Advanced Programming-Python

Presented by

Siddhanta Borah

Contents of Todays Discussion

INTRODUCTION- FUNCTION

Functions are self-contained programs that perform some particular tasks. Once a function is created by the programmer for a specific task, this function can be called anytime to perform that task.

Suppose, we want to perform a task several times, in such a scenario, rather than writing code for that particular task repeatedly, we create a function for that task and call it when we want to perform the task. Each function is given a name, using which we call it. A function may or may not return a value.

There are many advantages of using functions:

a) They reduce duplication of code in a program.
b) They break the large complex problems into small parts.
c) They help in improving the clarity of code (i.e., make the code easy to understand).
d) A piece of code can be reused as many times as we want with the help of functions.

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Type Conversion

There are some built-in functions in the Python programming language that can convert one type of data into another type. For example, the `int` function can take any number value and convert it into an integer.

Example

```
>>>int(5.5)
5           # Output
>>>int('Python')
Traceback (most recent call last):          # Output
  File "<pyshell#21>", line 1, in <module>
int('Python')
ValueError: invalid literal for int() with base 10: 'Python'
>>>int('5')
5           # Output
```

In the above examples, you can see that in the first case, we took a floating-point number 5.5 that was converted to an integer number 5 by the `int` function. In the second case, we took a string that was not a number and applied `int` function to it, but got an error. This means that a string that is not a number cannot be converted to an integer. However, in the third case, we took a number in string form and converted it to an integer 5 using `int` function.

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Similarly, we have a function `float`, which can convert integers and string into floating-point numbers.

Example

```
>>>float(45)
     45.0          # Output
>>> float ('5')
     5.0           # Output
```

Finally, Python has a `str` function that converts the types into strings.

Example

```
>>>str(67)
     '67'          # Output
```

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Type Coercion

Type conversion discussed above is known as explicit conversion.

There is also another kind of type conversion in the Python language, known as implicit conversion. Implicit conversion is also known as type coercion and is automatically done by the interpreter.

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Type coercion is a process through which the Python interpreter automatically converts a value of one type into a value of another type according to the requirement.

Example

Suppose we want to calculate an elapsed fraction of an hour. The expression `minutes/60` does integer arithmetic and gives result, even 59 minutes past hour.

One solution is that we convert the `minutes` to a floating-point number using type conversion and do floating-point division:

```
>>>minute=59
>>>float(minute)/60
0.98333333333            # Output
```

Alternatively, we can take advantage of type coercion process in Python. For the mathematical operators, if either operand is a float, the other is automatically converted to float:

```
>>>minute=59
>>>minute/60.0
0.98333333333            # Output
```

Hence, in the example given above, we make the denominator a float number. The Python interpreter automatically converts the numerator into float and does the calculation.

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Mathematical Functions

In mathematics, we have functions such as `sin` and `log` and we have to evaluate some expressions like `sin(pi/4)` and `log(1/x)`. We follow a process to solve this type of expressions; first, we solve the innermost part of the parenthesis, and then move on to the outer functions.

Python provides us a *Math module* that contains most of the familiar and important mathematical functions. A *module* is a file that contains some predefined Python codes. A module can define functions, classes and variables. It is a collection of related functions grouped together.

Before using a module in Python, we have to import it.

For example, to import the math module, we use:

```
>>> import math
```

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Example

Find the `cos` of `45`degrees.

```
>>>degree = 45
>>>angle = degree * 2 * math.pi/360.0
>>>math.cos(angle)
0.7071067811865476                # Output
```

In the given example, we use a `math.pi` function in order to get the variable `pi` from the `math` module. The value of this variable is an approximation of π up to 15 digits.

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Date and Time

Python provides the built-in modules `time` and `calendar` through which we can handle date and time in several ways. For example, we can use these modules to get the current time and date. In order to use the time module, we need to import it into our program first. Similarly, for working with dates, we have to import the calendar module first.

Examples

Getting current date and time:

```
>>> import time;
>>>
>>> localtime = time.localtime(time.time())
>>> print "Local current time : ", localtime
#Output
Local current time :  time.struct_time(tm_year=2016, tm_mon=5, tm_mday=31,
tm_hour=19, tm_min=21, tm_sec=50, tm_wday=1, tm_yday=152, tm_isdst=0)
```

This example gives us the current time and date. This function returns a time-tuple with nine items. If we want, we can change the format in which the time and date is given.

TYPES OF FUNCTION BUILT-IN FUNCTIONS

Getting calendar for a month:

Python provides us a calendar module through which we can use yearly and monthly calendars according to our requirement. In the example below, we print a calendar for the month of October, 2015.

```
>>> import calendar

>>> c = calendar.month(2015,10)
>>> print "Calender for October, 2015: \n", c

Calender for October, 2015:              #Output
    October 2015
Mo Tu We Th Fr Sa Su
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

TYPES OF FUNCTION BUILT-IN FUNCTIONS

`help()` FUNCTION

`help()` function is a built-in function in Python Programming Language which is used to invoke the help system. It takes an object as an argument. It gives all the detailed information about that object like if it's a module, then it will tell you about the sub-modules, functions, variables and constants in details.

Example

```
# import math module
>>> import math
>>> help(math.sin)   #give detailed info about sin function in math module
Help on built-in function sin in module math:

sin(...)
    sin(x)

    Return the sine of x (measured in radians).

>>> help(math.cos)   #give detailed info about cos function in math module
Help on built-in function cos in module math:

cos(...)
    cos(x)

    Return the cosine of x (measured in radians).
```

TYPES OF FUNCTION COMPOSITION OF FUNCTIONS

Composition is a concept that you might have come across in algebra pre-calculus. The syntax of composition in mathematics is as follows:

```
f(g(x)) = f o g(x),
```

where, `f` and `g` are functions. This means the return value of function `g` is passed into the function `f` as parameters/arguments.

Just as with the mathematical functions, Python functions can also be composed. We can use any kind of expression including arithmetic operators as an argument to a function.

Example

```
>>> x = math.sin(angle + math.pi/4)
```

In the given example, we have used an expression `angle + math.pi/4` as an argument to the function `math.sin`. First, the value of the innermost expression is computed, and then the resulting value is used as the argument for the function `math.sin`.

TYPES OF FUNCTION COMPOSITION OF FUNCTIONS

Similarly, we can also take a function as an argument to another function.

Example

```
>>> x = math.exp(math.log(10.0))
```

Here, the value of the function `math.log(10.0)` is calculated first and then used as the argument for the function `math.exp`.

EXAMPLE

- **Write a program to print the calendar for the month of March, 1991.**
- **Write a python program to evaluate the following using *math* function**
  a. $Y = \text{Sin}\,(e^{x+2})$
  b. $Y = \text{Cos}(x+4)$
  c. $Y = \log(1+1/x)$

TYPES OF FUNCTION USER DEFINED FUNCTIONS

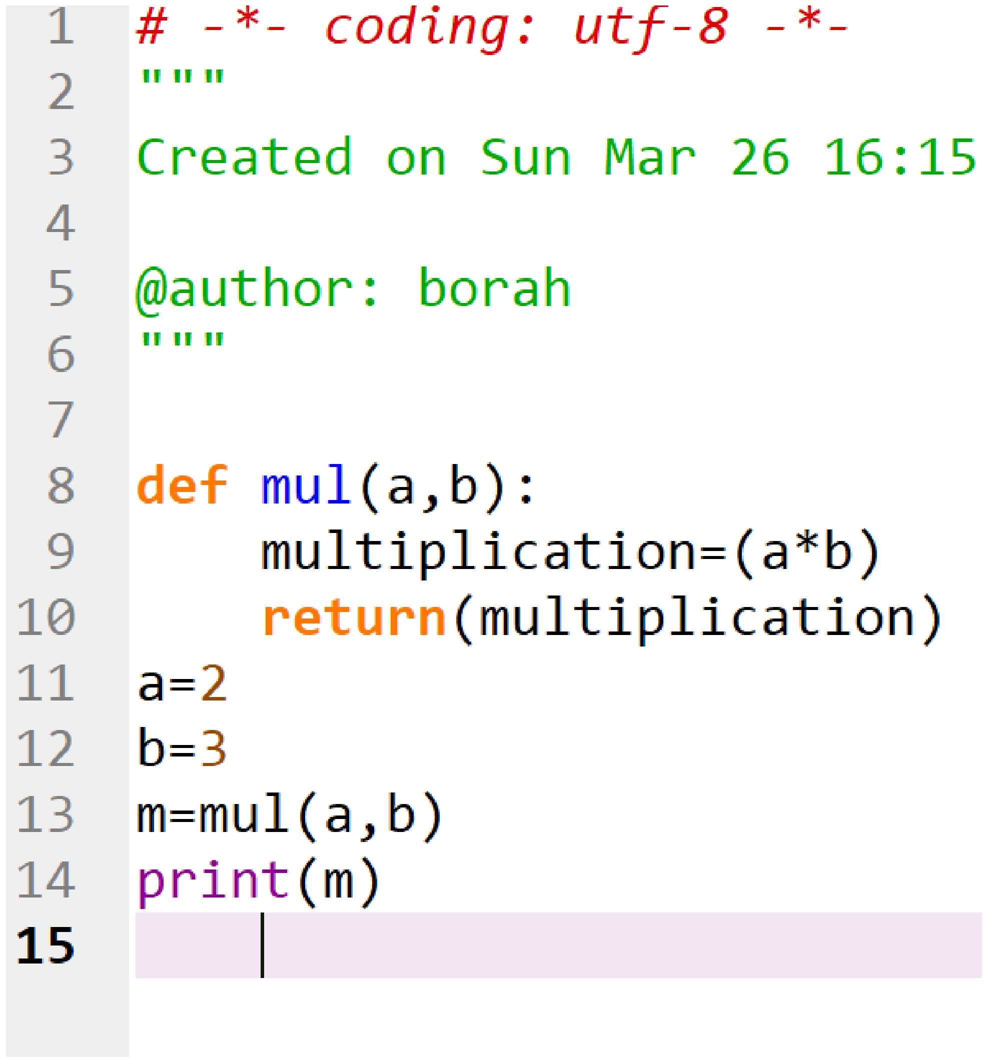
Until now, we have seen only the built-in functions of Python, but as with many other languages, Python also allows users to define their own functions. To use their own functions in Python, users have to define the function first; this is known as *Function Definition*. In a function definition, users have to define a name for the new function and also the list of the statements that will execute when the function will be called.

The block of the function starts with a keyword *def* after which the function name is written followed by parentheses. We can also give some input parameters or arguments to a function by placing them within these parentheses. The parameters can also be defined within these parentheses. The block of statements always starts with a colon `(:)`. After writing the code statements, the block is ended with a return statement whose syntax is `return [expression]`. As we have stated earlier, a function may or may not return a value. If you want to return more than one value, separate the values using commas. The default return value is NONE.

Syntax

```
def functionname(parameters):
    "function_docstring"
    statement(s)
    return [expression]
```

EXAMPLE



```python
# -*- coding: utf-8 -*-
"""
Created on Sun Mar 26 16:15

@author: borah
"""

def mul(a,b):
    multiplication=(a*b)
    return(multiplication)
a=2
b=3
m=mul(a,b)
print(m)
```

```
>>> def print_lines():
    print "Hello Python!!"
    print "Welcome to Python Programming!!"

>>> def new_print():
     print_lines()
     print_lines()

>>> new_print()

Hello Python!!                  # Output
Welcome to Python Programming!!
Hello Python!!
Welcome to Python Programming!!
```

PARAMETERS AND ARGUMENT

Parameters and arguments are the values or expressions passed to the functions between parentheses. As we have seen in earlier sections, many of the built in functions need arguments to be passed with them: the `math.cos()` function takes a number, i.e., the value of the angle as an argument. Many functions require two or more arguments to be passed such as the power function in math module `math.pow()`, where we have to pass two arguments, the base and the exponent.

The value of the argument is always assigned to a variable known as parameter. At the time of function definition, we have to define some parameters if that function requires some arguments to be passed at the time of calling.

Example

```
>>>defprint_lines(line):
... print line
... print line
```

PARAMETERS AND ARGUMENT

In this function we have defined a variable `line` which is a parameter. Now, when the function is called, it prints the value of the parameter `line` twice.

```
>>>print_lines('Hello')
Hello                 # Output
Hello                 # Output

>>>print_lines(17)
17            # Output
17            # Output

>>>print_lines(math.pi)
3.14159265359            # Output
3.14159265359            # Output
```

PARAMETERS AND ARGUMENT

There can be four types of formal arguments using which a function can be called which are as follows:

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

PARAMETERS AND ARGUMENT

1. Required arguments When we assign the parameters to a function at the time of function definition, at the time of calling, the arguments should be passed to a function in correct positional order; furthermore, the number of arguments should match the defined number of parameters.

Example

```
>>>defprint_lines(str)
... print str
... return;
```

We have defined one parameter `str` to the function `print_lines`. Hence, at the time of calling, we have to pass exactly one argument to the function, otherwise it will produce an error.

```
# function calling here
>>>print_lines();
Traceback (most recent call last):              # Output
  File "<pyshell#18>", line 1, in <module>
print_lines()
TypeError: print_lines() takes exactly 1 argument (0 given)
```

There is an error because we did not pass any argument to the function `print_lines`, while according to the function definition, the function `print_lines` must take exactly one argument.

PARAMETERS AND ARGUMENT

2. Keyword arguments In keyword arguments, the caller recognises the arguments by the parameter's names. This type of argument can also be skipped or can also be out of order.

Example 1

```
>>>defprint_lines():
... print str
... return
...

# function calling here
>>>print_lines(str = "Hello Python");
Hello Python                  # Output
```

Example 2

```
# Function Definition
>>>defprint_info(name, age):
... print "Name: ", name
... print "Age: ", age
... return
...

# function calling
>>>print_info(age=15, name='john');
Name: john             # Output
Age: 15                # Output
```

PARAMETERS AND ARGUMENT

3. Default arguments In default arguments, we can assign a value to a parameter at the time of function definition. This value is considered the default value to that parameter. If we do not provide a value to the parameter at the time of calling, it will not produce an error. Instead it will pick the default value and use it.

Example

```
# Function definition
>>>defprint_info(name, age=35):
... print "Name: ",name
... print "Age: ", age
... return
...

# function calling
>>>print_info(age=20, name='john');
Name: john                 # Output
Age: 20             # Output
>>>
>>>print_info(name='john');
Name: john                # Output
Age: 35             # Output
```

PARAMETERS AND ARGUMENT

In the given example, we have given a value `35` to the parameter `age`. It is the default value for `age`. Now, in the first function call, we provide the value of `age` as `20`. Hence, the Python interpreter takes the value provided by us and does not use the default value.

However, in the second function call, we do not provide the value for age. Hence, the Python interpreter does not produce any error. Rather it picks the default value from the function definition and displays it. This is how default arguments are used in function calling.

PARAMETERS AND ARGUMENT

4. Variable-length arguments There are many cases where we are required to process a function with more number of arguments than we specified in the function definition. These types of arguments are known as variable-length arguments. The names for these arguments are not specified in the function definition. Instead we use an asterisk (*) before the name of the variable which holds the value for all non-keyword variable arguments.

Syntax

```
def function_name([formal_args] *var_args_tuple):
    "function_docstring"
    function_body
    return[expression]
```

PARAMETERS AND ARGUMENT

Example

```
# function definition here
>>>defprint_info(arg1,*vartuple):
... print "Result is: "
... print arg1
... for var in vartuple:
...          print var
... return

# function call
>>>print_info(10);
10          # Output
>>>print_info(90,60,40);
90          # Output
60          # Output
40          # Output
```

FUNCTIONS CALLS

We define a function by giving it a name with some parameters (optional) and then a sequence of statements. Later, we can call the function when we need it. A function is called using the name with which it was defined earlier, followed by a pair of parentheses (`()`). Any input parameters or arguments are to be placed within these calling parentheses.

All parameters (arguments) which are passed in functions are always passed by reference in Python. This means that if the values of the parameters are changed in the function, it will also reflect the change in the calling function.

FUNCTIONS CALLS

Write a function which accepts two numbers and returns their sum.

```
>>>def sum(arg1,arg2):
...  sum = arg1 + arg2
...  return sum
# Now calling the function here
>>> a = 4
>>> b = 3
>>>total = sum(a,b)  # calling the mult function
>>>print(total)
```

THE RETURN STATEMENT

The return statement is used to exit a function. A function may or may not return a value. If a function returns a value, it is passed back by the return statement as argument to the caller. If it does not return a value, we simply write return with no arguments.

Syntax

```
return [expression]
```

Example

```
# function definition here
>>>def div(arg1, arg2):
... division = arg1/arg2
... return division
...

# function call here
>>> arg3 = div(20,10)
>>> print "division: ", arg3
division: 2               # Output
```

THE RETURN STATEMENT

In the given example, we define a function `div` that divides one argument by another and stores the result in the variable `division`; then the value of `division` variable is returned by the function to the caller variable `arg3`.

PYTHON RECURSIVE FUNCTION

Recursion is generally understood to be the process of repeating something in a self-similar way. For example, if an object is placed between two mirrors facing each other, the object will be reflected recursively.

In the programming context, the meaning of recursion remains the same. Here, if a function, procedure or method calls itself, it is called recursive. In Python, we know that a function can call another function, but it is also possible that a function calls itself.

Let us look at an example of a recursive function by computing the factorial of a number. The factorial of any number is defined by multiplying all the integers from 1 to that number. For example, the factorial of 5 is `1*2*3*4*5 = 120`.

PYTHON RECURSIVE FUNCTION

Example

```
>>> def fact_rec(x):
    'Recursive function to find the factorial of an integer'
... if x == 1:
...         return 1
... else:
...         return(x * fact_rec(x-1))

>>> fact_rec(4)
24                              # Output
>>> fact_rec(10)
3628800                         # Output
```

ANONYMOUS FUNCTIONS

The anonymous functions are the functions created using a *lambda* keyword. They are not defined by using *def* keyword. For this reason, they are called anonymous functions.

We can pass any number of arguments to a lambda form functions, but still they return only one value in the form of expression. An anonymous function cannot directly call `print` command as the lambda needs an expression. It cannot access the parameters that are not defined in its own namespace. An anonymous function is a single line statement function.

Syntax

```
lambda [arg1[,arg2,......argn]]:expression
```

The syntax of the lambda function is a single statement.

ANONYMOUS FUNCTIONS

Example

```
# function definition here
>>>mult = lambda val1, val2: val1*val2;

# function call here
>>> print "value: ", mult(20,40)
Value: 800          # Output
```

In the given example, the `lambda` function is defined with two arguments `val1` and `val2`. The expression `val1*val2` does the multiplication of the two values. Now, in function call, we can directly call the `mult` function with two valid values as arguments and produce the output as above.

EXAMPLE

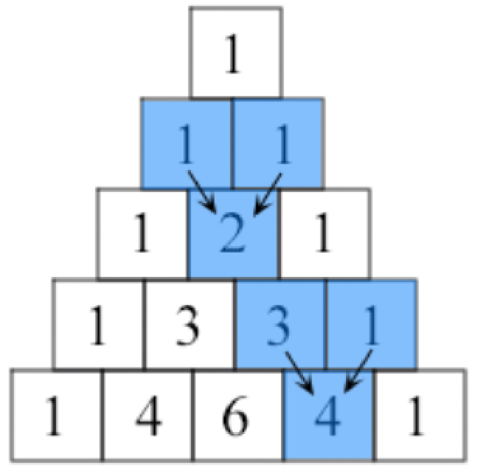
- **Write a function to display Fibonacci sequence using recursion.**
- Write a function to find the sum of several natural numbers using recursion.

ASSIGNEMENT-1

1) Write a user defined function to find the HCF of numbers provided by user.
2) Write a user defined function to find the LCM of numbers provided by user.
3) Write a user defined function to find the current date and time.
4) Write a python program to make a simple calculator.
5) Write a python function to find the volume and surface area of a sphere and cylinder.
6) Write a Python function to check whether a number falls within a given range.
7) Write a Python function to check whether a number is "Perfect" or not.
8) Write a Python function that prints out the first n rows of Pascal's triangle.

In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum). Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself). *Example* : The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and 1 + 2 + 3 = 6. Equivalently, the number 6 is equal to half the sum of all its positive divisors: (1 + 2 + 3 + 6) / 2 = 6. The next perfect number is 28 = 1 + 2 + 4 + 7 + 14. This is followed by the perfect numbers 496 and 8128.

Sample Pascal's triangle :



Thank You