Python Overview Part 2

RAVINDRA KUDACHE

Session 4

RAVINDRA KUDACHE

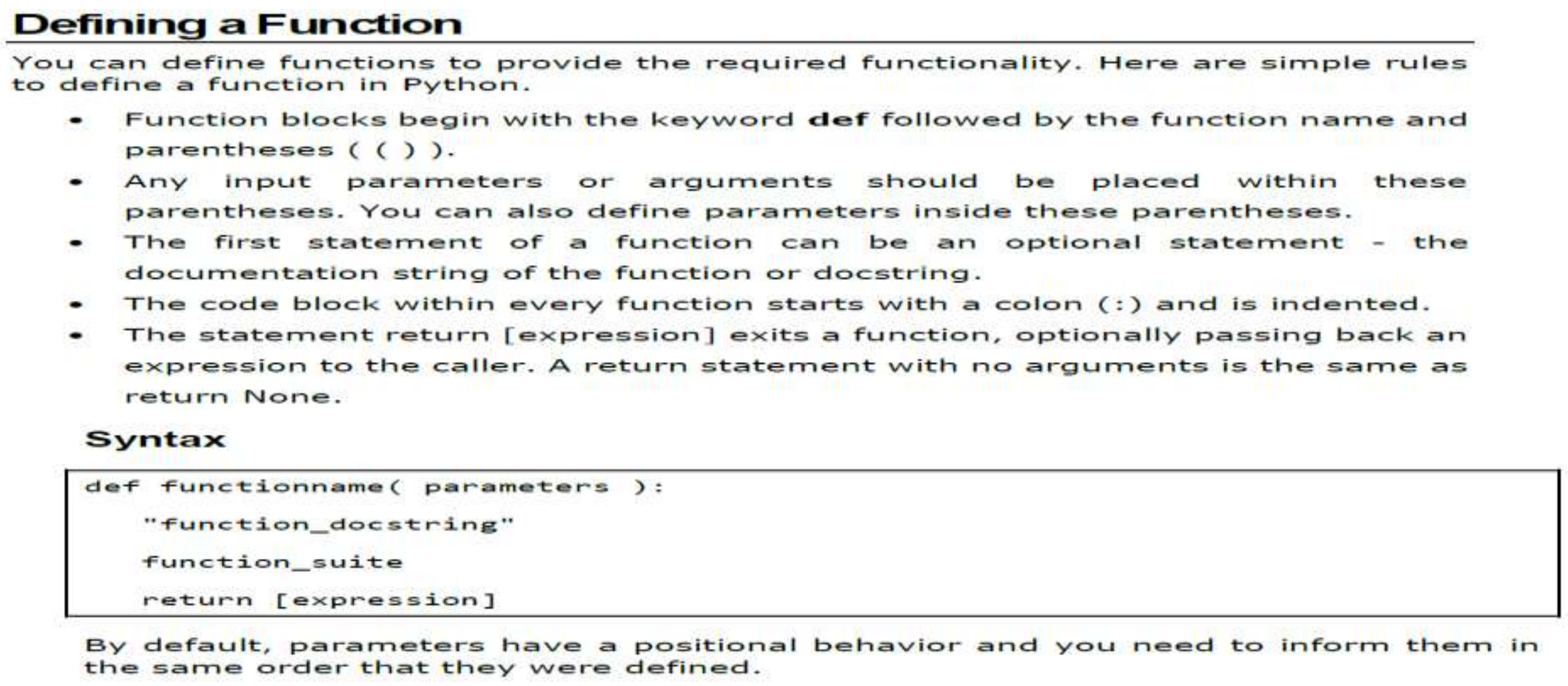
FUNCTIONS

Why functions?

It may not be clear why it is worth the trouble to divide a program into functions. There are several reasons:

- Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read 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.

FUNCTIONS



Defining a Function

You can define functions to provide the required functionality. 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 first statement of a function can be an optional statement - the documentation string of the function or docstring.
- 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.

Syntax

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

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

FUNCTIONS

Function calls

We have already seen one example of a **function call**:

```
>>> type(42)
<class 'int'>
```

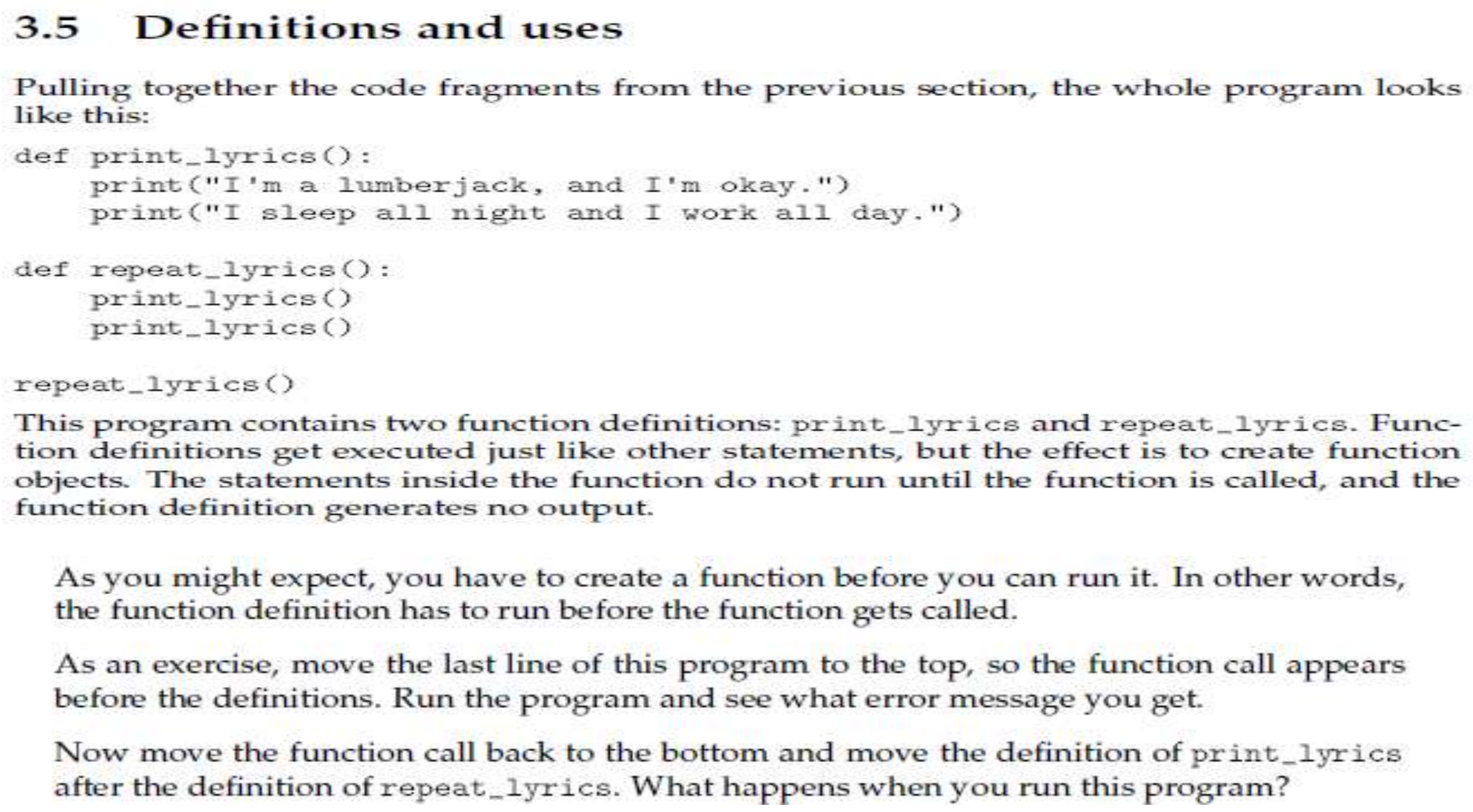
The name of the function is type. The expression in parentheses is called the **argument** of
the function. The result, for this function, is the type of the argument.
It is common to say that a function "takes" an argument and "returns" a result. The result is also called the **return value**.
Python provides functions that convert values from one type to another. The int function takes any value and converts it to an integer, if it can, or complains otherwise:

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int can convert floating-point values to integers, but it doesn't round off; it chops off the fraction part:

```
>>> int(3.99999)
```

FUNCTIONS



3.5 Definitions and uses

Pulling together the code fragments from the previous section, the whole program looks like this:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

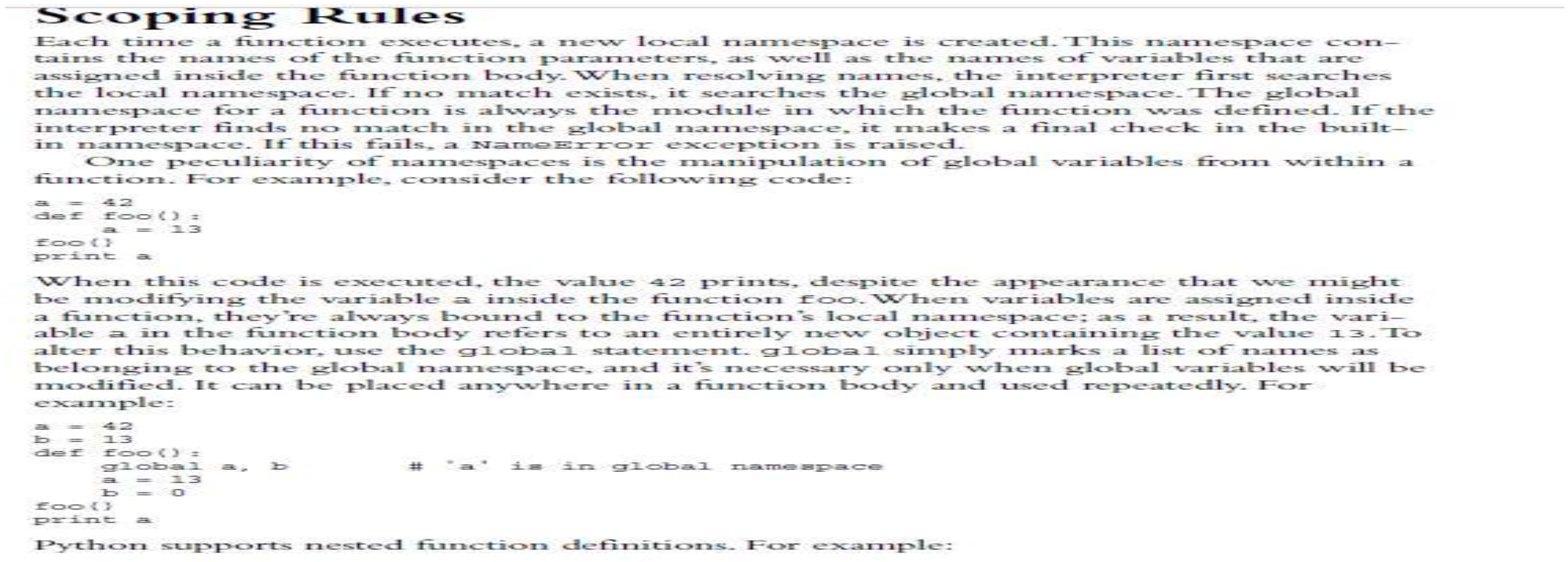
This program contains two function definitions: `print_lyrics` and `repeat_lyrics`. Function definitions get executed just like other statements, but the effect is to create function objects. The statements inside the function do not run until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can run it. In other words, the function definition has to run before the function gets called.

As an exercise, move the last line of this program to the top, so the function call appears before the definitions. Run the program and see what error message you get.

Now move the function call back to the bottom and move the definition of `print_lyrics` after the definition of `repeat_lyrics`. What happens when you run this program?

FUNCTIONS



Scoping Rules

Each time a function executes, a new local namespace is created. This namespace contains the names of the function parameters, as well as the names of variables that are assigned inside the function body. When resolving names, the interpreter first searches the local namespace. If no match exists, it searches the global namespace. The global namespace for a function is always the module in which the function was defined. If the interpreter finds no match in the global namespace, it makes a final check in the built-in namespace. If this fails, a `NameError` exception is raised.

One peculiarity of namespaces is the manipulation of global variables from within a function. For example, consider the following code:

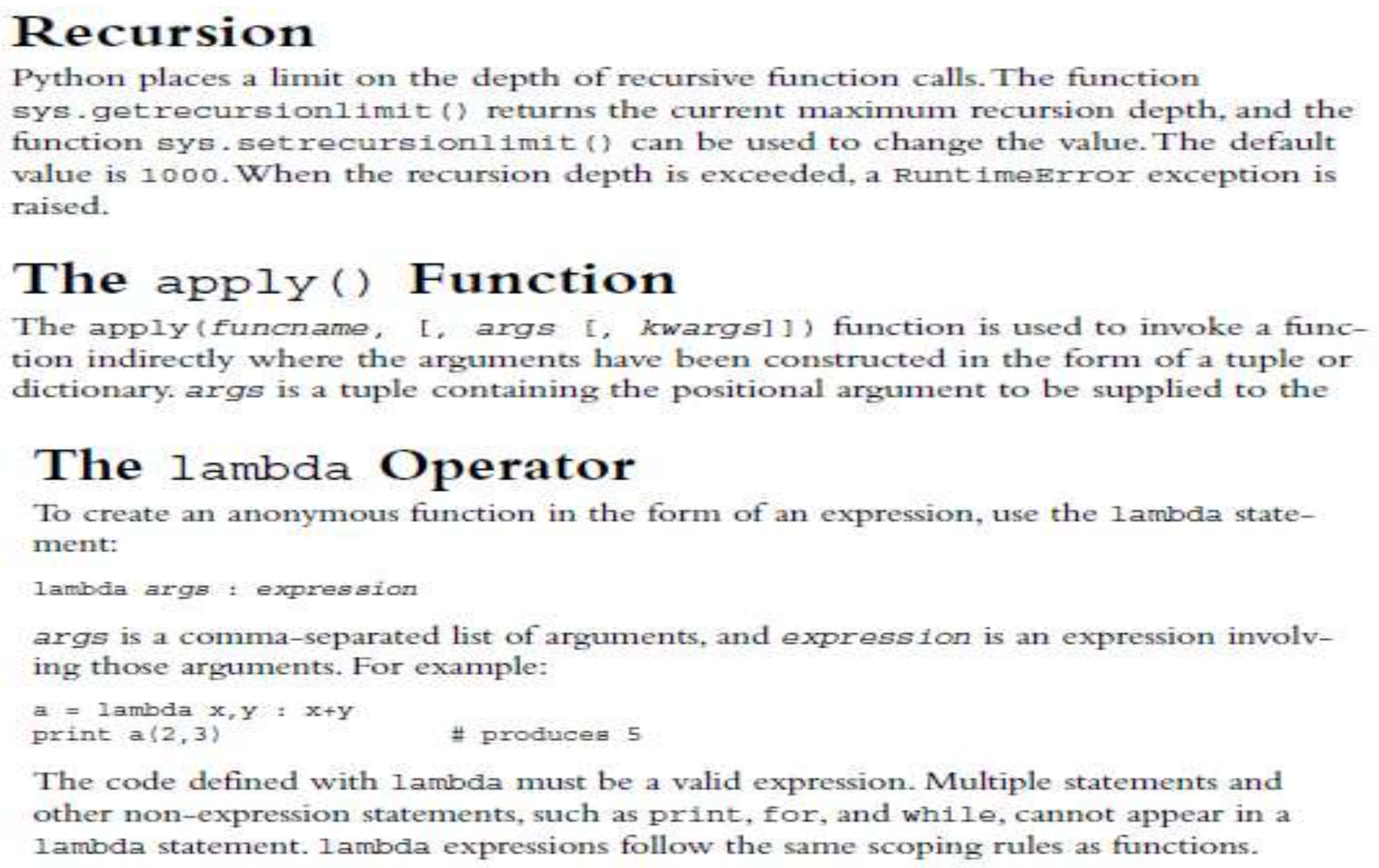
```
a = 42
def foo():
    a = 13
foo()
print a
```

When this code is executed, the value 42 prints, despite the appearance that we might be modifying the variable `a` inside the function `foo`. When variables are assigned inside a function, they're always bound to the function's local namespace; as a result, the variable `a` in the function body refers to an entirely new object containing the value 13. To alter this behavior, use the `global` statement. `global` simply marks a list of names as belonging to the global namespace, and it's necessary only when global variables will be modified. It can be placed anywhere in a function body and used repeatedly. For example:

```
a = 42
b = 13
def foo():
    global a, b          # 'a' is in global namespace
    a = 13
    b = 0
foo()
print a
```

Python supports nested function definitions. For example:

FUNCTIONS



Recursion

Python places a limit on the depth of recursive function calls. The function `sys.getrecursionlimit()` returns the current maximum recursion depth, and the function `sys.setrecursionlimit()` can be used to change the value. The default value is `1000`. When the recursion depth is exceeded, a `RuntimeError` exception is raised.

The `apply()` Function

The `apply(`*`funcname`*`, [, `*`args`*` [, `*`kwargs`*`]])` function is used to invoke a function indirectly where the arguments have been constructed in the form of a tuple or dictionary. *`args`* is a tuple containing the positional argument to be supplied to the

The `lambda` Operator

To create an anonymous function in the form of an expression, use the `lambda` statement:

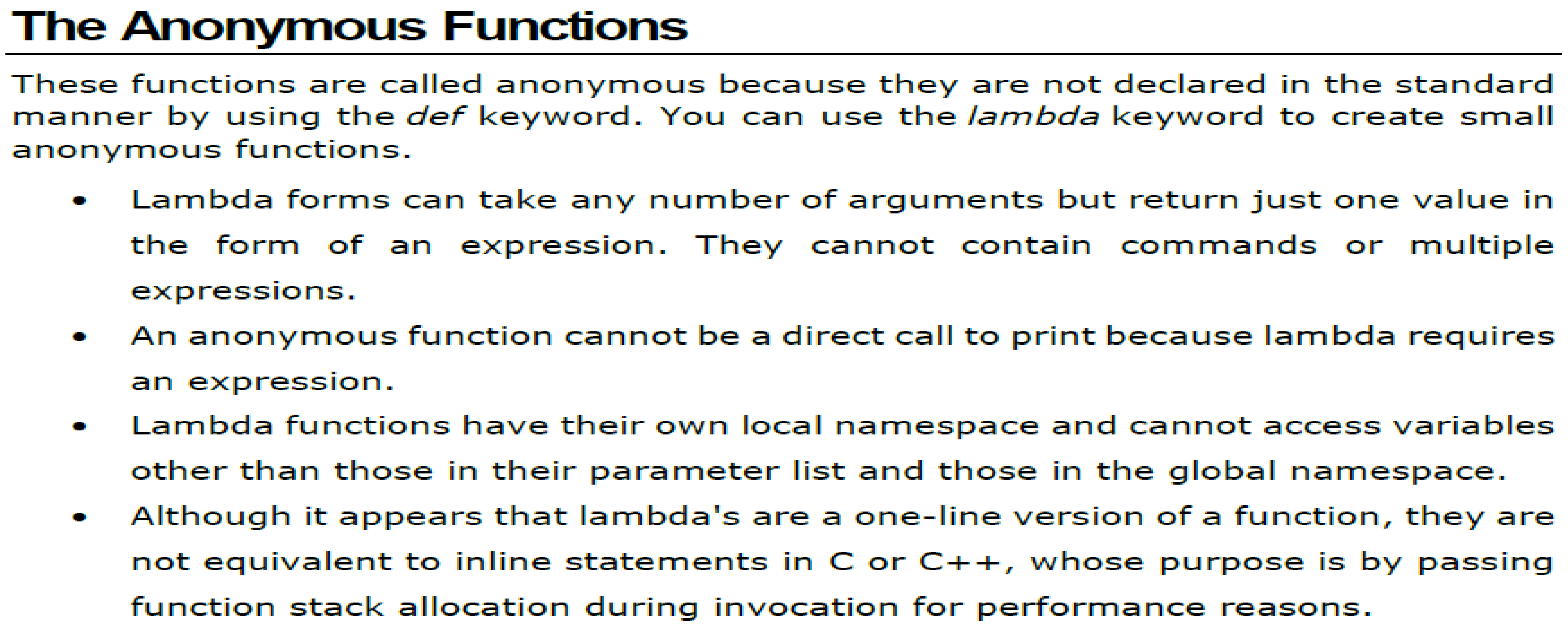
```
lambda args : expression
```

`args` is a comma-separated list of arguments, and *`expression`* is an expression involving those arguments. For example:

```
a = lambda x,y : x+y
print a(2,3)              # produces 5
```

The code defined with `lambda` must be a valid expression. Multiple statements and other non-expression statements, such as `print`, `for`, and `while`, cannot appear in a `lambda` statement. `lambda` expressions follow the same scoping rules as functions.

FUNCTIONS

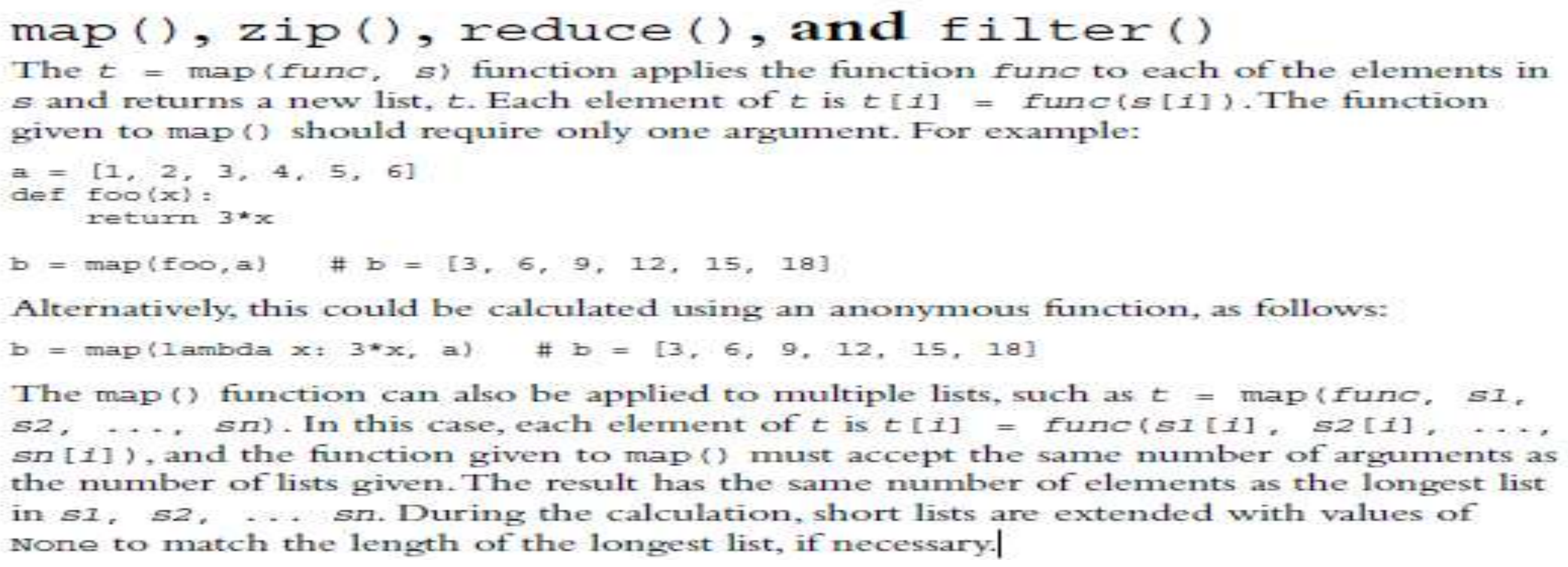


The Anonymous Functions

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

- 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.
- Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

FUNCTIONS



map(), zip(), reduce(), and filter()

The `t = map(func, s)` function applies the function *func* to each of the elements in *s* and returns a new list, *t*. Each element of *t* is `t[i] = func(s[i])`. The function given to `map()` should require only one argument. For example:

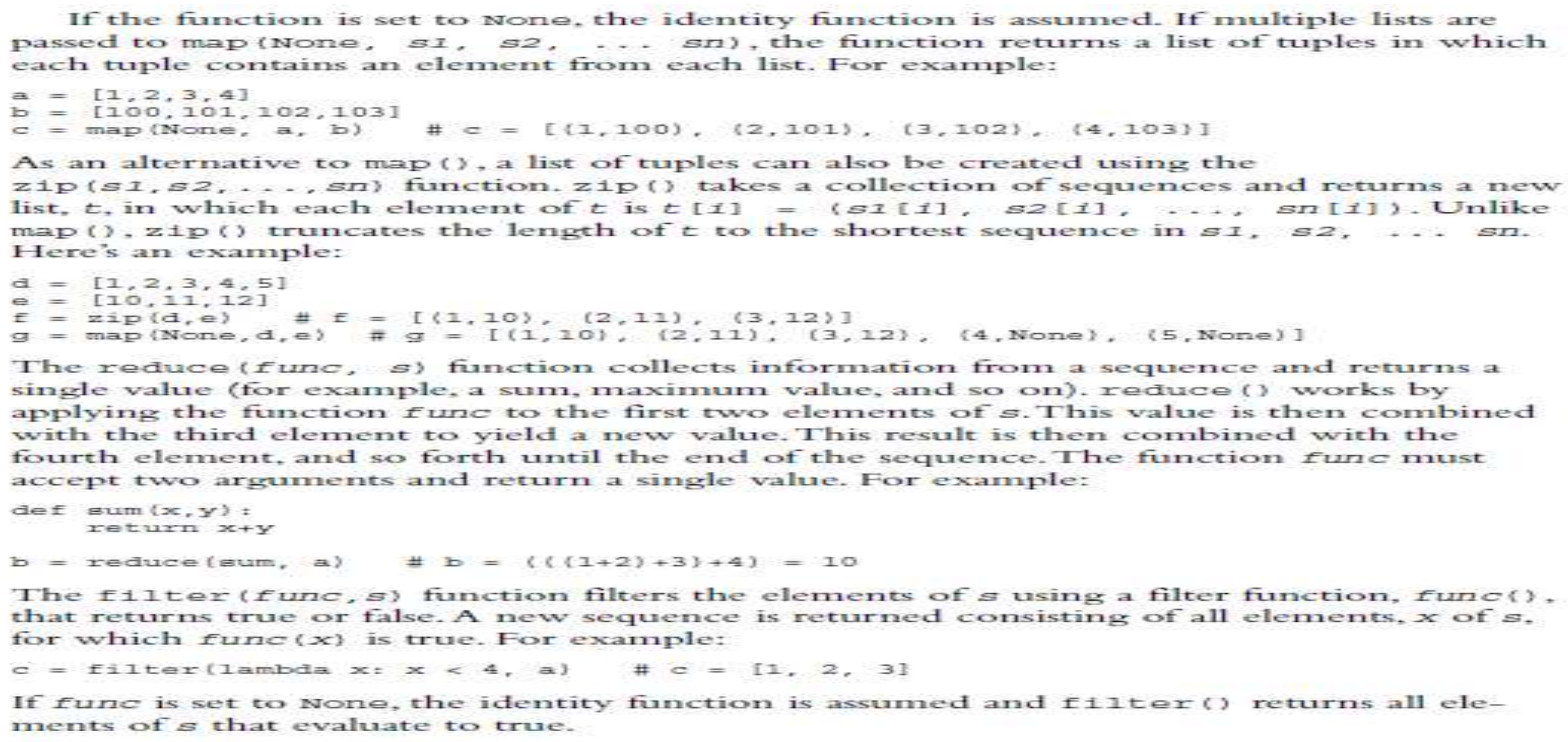
```
a = [1, 2, 3, 4, 5, 6]
def foo(x):
    return 3*x

b = map(foo,a)    # b = [3, 6, 9, 12, 15, 18]
```

Alternatively, this could be calculated using an anonymous function, as follows:

```
b = map(lambda x: 3*x, a)    # b = [3, 6, 9, 12, 15, 18]
```

The `map()` function can also be applied to multiple lists, such as `t = map(func, s1, s2, ..., sn)`. In this case, each element of *t* is `t[i] = func(s1[i], s2[i], ..., sn[i])`, and the function given to `map()` must accept the same number of arguments as the number of lists given. The result has the same number of elements as the longest list in *s1, s2, ... sn*. During the calculation, short lists are extended with values of `None` to match the length of the longest list, if necessary.

FUNCTIONS



If the function is set to None, the identity function is assumed. If multiple lists are passed to map(None, *s1*, *s2*, ... *sn*), the function returns a list of tuples in which each tuple contains an element from each list. For example:

```
a = [1,2,3,4]
b = [100,101,102,103]
c = map(None, a, b)    # c = [(1,100), (2,101), (3,102), (4,103)]
```

As an alternative to map(), a list of tuples can also be created using the zip(*s1*,*s2*,...,*sn*) function. zip() takes a collection of sequences and returns a new list, *t*, in which each element of *t* is *t*[*i*] = (*s1*[*i*], *s2*[*i*], ..., *sn*[*i*]). Unlike map(), zip() truncates the length of *t* to the shortest sequence in *s1*, *s2*, ... *sn*. Here's an example:

```
d = [1,2,3,4,5]
e = [10,11,12]
f = zip(d,e)    # f = [(1,10), (2,11), (3,12)]
g = map(None,d,e)   # g = [(1,10), (2,11), (3,12), (4,None), (5,None)]
```

The reduce(*func*, *s*) function collects information from a sequence and returns a single value (for example, a sum, maximum value, and so on). reduce() works by applying the function *func* to the first two elements of *s*. This value is then combined with the third element to yield a new value. This result is then combined with the fourth element, and so forth until the end of the sequence. The function *func* must accept two arguments and return a single value. For example:

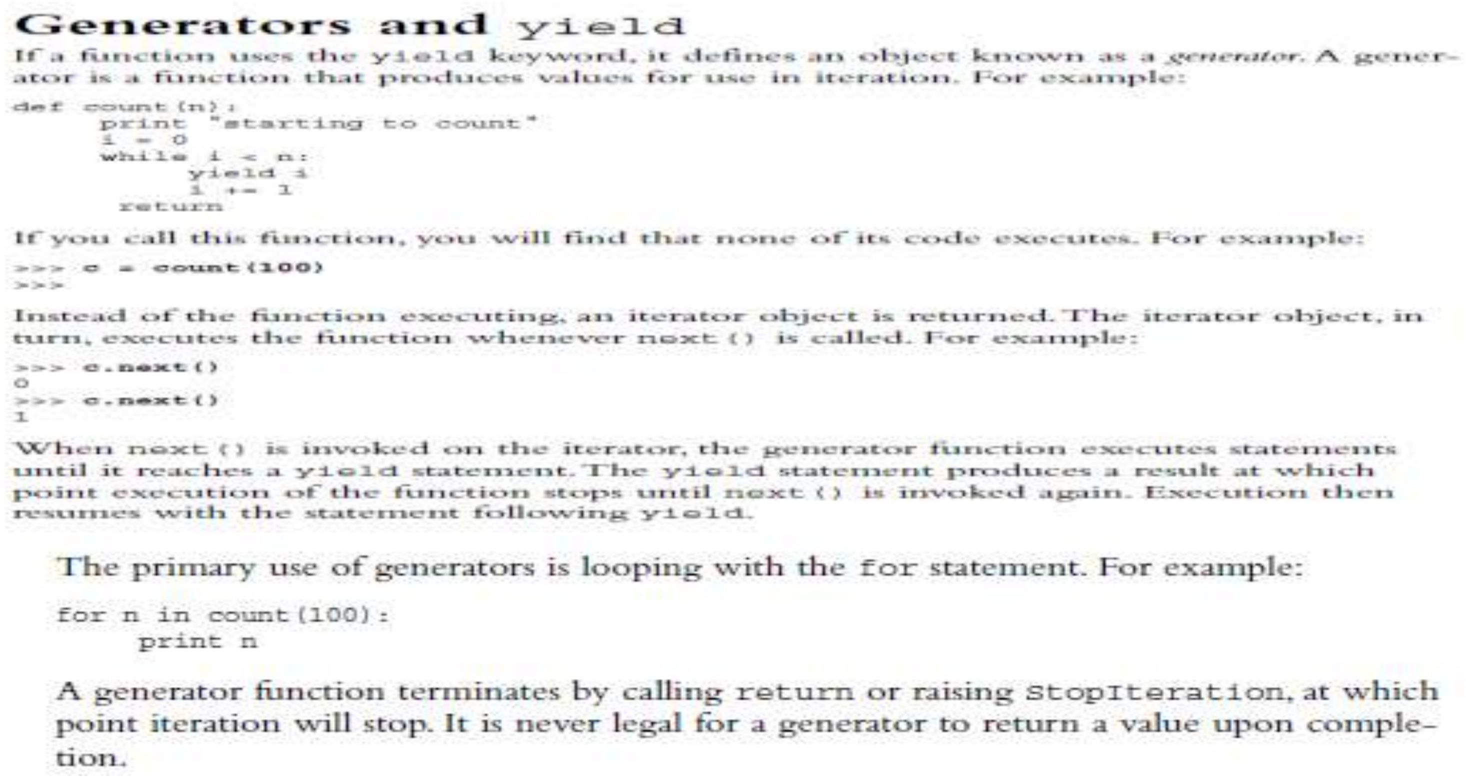
```
def sum(x,y):
    return x+y

b = reduce(sum, a)    # b = (((1+2)+3)+4) = 10
```

The filter(*func*,*s*) function filters the elements of *s* using a filter function, *func*(), that returns true or false. A new sequence is returned consisting of all elements, *x* of *s*, for which *func*(*x*) is true. For example:

```
c = filter(lambda x: x < 4, a)    # c = [1, 2, 3]
```

If *func* is set to None, the identity function is assumed and filter() returns all elements of *s* that evaluate to true.

FUNCTIONS



Generators and `yield`

If a function uses the `yield` keyword, it defines an object known as a *generator*. A generator is a function that produces values for use in iteration. For example:

```
def count(n):
     print "starting to count"
     i = 0
     while i < n:
          yield i
          i += 1
      return
```

If you call this function, you will find that none of its code executes. For example:

```
>>> c = count(100)
>>>
```

Instead of the function executing, an iterator object is returned. The iterator object, in turn, executes the function whenever `next()` is called. For example:

```
>>> c.next()
0
>>> c.next()
1
```

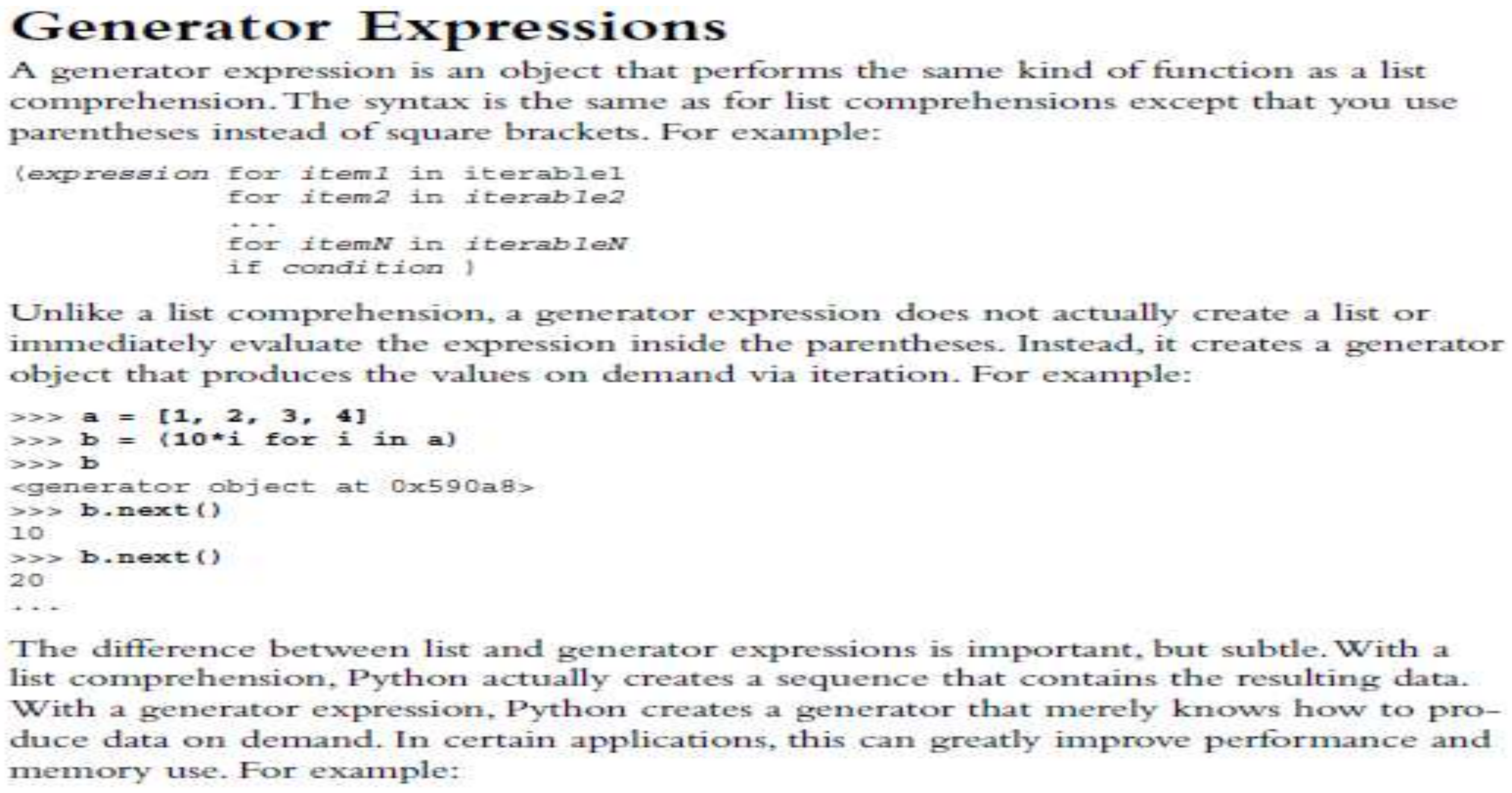
When `next()` is invoked on the iterator, the generator function executes statements until it reaches a `yield` statement. The `yield` statement produces a result at which point execution of the function stops until `next()` is invoked again. Execution then resumes with the statement following `yield`.

The primary use of generators is looping with the `for` statement. For example:

```
for n in count(100):
     print n
```

A generator function terminates by calling `return` or raising `StopIteration`, at which point iteration will stop. It is never legal for a generator to return a value upon completion.

FUNCTIONS



Generator Expressions

A generator expression is an object that performs the same kind of function as a list comprehension. The syntax is the same as for list comprehensions except that you use parentheses instead of square brackets. For example:

```
(expression for item1 in iterable1
            for item2 in iterable2
            ...
            for itemN in iterableN
            if condition )
```

Unlike a list comprehension, a generator expression does not actually create a list or immediately evaluate the expression inside the parentheses. Instead, it creates a generator object that produces the values on demand via iteration. For example:

```
>>> a = [1, 2, 3, 4]
>>> b = (10*i for i in a)
>>> b
<generator object at 0x590a8>
>>> b.next()
10
>>> b.next()
20
...
```

The difference between list and generator expressions is important, but subtle. With a list comprehension, Python actually creates a sequence that contains the resulting data. With a generator expression, Python creates a generator that merely knows how to produce data on demand. In certain applications, this can greatly improve performance and memory use. For example:

FUNCTIONS

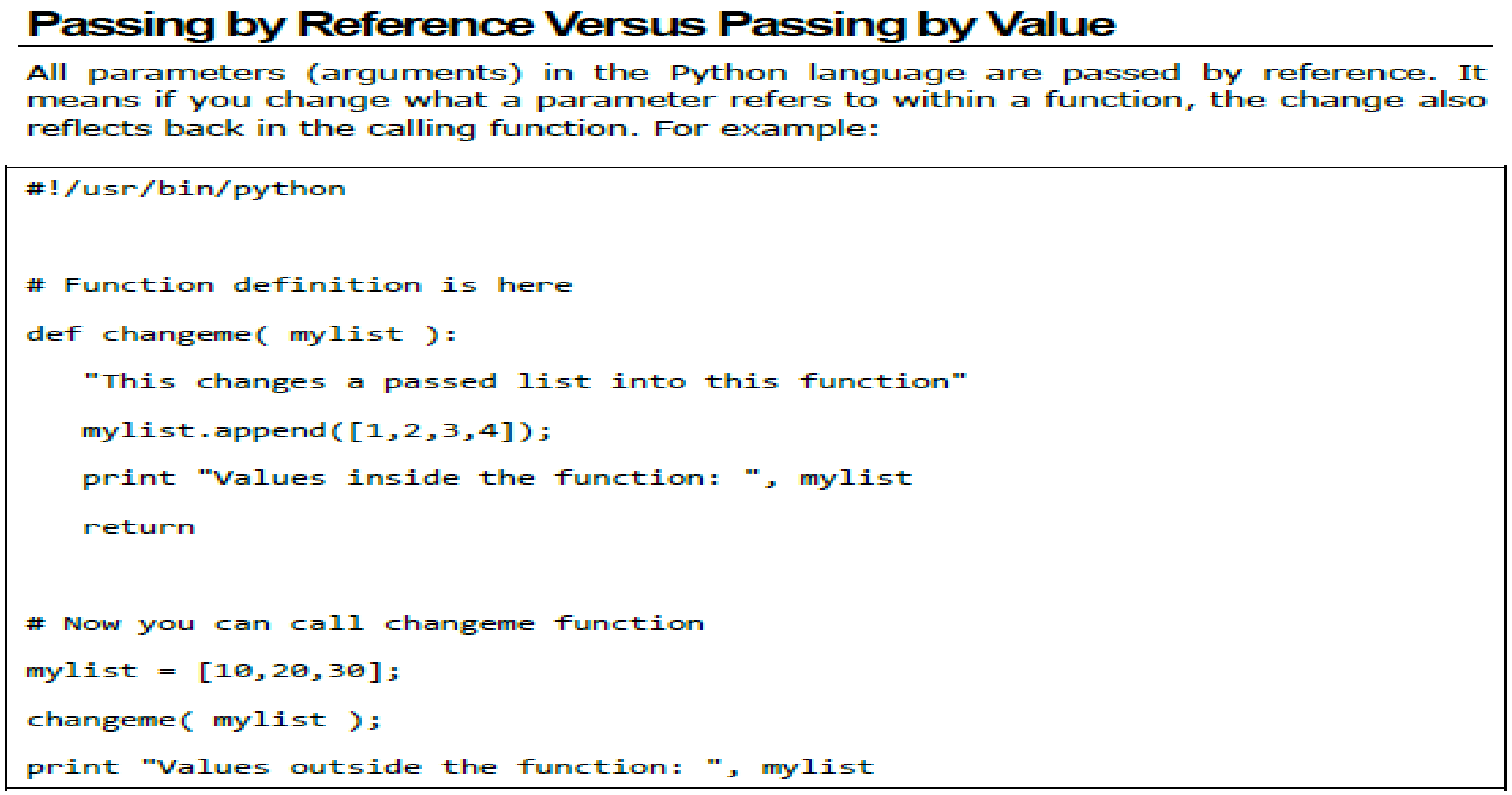
Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python:

- Global variables
- Local variables

FUNCTIONS

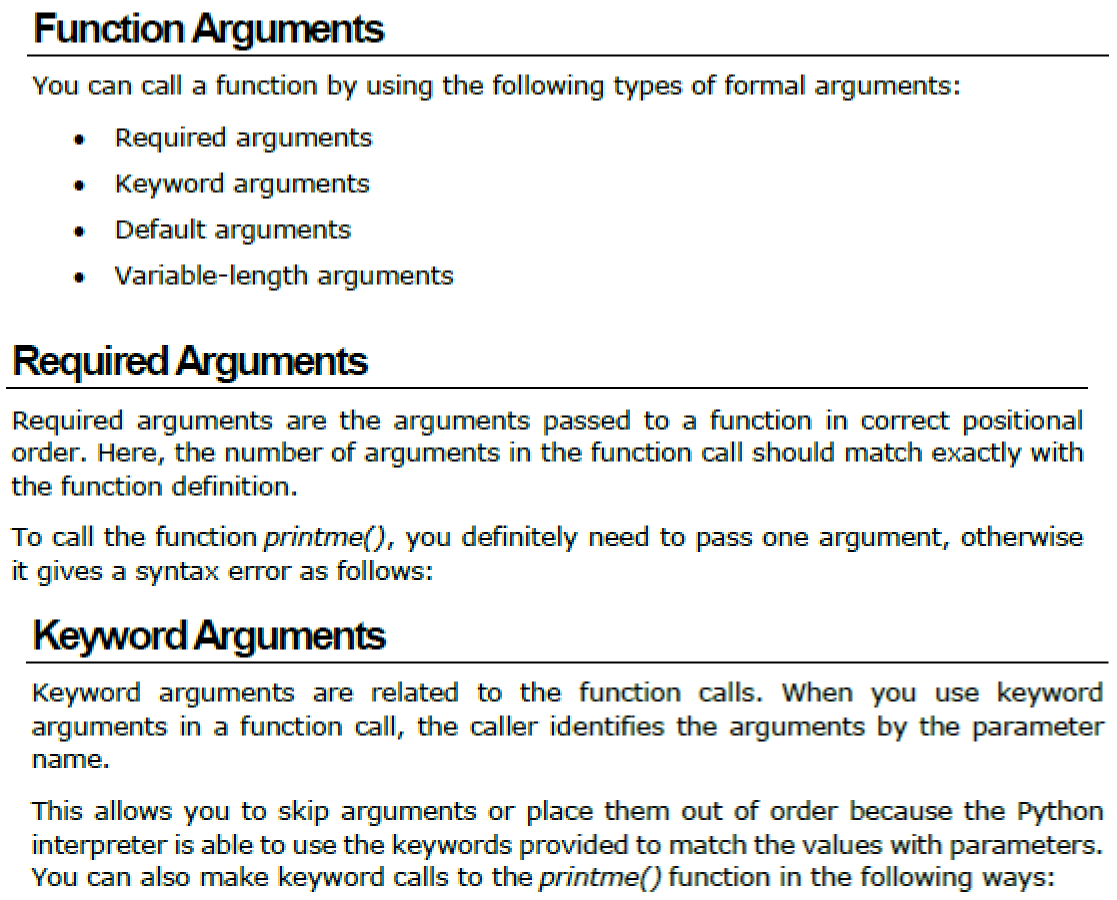


Passing by Reference Versus Passing by Value

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

```
#!/usr/bin/python


# Function definition is here

def changeme( mylist ):

   "This changes a passed list into this function"

   mylist.append([1,2,3,4]);

   print "Values inside the function: ", mylist

   return


# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist
```

FUNCTIONS



Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required Arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

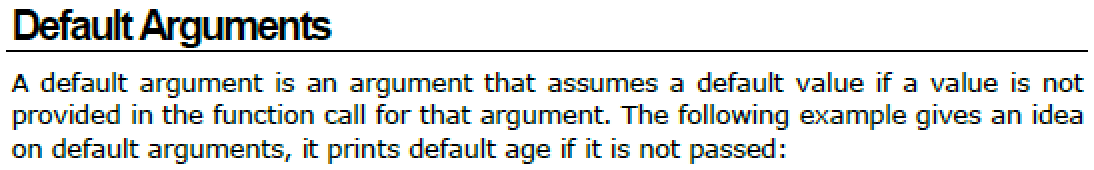
To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows:

Keyword Arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

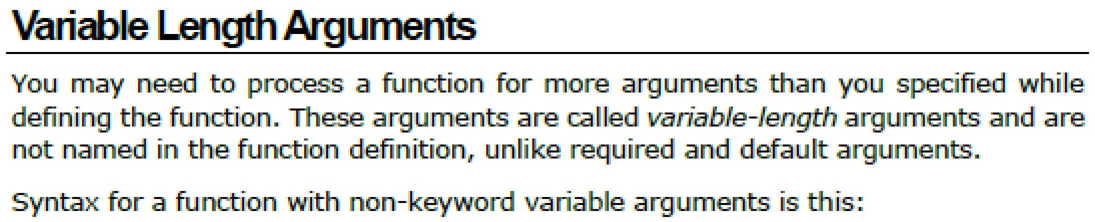
This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways:

FUNCTIONS



Default Arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed:

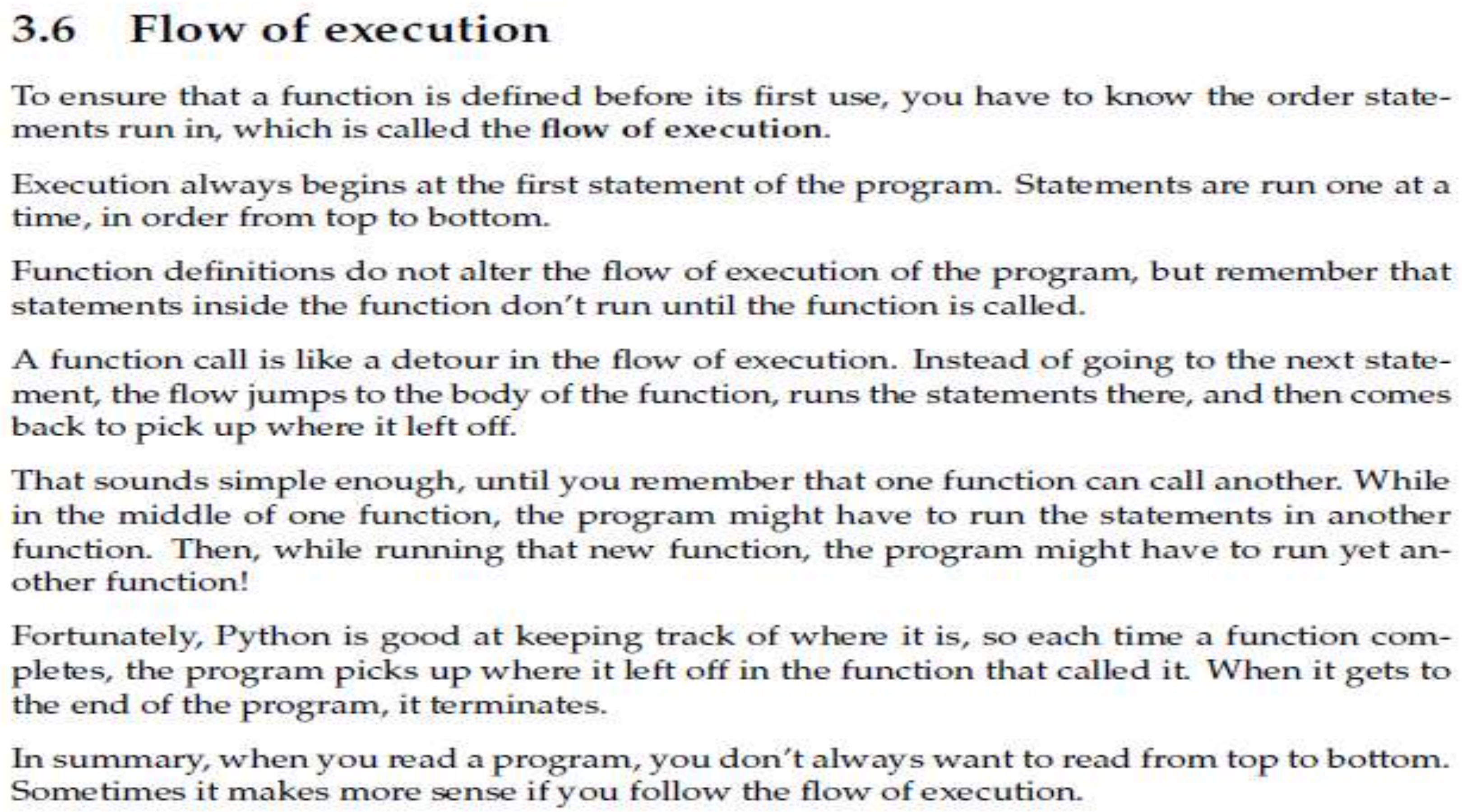


Variable Length Arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this:

FUNCTIONS



3.6 Flow of execution

To ensure that a function is defined before its first use, you have to know the order statements run in, which is called the **flow of execution**.

Execution always begins at the first statement of the program. Statements are run one at a time, in order from top to bottom.

Function definitions do not alter the flow of execution of the program, but remember that statements inside the function don't run until the function is called.

A function call is like a detour in the flow of execution. Instead of going to the next statement, the flow jumps to the body of the function, runs the statements there, and then comes back to pick up where it left off.

That sounds simple enough, until you remember that one function can call another. While in the middle of one function, the program might have to run the statements in another function. Then, while running that new function, the program might have to run yet another function!

Fortunately, Python is good at keeping track of where it is, so each time a function completes, the program picks up where it left off in the function that called it. When it gets to the end of the program, it terminates.

In summary, when you read a program, you don't always want to read from top to bottom. Sometimes it makes more sense if you follow the flow of execution.