

Functions

Chapter 9

This chapter covers

- Defining functions
- Using function parameters
- Passing mutable objects as parameters
- Understanding local and global variables
- Creating and using generator functions
- Creating and using lambda expressions
- Using decorators

Basic function definitions

```
1  # syntax
2  # def name(param1, param2,...):
3  #     body
4
5  # function that will return the factorial
6  def fact(n):
7      """ Return the factorial of the given number """
8      r = 1
9      while n > 0:
10         r = r * n
11         n = n - 1
12     return r
13
14 print(fact(4))
```

Function with parameters

```
16  # function that will return the power
17  def power(x, y):
18      r = 1
19      while y > 0:
20          r = r * x
21          y = y - 1
22      return r
23
```

Function with default parameter

```
# function with default par
def power2(x, y=2):
    r = 1
    while y > 0:
        r = r * x
        y = y - 1
    return r
```

```
print(power2(3, 3))
print(power2(3))
```

```
26 # function with default param
27 def power2(x, y=2):
28     r = 1
29     while y > 0:
30         r = r * x
31         y = y - 1
32     return r
33
34 print(power2(3, 3))
35 print(power2(3))
36
37 # named parameters
38 print(power2(y=4, x =2))
```

Named parameter

Variable numbers of arguments

```
40  # varargs
41  def maximum(*numbers):
42      if len(numbers) == 0:
43          return None
44      else:
45          maxnum = numbers[0]
46          for n in numbers[1:]:
47              if n > maxnum:
48                  maxnum = n
49          return maxnum
50
51  print(maximum(3,2,8))
52  print(maximum(1,5,9,-2,2))
```

Indefinite number of arguments

```
54 # indefinite number of arguments passed by keyword
55 def example_fun(x,y,**other):
56     print("x: {0}, y: {1}, keys in 'other': {2}".format(x,y,list(other.keys())))
57     other_total = 0
58     for k in other.keys():
59         other_total = other_total + other[k]
60     print("The total of values in 'other' is {0}".format(other_total))
61
62 example_fun(2,y="1", foo=3,bar=4)
```


Mutable objects as arguments

```
64 # mutable objects as arguments
65 def f(n, list1, list2):
66     list1.append(3)
67     list2 = [4,5,6]
68     n = n + 1
69
70 x = 5
71 y = [1,2]
72 z = [4,5]
73 f(x,y,z)
74 print(x, y, z) # y is modified
```

Function as parameter

```
76 # assigning functions to variables
77 def f_to_kelvin(degrees_f):
78     return 273.15 + (degrees_f - 32) * 5 / 9
79
80 def c_to_kelvin(degrees_c):
81     return 273.15 + degrees_c
82
83 abs_temperature = f_to_kelvin
84 print(abs_temperature(32))
85
86 abs_temperature = c_to_kelvin
87 print(abs_temperature(0))
88
89 # place them in a list, tuples or dictionaries
90 t = {'FtoK':f_to_kelvin, 'CtoK':c_to_kelvin}
91 print(t['FtoK'](32))
92 print(t['CtoK'](0))
```

Lambda Expressions

- lambda expressions are anonymous little functions that you can quickly define inline.

```
1  ✓ t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,  
2      |      'CtoK': lambda deg_c: 273.15 + deg_c}  
3  
4  result = t2['FtoK'](32)  
5  print(result)
```

Generator functions

- A generator function is a special kind of function that you can use to define your own iterators.
- When you define a generator function, you return each iteration's value using the yield keyword.
- The generator will stop returning values when there are no more iterations, or it encounters either an empty return statement or the end of the function.
- Local variables in a generator function are saved from one call to the next, unlike in normal functions

```
1  ✓ def four():
2      x = 0
3  ✓  while x < 4:
4      |     print("in generator, x =", x)
5      |     yield x
6      |     x += 1
7
8  ✓ for i in four():
9      |     print(i)
```

Decorators

- A decorator is syntactic sugar for this process and lets you wrap one function inside another with a one-line addition.
- It still gives you exactly the same effect as the previous code, but the resulting code is much cleaner and easier to read.

```
1 def decorate(func):
2     print("in decorate function, decorating", func.__name__)
3     def wrapper_func(*args):
4         print("Executing", func.__name__)
5         return func(*args)
6     return wrapper_func
7
8 def myfunction(parameter):
9     print(parameter)
10
11 myfunction = decorate(myfunction)
12
13 myfunction("hello")
```

Lab

Useful functions

Summary

- External variables can easily be accessed within a function by using the global statement.
- Arguments may be passed by position or by parameter name.
- Default values may be provided for function parameters.
- Functions can collect arguments into tuples, giving you the ability to define functions that take an indefinite number of arguments.
- Functions can collect arguments into dictionaries, giving you the ability to define functions that take an indefinite number of arguments passed by parameter name.
- Functions are first-class objects in Python, which means that they can be assigned to variables, accessed by way of variables, and decorated.