Python Programming

RGM College of Engineering & Technology (Autonomous)

Department of Computer Science & Engineering

Academic Year : 2020-2021

FUNCTIONS - 6

Agenda:

1. Function Aliasing
2. Nested Functions
3. Fruitful Functions

7. FUNCTION ALIASING

Function Aliasing

For the existing function we can give another name, which is nothing but function aliasing.

Eg:

```
def wish(name):
    print("Good Morning:",name)
greeting = wish
print(id(wish))
print(id(greeting))
greeting('Karthi')
wish('Karthi')
```

```
1552601886504
1552601886504
Good Morning: Karthi
Good Morning: Karthi
```

Note:

- In the above example only one function is available but we can call that function by using either wish name or greeting name.
- If we delete one name still we can access that function by using alias name.

Eg:

```
def wish(name):
    print("Good Morning:",name)
greeting=wish
greeting('Karthi')
wish('Karthi')
del wish
wish('Karthi')          #NameError: name 'wish' is not defined
greeting('Saha')
```

```
Good Morning: Karthi
Good Morning: Karthi
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-f292fb57f669> in <module>
      8
      9 del wish
---> 10 wish('Karthi') #NameError: name 'wish' is not defined
     11 greeting('Saha')

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

Eg:

```
def wish(name):
    print("Good Morning:",name)
greeting=wish
rgm = greeting
greeting('Karthi')
wish('Karthi')
rgm('Karthi')
del wish
#wish('Karthi')
greeting('Saha')
```

```
Good Morning: Karthi
Good Morning: Karthi
Good Morning: Karthi
Good Morning: Saha
```

#NameError: name 'wish' is not defined

8. NESTED FUNCTIONS

Nested Functions

- We can declare a function inside another function, such type of functions are called **Nested functions**. Where we have this type of requirement?

If a group of statements inside a function are repeatedly requires, then these group of statements we will define as inner function and we can call this inner function whenever need arises.

Eg:

```
def outer():
    print("outer function started")
    def inner():
        print("inner function execution")          # It is function declaration
    print("outer function calling inner function")
    inner()                                        # It is a function call
outer()
```

```
outer function started
outer function calling inner function
inner function execution
```

Eg:

```
def outer():
    print("outer function started")
    def inner():
        print("inner function execution")
    print("outer function calling inner function")
    inner()
outer()
inner()
```

```
outer function started
outer function calling inner function
inner function execution

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-16-63fb5b235d22> in <module>
      6     inner()
      7 outer()
----> 8 inner() #NameError: name 'inner' is not defined

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

In the above example inner() function is local to outer() function and hence it is not possible to call inner() function directly from outside of outer() function.

Another Example:

```
def f1():
    def inner(a,b):
        print('The Sum :',a+b)
        print('The Average :',(a+b)/2)
    inner(10,20)
    inner(20,30)
    inner(40,50)
    inner(100,200)
f1()
```

```
The Sum : 30
The Average : 15.0
The Sum : 50
The Average : 25.0
The Sum : 90
The Average : 45.0
The Sum : 300
The Average : 150.0
```

Is it possible to pass a function as an argument to another function?

- Yes, a function can take another function as an argument.

For example,

- filter(function, Sequence)
- map(function, Sequence)
- reduce(function, Sequence)

Note:

- **A function can return another function**. For example,

```
def outer():
    print("outer function started")
    def inner():
        print("inner function execution")
    print(id(inner))
    print("outer function returning inner function")
    return inner
f1=outer()      # f1 is pointing to inner function.
print(id(f1))
print(type(f1))
f1()            # Now directly 'inner()' function is calling
f1()
f1()
```

```
outer function started
1581911874824
outer function returning inner function
1581911874824
<class 'function'>
inner function execution
inner function execution
inner function execution
```

Eg:

```
def outer():
    print("outer function started")
    def inner():
        print("inner function execution")
    print("outer function returning inner function")
    return inner()        # inner() is not returning anything, so you will get 'None‘


f1=outer()            # f1 is pointing to inner function.
print(f1)
```

```
outer function started
outer function returning inner function
inner function execution
None
```

Q. What is the difference between the following lines?

f1 = outer

f1 = outer()

- In the first case for the outer() function we are providing another name f1(function aliasing).
- But in the second case we calling outer() function, which returns inner() function. For that inner() function we are providing another name 'f1'.

9.FRUITFUL FUNCTIONS

Fruitful functions and void functions

- Some of the functions we are using, such as the math functions, which yield results, we call them as **fruitful functions**.
- Other functions, like print_twice(), perform an action but don't return a value. They are called **void functions**.
- When you call a fruitful function, you almost always want to do something with the result;
- For example, you might assign it to a variable (or) use it as part of an expression.

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

- When you call a function in interactive mode, Python displays the result:

```
>>> math.sqrt(5)
2.23606797749979
```

- But in a script, if you call a fruitful function and do not store the result of the function in a variable, the return value vanishes into the mist!

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

- This script computes the square root of 5, but since it doesn't store the result in a variable or display the result, it is not very useful.

- Void functions might display something on the screen or have some other effect, but they don't have a return value. If you try to assign the result to a variable, you get a special value called None.

Eg:

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

- The value None is not the same as the string "None". It is a special value that has its own type:

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

- To return a result from a function, we use the return statement in our function.
- For example, we could make a very simple function called addtwo() that adds two numbers together and returns a result.

```
def addtwo(a, b):
    added = a + b
    return added
x = addtwo(3, 5)
print(x)
```

- When this script executes, the print statement will print out "**8**" because the addtwo function was called with **3** and **5** as arguments. Within the function, the parameters **a** and **b** were **3** and **5** respectively.
- The function computed the sum of the two numbers and placed it in the local function variable named **added**. Then it used the return statement to send the computed value back to the calling code as the function result, which was assigned to the variable **x** and printed out.

CONCLUSIONS

Why functions?

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
- Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
- Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
- Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Debugging

- If you are using a text editor to write your scripts, you might run into problems with spaces and tabs. The best way to avoid these problems is to use spaces exclusively (no tabs). Most text editors that know about Python do this by default, but some don't.
- Tabs and spaces are usually invisible, which makes them hard to debug, so try to find an editor that manages indentation for you.
- Also, don't forget to save your program before you run it. Some development environments do this automatically, but some don't. In that case, the program you are looking at in the text editor is not the same as the program you are running.

- Debugging can take a long time if you keep running the same incorrect program over and over!
- Make sure that the code you are looking at is the code you are running. If you're not sure, put something like print("hello") at the beginning of the program and run it again. If you don't see hello, you're not running the right program!

Glossary

- **algorithm** A general process for solving a category of problems.
- **argument** A value provided to a function when the function is called. This value is assigned to the corresponding parameter in the function.
- **body** The sequence of statements inside a function definition.
- **composition** Using an expression as part of a larger expression, or a statement as part of a larger statement.
- **deterministic** Pertaining to a program that does the same thing each time it runs, given the same inputs.
- **dot notation** The syntax for calling a function in another module by specifying the module name followed by a dot (period) and the function name.
- **flow of execution** The order in which statements are executed during a program run.
- **fruitful function** A function that returns a value.
- **function** A named sequence of statements that performs some useful operation. Functions may or may not take arguments and may or may not produce a result.

- **function call** A statement that executes a function. It consists of the function name followed by an argument list.
- **function definition** A statement that creates a new function, specifying its name, parameters, and the statements it executes.
- **function object** A value created by a function definition. The name of the function is a variable that refers to a function object.
- **header** The first line of a function definition.
- **import statement** A statement that reads a module file and creates a module object.
- **module object** A value created by an import statement that provides access to the data and code defined in a module.
- **parameter** A name used inside a function to refer to the value passed as an argument.
- **return value** The result of a function. If a function call is used as an expression, the return value is the value of the expression.
- **void function** A function that does not return a value.

Any question?

If you try to practice programs yourself, then you will learn many things automatically

Spend few minutes and then enjoy the study

Thank You