Functions

--created by David Song

Contents

Parameter Passing

When values are passed into functions, if they are immutable, they are passed by-value, but if they are mutable, then they are passed by-reference. Since scalar values are cached for performance reasons, they are considered immutable. Strings and tuples are also immutable.

Default Parameters

The values of default parameters are set when the function is defined. This means when the module (in which this function resides) gets loaded, the default values are set to what the function (actually the interpreter) can see at this point, and cannot be changed even if the default value are defined as a type or a constant which could be redefined later in the same module.

Positional, Keyword, and Multiple Parameters

Positional Parameters can be passed in by order:

```
def display_info(name, age, spouse):
    print name, age, spouse

display_info('Rob', 45, 'Sally')
```

Keyword parameters make it easy to read function definitions.

```
display_info(name='Bob', age=37, spouse='Sally')
```

When positional and keyword parameters are mixed, keyword parameters must come as last.

```
display_info('Rob', 45, spouse='Sally')
```

A multiple parameter is the number of parameters is unknown when the function is defined until runtime. The asterisk (*) is used to define a multi-parameter argument. Multiple keyword parameters can be defined using double asterisks (**).

```
def display_info (name, age, spouse, *children,**friends):
    print name, age, spouse, children, friends

display_info('Rob',     45,     'Sally',     'Sid','Adam',friend1='Kathy',
friend2='John')

output:
Rob 45 Sally ('Sid', 'Adam') {'friend2': 'John', 'friend1': 'Kathy'}
```

The rule here: it will scan positional, keyword parameters, take the remaining as a tuple, until it sees the keyword parameters which will be taken into the double asterisks parameters as a dictionary.

When an asterisk is placed in the call to a function, it implies the ‘spreading’ or ‘dispersing’ of collections into the defined function parameters.

```
display_info('Rob', 45, spouse='Sally', *('Sid','Adam'),
**{'friend1':'Kathy', 'friend2':'John'})
```

This also valid:

```
display_info(*('Sid',45),**{'spouse':'Sally'})

*() implies disperse those into positional parameters;
**{} implies disperse into keyword parameters.
```

Caution: it’s a bad practice to use a combination of positional parameters and wild card parameters. This needs to be make sure that each method call, positional parameters should be put in there, followed by each wild car parameters. It’s easy to cause runtime exceptions.

For example:

```
display_info(*('Sid',45),**{'spouse':'Sally'})
```

above method call will fill only the first 3 parameters and ignore the last two wild card parameters. Coincidently the ** multiple keyword parameter has the same keyword spouse as the method signature. If we change the keyword, it will throw an exception, like below:

```
display_info(*('Sid',45),**{'Nadia':'Sally'})
```

Change the method signature to only wild card parameter should work, like below:

```
def display_info ( *children,**friends):
    print children, friends

display_info(*('Sid',45),**{'Nadia':'Sally'})
```

output:

```
('Sid', 45) {'Nadia': 'Sally'}
```

Variable Scope

Global refers to module scope. When a global variable gets assigned a value inside a function, a new local variable will be created with the same name. To refer to a global variable inside a function, use keyword global in front of the variable name in the function.

Lambdas

Lambdas are functions that can be used as variables or as higher order objects.

```
fun_name= lambda <arguments>:<expression>

example: myfunc=lambda a, b: a+b
```

Generator

A generator is a special function that when called does not actually begin executing the code within the function. Instead, they return an *iterator object*. A generator is automatically created whenever a *yield* statement is found within that function. Generators retrieve results in a lazy fashion.

Below example shows a simple generator:

```
def generator_1(x):
    for i in range(x):
        yield i
```

```
g = generator_1(5)
print g.next()       # print 0

for i in g:
    print i          # print 1,2,3,4
```

Below code shows to achieve the same as above:

```
def generator_2():
    yield 0
    yield 1
    yield 2
    yield 3
    yield 4

g = generator_2()
for i in g:
    print i  # print 0,1,2,3,4
```

Generally speaking, a generator returns an *iterator*. when the *next()* method is called for the first time, the function runs up to the yield statement and returns the value associate with it, and then suspend its execution, until the *next* method gets called again.

Generator is very useful for handling large dat, e.g. read from files, database, etc. since it can use little memory.

Below shows how to use generator to read a file:

```
def read_file_generator(file_path, size):

    with open(file_path) as f:
        data = f.read(size)
        while data:
            yield data
            data = f.read(size)


input_file=r'sample.dat'
for idx, data in enumerate(read_file_generator(input_file,1024)):
    print('{}:{}'.format(idx, data))
```

Luckily, most of Python read file and database operations are implemented in lazy fashion, so you don't need to write your own generator to wrap those operations; just keep in mind when you need it.

Coroutines

Coroutines are *generators* that allow data to be **sent** into the function. The send() method can be used to pass data into the function (to the *yield*).

```
def grep(regex):
    print "Searching for: %s" % regex
    while True:
        #
        # yield is waiting for input from send method
        # yield is on RHS
        #
        input = (yield)
        print('input:{}'.format(input))
        if regex in input:
            print('Found: regex:{} in input:{}'.format(regex, input))
        else:
            print('Not Found: regex:{} in input:{}'.format(regex, input))


#
# set the regex variable
# function will not run until next() is called,
# because of its lazy fashion
#
g = grep('happy')

#
# runs into the yield statement
# must be called before calling send
#
g.next()

#
# set input=(yield) with whatever in the send parameter
# yield is waiting for input from send method
#
g.send('Happy happy birthday')              # set input='Happy happy birthday'
g.send('Happiness is a state of mind.')
g.send('Stay happy with Python.')
```

In simple words, a generator is a function implemented in a lazy fashion via the yield statement, a coroutine is a generator takes input. The difference is generator return result with *yield* result; while coroutine use *yield* to take input and assign it to a local variable inside the coroutine.

Built-in Functions

http://docs.python.org/library/functions.html

`map`(*function*, *iterable*, ...)

> Apply *function* to every item of *iterable* and return a list of the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel.
>
> ```
> list1=[1,2,3]
> list2=[2,3,4]
> map(str, list1)  → ['1', '2', '3']
> ```

```
map(None, list1) → [1, 2, 3]
map(None, list1, list2) → [(1, 2), (2, 3), (3, 4)]
```

`enumerate`(*sequence*[, *start=0*])

Return an enumerate object. The *sequence* parameter must be a sequence, an *iterator*, or some other object which supports iteration. The `next()` method of the iterator returned by `enumerate()` returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *sequence*.

```
seasons = ['Spring', 'Summer', 'Fall', 'Winter']

for item in enumerate(seasons):
    print item

(0, 'Spring')
(1, 'Summer')
(2, 'Fall')
(3, 'Winter')
```