

Kada treba nešto uradi bez obzira da li se izuzetak desio ili ne:

Primer

```
try:
```

```
do_some_stuff()
```

```
finally:
```

```
cleanup_stuff()
```

Primer >>> while True:

```
... try:
```

```
... x = int(raw_input("Please enter a number: "))
```

```
... break
```

```
... except ValueError:
```

```
... print "Oops! That was no valid number. Try again..."
```

Except izjava može imati više izuzetaka

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

```
... pass
```

Primer

```
import sys
```

```
try:
```

```
f = open('myfile.txt')
```

```
s = f.readline()
```

```
i = int(s.strip())
```

```
except IOError as e:
```

```
print "I/O error({0}): {1}".format(e.errno, e.strerror)
```

```
except ValueError:
```

```
print "Could not convert data to an integer."
```

```
except:
```



```

print "Unexpected error:", sys.exc_info()[0]
raise
Try ... except izjava ima opciju else.
for arg in sys.argv[1:]:
try:
f = open(arg, 'r')
except IOError:
print 'cannot open', arg
else:
print arg, 'has', len(f.readlines()), 'lines'
f.close()

```

Exception handlers don't just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

Manipulatori izuzetcima se ne koriste samo za izuzetke koji se odmah pojave u try strukturi ali i ako se dogode unutar funkcija koje su pozvane (čak i indirektno) u try strukturi

Primer

```

>>>
>>> def this_fails():
... x = 1/0
>>> try:
... this_fails()
... except ZeroDivisionError as detail:
... print 'Handling run-time error:', detail

```

Obradivanje greške pri izvršenju (engl. run-time error), dijeljenje sa 0 i modul sa 0

```

>>>
>>> def this_fails():
... x = 1/0
>>> try:
... this_fails()
... except ZeroDivisionError as detail:
... print 'Handling run-time error:', detail

```

Dodatak

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

## Syntax

Here is simple syntax of *try....except...else* blocks:

try:

You do your operations here;

.....

except *ExceptionI*:

If there is *ExceptionI*, then execute this block.

except *ExceptionII*:

If there is *ExceptionII*, then execute this block.

.....

else:

If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax:

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

## Example

Here is simple example, which opens a file and writes the content in the file and comes out gracefully because there is no problem at all:

```
#!/usr/bin/python
```

```
try:
```

```
fh = open("testfile", "w")
```

```
fh.write("This is my test file for exception handling!!!")
```

```
except IOError:
```

```
print "Error: can't find file or read data"
```

```
else:
```

```
print "Written content in the file successfully"
```

```
fh.close()
```

This will produce the following result:

Written content in the file successfully

## Example

Here is one more simple example, which tries to open a file where you do not have permission to write in the file, so it raises an exception:

```
#!/usr/bin/python
```

```
try:
```

```
fh = open("testfile", "r")
```

```
fh.write("This is my test file for exception handling!!")
```

```
except IOError:
```

```
print "Error: can't find file or read data"
```

```
else:
```

```
print "Written content in the file successfully"
```

This will produce the following result:

```
Error: can't find file or read data
```

The *except* clause with no exceptions:

You can also use the *except* statement with no exceptions defined as follows:

```
try:
```

```
You do your operations here;
```

```
.....
```

```
except:
```

```
If there is any exception, then execute this block.
```

```
.....
```

```
else:
```

```
If there is no exception then execute this block.
```

This kind of a try-except statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The *except* clause with multiple exceptions:

You can also use the same *except* statement to handle multiple exceptions as follows:

```
try:
```

```
You do your operations here;
```

```
.....
```

```
except(Exception1[, Exception2[,...ExceptionN]]):
```

```
If there is any exception from the given exception list,  
then execute this block.
```

```
.....
```

```
else:
```

```
If there is no exception then execute this block.
```

The try-finally clause:

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this:

try:

You do your operations here;

.....

Due to any exception, this may be skipped.

finally:

This would always be executed.

.....

Note that you can provide except clause(s), or a finally clause, but not both. You can not use *else* clause as well along with a finally clause.

Example

```
#!/usr/bin/python
```

try:

```
fh = open("testfile", "w")
```

```
fh.write("This is my test file for exception handling!!")
```

finally:

```
print "Error: can't find file or read data"
```

If you do not have permission to open the file in writing mode, then this will produce the following result:

```
Error: can't find file or read data
```

Same example can be written more cleanly as follows:

```
#!/usr/bin/python
```

try:

```
fh = open("testfile", "w")
```

try:

```
fh.write("This is my test file for exception handling!!")
```

finally:

```
print "Going to close the file"
```

```
fh.close()
```

except IOError:

```
print "Error: can't find file or read data"
```

When an exception is thrown in the *try* block, the execution immediately passes to the *finally* block.

After all the statements in the *finally* block are executed, the exception is raised again and is handled in the *except* statements if present in the next higher layer of the *try-except* statement.

## Aktiviranje izuzetka

---

### Cilj

- Aktiviranje izuzetka

### Aktiviranje izuzetka

**Raise** izjava omogućava programeru da izazove poseban izuzetak.

#### Primer

```
>>>
```

```
>>> raise NameError('HiThere')
```

Traceback (most recent call last):

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

NameError: HiThere

Nekada programmer želi samo da detektuje izuzetak, ali ne želi da ga obrađuje. Postoji jednostavnija forma *raise* izjave.

#### Primer

```
>>> try:
```

```
... raise NameError('HiThere')
```

```
... except NameError:
```

```
... print 'An exception flew by!'
```

```
... raise
```

```
...
```

An exception flew by!

Traceback (most recent call last):

File "<stdin>", line 2, in ?

NameError: HiThere

### Dodatak

#### Raising an exceptions

You can raise exceptions in several ways by using the *raise* statement.

The general syntax for the *raise* statement.

#### Syntax

```
raise [Exception [, args [, traceback]]]
```

Here, *Exception* is the type of exception (for example, `NameError`) and *argument* is a value for the exception argument. The argument is optional; if not supplied, the exception argument is `None`.

The final argument, `traceback`, is also optional (and rarely used in practice), and if present, is the `traceback` object used for the exception.

### Example

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows:

```
def functionName( level ):
    if level < 1:
        raise "Invalid level!", level
    # The code below to this would not be executed
    # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write our except clause as follows:

```
try:
    Business Logic here...
except "Invalid level!":
    Exception handling here...
else:
    Rest of the code here...
```

## Izuzetci programera

---

### Cilj

- Upoznavanje sa Python izuzetcima programera

### Izuzetci programera

Izuzetci koje generiše programer

Programer može dati ime svom izuzetku kreiranjem nove klase izuzetaka. Izuzetci se kreiraju iz klase `Exception` bilo Direktno ili indirektno.

Primer

```
>>>
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
```

```

... return repr(self.value)
>>> try:
... raise MyError(2*2)
... except MyError as e:
... print 'My exception occurred, value:', e.value
...

```

My exception occurred, value: 4

```
>>> raise MyError('oops!')
```

Traceback (most recent call last):

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

```
__main__.MyError: 'oops!'
```

Kada se kreira modul koji treba da se aktivira s nekoliko tipova grešaka, obično se kreira bazna klasa za izuzetke definisane u tom modulu i podklase za kreiranje posebnih klasa izuzetaka za različite tipove grešaka:

```
class Error(Exception):
```

```
    """Base class for exceptions in this module."""
```

```
    pass
```

```
class InputError(Error):
```

```
    """Exception raised for errors in the input.
```

Attributes:

expr -- input expression in which the error occurred

msg -- explanation of the error

```
    """
```

```
    def __init__(self, expr, msg):
```

```
        self.expr = expr
```

```
        self.msg = msg
```

```
class TransitionError(Error):
```

```
    """Raised when an operation attempts a state transition that's not allowed.
```

Attributes:

prev -- state at beginning of transition

next -- attempted new state

msg -- explanation of why the specific transition is not allowed

```
    """
```

```
    def __init__(self, prev, next, msg):
```

```
        self.prev = prev
```

```
        self.next = next
```

```
        self.msg = msg
```

Dodatak

What is Exception?

An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it can't cope with, it raises an exception. An exception is a Python object that represents an error.

When a Python script raises an exception, it must either handle the exception immediately otherwise it would terminate and come out.

Handling an exception

If you have some *suspicious* code that may raise an exception, you can defend your program by placing the suspicious code in a try: block. After the try: block, include an except: statement, followed by a block of code which handles the problem as elegantly as possible.

Syntax

Here is simple syntax of *try....except...else* blocks:

try:

You do your operations here;

.....

except *ExceptionI*:

If there is *ExceptionI*, then execute this block.

except *ExceptionII*:

If there is *ExceptionII*, then execute this block.

.....

else:

If there is no exception then execute this block.

Here are few important points about the above-mentioned syntax:

A single try statement can have multiple except statements. This is useful when the try block contains statements that may throw different types of exceptions.

You can also provide a generic except clause, which handles any exception.

After the except clause(s), you can include an else-clause. The code in the else-block executes if the code in the try: block does not raise an exception.

The else-block is a good place for code that does not need the try: block's protection.

Example

Here is simple example, which opens a file and writes the content in the file and comes out gracefully because there is no problem at all:

```
#!/usr/bin/python
```

```
try:
```

```
fh = open("testfile", "w")
```



```

fh.write("This is my test file for exception handling!!")
except IOError:
    print "Error: can't find file or read data"
else:
    print "Written content in the file successfully"
fh.close()

```

This will produce the following result:  
 Written content in the file successfully

### User-Defined Exceptions

Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

Here is an example related to `RuntimeError`. Here, a class is created that is subclassed from *`RuntimeError`*. This is useful when you need to display more specific information when an exception is caught.

In the try block, the user-defined exception is raised and caught in the except block. The variable `e` is used to create an instance of the class `Networkerror`.

```

class Networkerror(RuntimeError):
    def __init__(self, arg):
        self.args = arg

```

So once you defined above class, you can raise your exception as follows:

```

try:
    raise Networkerror("Bad hostname")
except Networkerror,e:
    print e.args

```

## L7: Osnove Python programiranja

---

### **Zaključak\***

Nakon studiranja sadržaja ovog poglavlja, studenti će steći početna znanja iz Python programiranja.

# LearningObject

---

## Python - kontrolne strukture, funkcije, strukture podataka

---

### Cilj

- Razumevanje python kontrolnih struktura, stringova, nizova i listi.

### Python nizovi i stringovi

#### Stringovi

Stringovi u pythonu predstavljaju kolekcije tekstualnog sadržaja. Python podržava razne operacije nad stringovima, a za razliku od nizova, python stringovi nisu promenljivi (tj kada vrednost stringa se ne može menjati - može se samo napraviti novi string).

```
>> "double quoted"
```

```
'double quoted'
```

```
>> 'single quoted'
```

```
'single quoted'
```

```
>> len("duzina stringa")
```

```
14
```

```
# konkatencija stringova (vraca novi string)
```

```
>> str = "Hello "
```

```
>> str = str + "World!"
```

```
>> str
```

```
"Hello world!"
```

```
# indeksiranje stringova
```

```
>> "Hello"[0]
```

```
'H'
```

```
# slicing - vraca string od prvog indeksa (inkluzivno) do drugog (ekskluzivno)
```

```
>> "Hello"[1:5]
```

```
'ello'
```

```
# Ukoliko se drugi indeks ne definiše, slice-uje se do kraja stringa
```

```
>> "Hello"[1:]
```

```
'ello'
```

```
# Ukoliko se prvi indeks ne definiše, slice-uje se od početka stringa
```

```
>> "Hello"[:3]
```

```
'Hel'
```

# Moguce je koristiti i negativne indekse:

```
>> "Hello"[1:-1]
```

```
'ell'
```

### Interpolacija stringova

Ubacivanje vrednosti izraza u stringove se radi operatorom '%'.

```
>> a = 5
```

```
>> b = 4
```

```
>> "Zbir %d i %d je %d" % (a, b, a + b)
```

```
'Zbir 5 i 4 je 9'
```

Karakter '%' u stringu se menja vrednošću iz n-torke desno od operatora %. Posle karaktera % se nalazi specifikator tipa, u ovom slučaju %d označava da je u pitanju celobrojna vrednost. %s bi označila da je u pitanju string.

### Pretvaranje vrednosti u stringove

Pretvaranje vrednosti u stringove se radi pozivanjem funkcije str().

```
>> str(5)
```

```
'5'
```

```
>> str(True)
```

```
'True'
```

Ekvivalentne funkcije postoje za pretvaranje stringova u druge tipove:

```
>> float("5")
```

```
5.0
```

```
>> int("5")
```

```
>> 5
```

```
>> bool("True")
```

```
True
```

### Nizovi

Nizovi se kreiraju literalom [], kao u JavaScript-u i Ruby-u. Nizovi mogu sadržati elemente bilo kog tipa, uključujući i druge nizove. Za razliku od stringova, nizovi se mogu menjati nakon kreiranja.

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

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

Sintaksa za indeksiranje i slicing je identična kao za stringove:

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

```
>> a[1:3]
```

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

# konkatencija nizova
>> [1,2,3] + [4,5]
[1, 2, 3, 4, 5]

# ponavljanje nizova
>> [1,2,3] * 2
[1, 2, 3, 1, 2, 3]

# duzina niza
>> len([1,2,3])
3
```

### **n-torke**

n-torke su kolekcije koje sadrže fiksni broj elemenata. U pythonu se kreiraju operatorom ",", sa ili bez zagrada.

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

>> (3, "a")
(3, 'a')
```

# n-torke se mogu koristiti za dodeljivanje vrednosti:

```
>> a, b, c = 1, 2, 3
>> a
1
>> b, c
(2, 3)
```

nizovi se mogu otpakovati na sličan način, pod uslovom da je broj elemenata niza jednak broju promenljivih:

```
>> a, b, c = [1, 2, 3]
>> a, b, c = [1, 2, 3, 4]
```

ValueError: too many values to unpack (expected 3)

## Kontrolne strukture i funkcije

Svaki početak novog bloka u python-u se započinje karakterom ":", novom linijom i povećanom indentacijom.

Primer if-a:

```
>>> x = int(raw_input("Please enter an integer: "))
```

```
Please enter an integer: 42
```

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

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

```
... print('Negative changed to zero')
```

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

```
... print 'Zero'
```

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

```
... print('Single')
```

```
... else:
```

```
... print('More')
```

```
...
```

```
More
```

## For petlja

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

```
>> for item in array:
```

```
... print(item)
```

```
...
```

```
1
```

```
2
```

```
3
```

```
4
```

```
5
```

## While petlja

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

```
>> i = 0
```

```
>> while i < len(array):
```

```
... print(array[i])
```

```
... i = i + 1
```

```
...
```

```
1
```

```
2
```

3

**Definisanje funkcija**

Za definisanje funkcija se koristi keyword **def**. Domen funkcije je blok koji sledi ispod zaglavlja funkcije (blok mora biti uvučen):

```
def add(a, b):
```

```
    return a + b
```

```
add(2, 3) # vraća 5
```

# funkcije ce baciti exception ukoliko ne dobiju definisan broj argumenata:

```
>> add(2)
```

```
TypeError: add() missing 1 required positional argument: 'b'
```

```
>> add(1,2,3)
```

```
TypeError: add() takes 2 positional arguments but 3 were given
```

**Default argumenti**

Funkcije u pythonu mogu imati default argumente.

```
def foo(a, b = 5):
```

```
    return a, b
```

```
>> foo(1, 2)
```

```
(1, 2)
```

```
>> foo(1)
```

```
(1, 5)
```

**Rekurzija**

Rekurzija je metod za rešavanje problema u kome se rešenje predstavlja u vidu rešenja istog problema manjeg opsega, do "baznih slučajeva" koji predstavljaju rešenja nekih od tih pod-problema. Konkretno:

```
# Racuna factorial broja n.
```

```
# fac(n) = 1 * 2 * 3 * ... * n
```

```
# ili
```

```
# fac(n) = n * n-1 * n-2 * ... * 1
```

```
def fac(n):
```

```
    if n == 1:
```

```
        return 1
```

```
    else:
```

```
        return n * fac(n-1)
```

```
print(factorial(5)) # ispisuje 120
```

U primeru iznad, `fac` je rekurzivna funkcija koja računa faktorial nekog broja. Rekurzivni deo problema se može pročitati kao "faktorial nekog broja  $n$  je taj broj pomnožen faktorialom  $n-1$ ". Bazni slučaj je `fac(1)`, za koji se samo vraća vrednost 1.

Izvršavanje ove funkcije rekurzivno izgleda ovako:

```
fac(5) = 5 * fac(4)
fac(5) = 5 * 4 * fac(3)
fac(5) = 5 * 4 * 3 * fac(2)
fac(5) = 5 * 4 * 3 * 2 * fac(1)
fac(5) = 5 * 4 * 3 * 2 * 1 # bazni slucaj
fac(5) = 5 * 4 * 3 * 2
fac(5) = 5 * 4 * 6
fac(5) = 5 * 24
fac(5) = 120
```

Da bi se rekurzivna funkcija završila (tj da ne bi bila izvršavana beskonačno dugo ili dok ne se ne zauzme dozvoljena stack memorija), bitno je obratiti pažnju na sve moguće bazne slučajeve i osigurati da se rekurzivna funkcija uvek završava.

# funkcija koja rekurzivno sumira sve elemente liste

```
def sum(arr):
    if len(arr) == 1:
        return arr[0]
    else:
        return arr[0] + sum(arr[1:])
```

U primeru iznad, bazni slučaj je niz sa samo jednim elementom, dok rekurzivni deo poziva samu funkciju sa sve manjim brojem elemenata.

## Greške i izuzeci

### Izuzeci

Izuzeci su posebna vrsta greške, koja "putuje" od mesta na kom je generisana (raised, thrown), ka spoljašnjosti steka izvršavanja. Ukoliko se exception ne uhvati, izazvaće kraj izvršavanja programa. Hvatanje izuzetka omogućava njegovu obradu, logovanje, i slično. Svaki izuzetak je zapravo instanca Exception klase (ili neke od njenih potomaka).

### Generisanje greški

```
raise RuntimeError("Poruka o gresci") # generiše izuzetak tipa RuntimeError sa porukom
```

Ukoliko se izuzetak baci iz funkcije koja je predhodno pozvana od strane neke druge funkcije, a ne bude uhvaćen, putovaće ka "spoljašnjosti" dok ne bude uhvaćen ili izazove fatalnu grešku. Na primer:

```
def inner():
    raise RuntimeError("Hello!")
```

```
def outer():
    inner()
```

outer() # izuzetak se generise u funkciji inner, prolazi kroz outer koji ga takodje ne hvata, i izaziva fatalnu gresku

### Hvatanje exception-a

Hvatanje izuzetaka se vrši try/except blokom. Kod koji može da baci exception se stavlja u try blok, dok se kod koji radi obradu ili oporavak od greške stavlja u except blok.

```
def throws_and_catches():
    try:
        a = 1/0 # Deljenje nulom je greska
    except ZeroDivisionError as error:
        print(error) # stampa "division by zero"

throws_and_catches()
```

Iz primera iznad se vidi da se exception hvata po tipu - moguće je hvatati nekoliko tipova exceptiona odjednom, postavljanjem except blokova jedan ispod drugog. Python će pokušati da izvrši jedan od blokova koji odgovara tipu greške, a ako ne uspe, exception će nastaviti niz stek izvršavanja.

Takođe se vidi da se exception objekat može dodeliti promenljivoj (except Tip as promenljiva), što omogućava ispisivanje greške ili steka izvršavanja.

### Finally

Finally je još jedan blok vezan za hvatanje exceptiona, ali za razliku od try ili except bloka, on će se uvek izvršiti, bez obzira na to da li je exception bačen ili ne. Ovo omogućava programeru da ispiše nešto u log bez obzira na greške, da zatvori ili oslobodi otvorene resurse i td.

```
def divide(x, y):
    try:
        result = x / y
        print("result is", result)
    except ZeroDivisionError:
        print("division by zero!")
    finally:
        print("executing finally clause")

divide(5, 2) #ispisuje "resut is 2.5" pa "executing finally clause"
divide(5, 0) #baca ZeroDivisionError, ali i dalje ispisuje "executing finally clause"
```

## Python - kontrolne strukture, greške, rekurzija

---



## Python - kontrolne strukture

1) Napisati funkciju koja kao argumente prima dva broja, a i b. Ukoliko je a manje od b, vraća njihov zbir, ukoliko je a veće ili jednako b, vraća njihov proizvod. Obratiti pažnju da funkcija treba da vrati broj, a ne da ga ispiše. Ispisivanje raditi funkcijom print van definicije same funkcije.

2) Napisati funkciju koja prihvata listu i vraća novu listu (ne menjajući staru), takvu da je svaki element originalne liste uvećan za jedan.

3) Napisati rekursivnu funkciju (ne koristeći ni jednu od while ili for petlji) koja prihvata dve liste i vraća novu listu (ne menja staru), takvu da je svaki element nove liste jednak zbiru elemenata iz dve prosleđene liste na istom indeksu. Na primer:

```
add_lists([1,3,5], [2,4,6]) # vraca listu [3, 7, 11]
```

U slučaju da su liste različitih dužina, baciti RuntimeError sa porukom "Liste su različitih dužina".