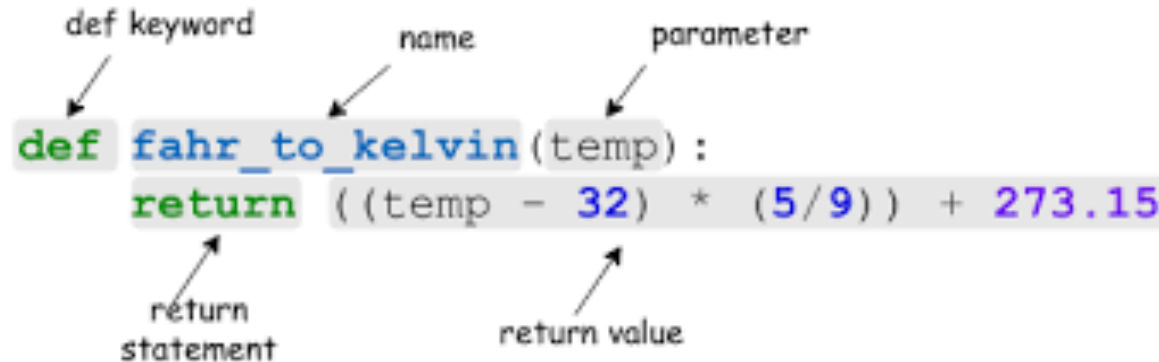


Function and Module

Python basic for everyone

24/06/2019

Functions



A diagram illustrating the components of a Python function definition. The code is: `def fahr_to_kelvin(temp):`
 `return ((temp - 32) * (5/9)) + 273.15`
Labels with arrows point to specific parts: 'def keyword' points to 'def', 'name' points to 'fahr_to_kelvin', 'parameter' points to '(temp)', 'return statement' points to 'return', and 'return value' points to the expression '((temp - 32) * (5/9)) + 273.15'.

A function is a piece of code in a program. The function performs a specific task. The advantages of using functions are:

- Reducing duplication of code
- Decomposing complex problems into simpler pieces
- Improving clarity of the code
- Reuse of code
- Information hiding

Defining Functions

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 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`.

```
>>> def my_func(x, y, z):  
...     a = x + y  
...     b = a * z  
...     return b  
...  
>>>
```

```
>>> my_func(1.0, 3.0, 2.0)  
8.0  
>>> my_func(1.0, 3.0, 1.0)  
4.0  
>>> my_func(5.0, 0.0, 1.0)  
5.0  
>>> my_func(2.0, 0, 3.0)  
6.0
```

Defining Functions

Function must be defined preceding their usage:

```
def f1():  
    print "f1() "  
  
f1()  
#f2()  
  
def f2():  
    print "f2() "
```

uncommenting f2()
will cause a NameError

Functions types

- always available for usage
- those contained in external modules
- programmer defined

```
>>> from math import sqrt
>>> def cube(x):
...     return x * x * x
...
>>> print abs(-1)
1
>>> print cube(9)
729
>>> print sqrt(81)
9.0
```

The return keyword

- is used to return value
- no return returns None

```
>>> n = [1, 2, 3, 4, 5]
>>> def stats(x):
...     mx = max(x)
...     mn = min(x)
...     ln = len(x)
...     sm = sum(x)
...
...     return mx, mn, ln, sm
...
>>> mx, mn, ln, sm = stats(n)
>>> print stats(n)
(5, 1, 5, 15)
>>>
>>> print mx, mn, ln, sm
5 1 5 15
```

```
>>> def cube(x):
...     return x * x * x
...
>>> def showMessage(msg):
...     print msg
...
>>> x = cube(3)
>>> print x
27
>>> showMessage("Some text")
Some text
>>> print showMessage("O, no!")
O, no!
None
>>> showMessage(cube(3))
27
>>>
```

Recursion functions

```
>>> def fact(n):  
...     if(n==0): return 1;  
...     m = 1;  
...     k = 1;  
...     while(n >= k):  
...         m = m * k;  
...         k = k + 1;  
...     return m;
```

Recursion:

```
>>> def fact(n):  
...     if n > 0:  
...         return n * fact(n-1)    # Recursive call  
...     return 1                    # exits function returning 1  
>>> print fact(100)  
>>> print fact(1000)
```

Function arguments

single arguments

```
>>> def C2F(c):  
...     return c * 9/5 + 32  
...  
>>>  
>>> print C2F(100)  
212  
>>> print C2F(0)  
32  
>>> print C2F(30)  
86  
>>>
```

multiple arguments

```
>>> def power(x, y=2):  
...     r = 1  
...     for i in range(y):  
...         r = r * x  
...     return r  
...  
>>> print power(3)  
9  
>>> print power(3, 3)  
27  
>>> print power(5, 5)  
3125  
>>>
```


Function arguments

named arguments

- order may be changed
- default value

```
>>> def display(name, age, sex):  
...     print("Name: ", name)  
...     print("Age: ", age)  
...     print("Sex: ", sex)  
...  
>>> display(age=43, name="Lary", sex="M")  
Name:  Lary  
Age:   43  
Sex:   M  
>>> display(name="Joan", age=24, sex="F")  
Name:  Joan  
Age:   24  
Sex:   F  
>>> display("Joan", sex="F", age=24)  
Name:  Joan  
Age:   24  
Sex:   F  
>>> display(age=24, name="Joan", "F")  
File "<stdin>", line 1  
SyntaxError: non-keyword arg after keyword arg  
>>>
```

Function arguments

arbitrary number of
arguments

```
>>> def sum(*args):  
...     '''Function returns the sum  
...     of all values'''  
...     s = 0  
...     for i in args:  
...         s += i  
...     return s  
...  
>>>  
>>> print sum.__doc__  
Function returns the sum  
    of all values  
>>> print sum(1, 2, 3)  
6  
>>> print sum(1, 2, 3, 4, 5)  
15  
>>>
```

Function arguments

passing by reference

Passing objects by reference has two important conclusions. The process is faster than if copies of objects were passed. Mutable objects that are modified in functions are permanently changed.

```
>>> n = [1, 2, 3, 4, 5]
>>>
>>> print "Original list:", n
Original list: [1, 2, 3, 4, 5]
>>>
>>> def f(x):
...     x.pop()
...     x.pop()
...     x.insert(0, 0)
...     print "Inside f():", x
...
...
>>> f(n)
Inside f(): [0, 1, 2, 3]
>>>
>>> print "After function call:", n
After function call: [0, 1, 2, 3]
>>>
```

Function variables

Global and Local

A variable defined in a function body has a local scope

We can get the contents of a global variable inside the body of a function. But if we want to change a global variable in a function, we must use the global keyword.

```
>>> name = "Jack"
>>> def f():
...     name = "Robert"
...     print "Within function", name
...
>>> print "Outside function", name
Outside function Jack
>>> f()
Within function Robert
>>> def f():
...     print "Within function", name
...
>>> print "Outside function", name
Outside function Jack
>>> f()
Within function Jack
```

```
>>> name = "Jack"
>>> def f():
...     global name
...     name = "Robert"
...     print "Within function", name
...
>>> print "Outside function", name
Outside function Jack
>>> f()
Within function Robert
>>> print "Outside function", name
Outside function Robert
>>>
```

Functions

The **Anonymous** Functions:

You can use the **lambda** keyword to create small anonymous functions. These functions are called anonymous because they are not declared by using the *def* keyword.

- Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
- An anonymous function cannot be a direct call to print because lambda requires an expression.
- Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.

```
# 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 )

Value of total : 30
Value of total : 40
```

