Introduction to Programming W9

Functions II., I/O

2022/06/03

1 Return statements

Remember that writing an explicit `return` statement to a function terminates it and calling the function returns the specified value back to the caller code. The following function takes no arguments and simply returns a number:

```
def return_number():
    return 5

print("Calling the function:", return_number())
print("Calling the function and multiplying the returned value by 6:", return_number()*6)
x = return_number() #no printing, just store the returned value in a new variable x
y = x**2 #using the value of x to make a new variable y
print(y) #print y
```

Since there was no print statement inside the function, if one would just call the function without storing the returned value or printing it, there will be no output.

```
def return_number():
    return 5

return_number() #the number is returned, but not stored or printed (no output)
```

Remember that if you do not provide a return statement, the function will return `None`.

```
def multiply(x, y):
    z = x*y

our_value = multiply(3,4)
print(our_value) #This will print None, because the function did not return z

def multiply(x, y):
    z = x*y
    return z

our_value = multiply(3,4)
print(our_value) #This will print 12
```

Multiple values can be returned in one return statement:

```
def operation(x, y):
    z = x*y
    n = y**2
    return z, n
number_1, number_2 = operation(3,4)
print(number_1)
print(number_2)
```

If a Python function has more than one return statement, the first encountered will decide the return value and the function will terminate.

```
def is_even(n):
    if n % 2 == 0:
        return print("Even number!")
    else:
        return print("Odd number")

is_even(41)
is_even(42)
```

Functions can return Booleans as well:

```
def is_divisible(x, y):
     if x % y == 0:
         return True
     return False

boolean = is_divisible(4,2)
print(boolean)
```

2 Variable scope

It is crucial to remember the **scope** of the variables: which part of the program the variable can be used. Remember that if a variable is created inside a function, it cannot be used outside of the function.

```
def f():
    n = 5
    return n*2

print("Calling function f and printing the returned value:", f())
print(n) #this will result in an error "name 'n' is not defined"
```

This means the the same variable names can be used in different functions, since those variables are *local* to the functions. Below, the variable name `n` is used in both functions, but there is no conflict since `n` is local to the functions.

```
def function_f():
    n = 5
    print(n)
def function_g():
    n = "Just a string"
    print(n)

print("Calling the function function_f:")
function_f()
print("Calling the function function_g:")
function_g()
```

If a variable is outside of a function, it has a *global scope*. Although it can be accessed inside the function, the value of it cannot be changed inside the function.

```
x = 7 #variable with global scope

def function_f(y): #takes one argument as an input: y
    return(x*y) #the value of x can be still accessed!

print("Calling function function_f and printing the returned value:", function_f(3))

def function_f(y):
    x = 3 #x is not redefined here, it is a new variable
    return(y*3)
print("Calling function function_f and printing the returned value:", function_f(3))
#the global scope variable x is not changed:
print(x) #still 7!
```

If you try modifying variables inside a function with global scope, you will encounter an error when calling the function:

```
n = 0
def add(x):
    n += x #trying to increment the value of n with argument x

add(3)
```

The above code results in "UnboundLocalError: local variable 'n' referenced before assignment", because the function cannot modify variables outside the function's scope. The reason for this is that integers are immutable. Mutable objects such as lists and dictionaries with a global scope can be "modified" inside a function (e.g., appending an element to a list), because the object a variable name refers to is not changed. This concept is easier understood with using the built-in `id()` function that returns a unique id (memory address) for an object.

```
n = 5555
print(id(n)) #will print the unique id to n
n += 3 #adding 3 to n
print(id(n)) #the object the variable name refers to is changed! (different id)
```

This is because `n` was an integer that is an immutable data type. This is the reason it cannot be modified inside a function like we tried. On the other hand, if you add an element to a list, the id will be the same!

```
my_list = ["oranges", "grapes"]
print("Original list:", my_list)
print(id(my_list))
my_list.append("apples")
print("List is modified by adding a new element to it:", my_list)
print(id(my_list)) #note that the id is still the same!
```

This means, that modifying a mutable object with a global scope inside a list will be valid:

```
my_list = ["orange", "grapes"]
print("Original list:", my_list)

def modify_list(elem, remove_index):
    my_list.append(elem)
    my_list.pop(remove_index)

modify_list("apple", 1)
print("Modified:", my_list)
```

However, letting functions to change global scope variables is not always desirable. Passing a copy of the mutable object as an argument and returning a new object with the function is usually a better option (especially if we would like to keep the original object intact).

```
my_list = ["orange", "grapes"]
print("Original list:", my_list)

def make_new_list(lst, elem, remove_index):
    lst.append(elem)
    lst.pop(remove_index)
    return lst

new_list = make_new_list(my_list.copy(), "apple", 1)
print("New list:", new_list)
print("Original list is not modified:", my_list)
#IDs will be different:
print("Id of the original list:", id(my_list))
print("Id of the new list:", id(new_list))
```

3 I/O, file operations

The `open()` built-in function returns a file object (*handle*), and can be used to open a file. The basic syntax is the following:

```
#open the file w9.txt at the current working directory (same path as your Python file)
file = open("w9.txt",'w', encoding="utf-8")
#specify the full path:
file = open("/Users/name/Documents/w9.txt",'w', encoding="utf-8")
```

The first argument is a string with a filename. The second is a string, that specifies the mode while opening a file. 'r' is used for read only, 'w' for writing (if there is an existing file with the same name, it will be overwritten), and 'r+' opens the file for both reading and writing. Finally, 'a' is used for appending (data will be added to the end). The mode argument is optional, and 'r' is the default value if omitted. The third argument is the encoding (this will be UTF-8 in almost all cases).
When you are done performing operations on the file, it needs to be closed. The `.close()` method can be used for this:

```
#open the file w9.txt at the current working directory
file = open("w9.txt",'r+', encoding="utf-8")
#perform operations
file.close() #the file is closed
```

However, it is a good practice to use the `with` keyword to open a file, because in that case we do not have to call the `.close()` method explicitly, and it will be done automatically.

```
#using the with keyword to open a file in the current directory
with open('w9.txt', 'w', encoding = 'utf-8') as file:
    #perform operations here at the indented block
```

The name `file` after the `as` is the name of the variable with the file object, so any other name can be used (but people often use `file` or `f`). We will use this way to handle files in this class.
The following code uses the `.write()` method to create a new text file and write into it:

```
with open("w9.txt",'w',encoding = 'utf-8') as f:
   f.write("File starts\n")
   f.write("This is a line\n")
   f.write("Another line..") #no newline character here!
   f.write("Another string")
```

Open the file created (it will be in the same directory as your Python file) with a text editor the confirm the contents. After that read the file to a variable using the `.read()` method:

```
with open("w9.txt",'r',encoding = 'utf-8') as f:
   data = f.read()

print(data)
```

The `.readlines()` method returns a list from the file, where each line in the file is a list item:

```
with open("w9.txt",'r',encoding = 'utf-8') as f:
   data = f.readlines()

print(data)
print(type(data))
```

Note that the newline characters are included in the list element strings. If the newline characters are not needed, the `.splitlines()` method can be used on the string that is created with the `.read()` method.

```
with open("w9.txt",'r',encoding = 'utf-8') as f:
   data = f.read().splitlines()

print(data)
```

It is possible to write numbers to a text file, but the numbers need to be converted to strings. The `str()` function does this, and can be used in the following way to write a list of integers to a file (if the list elements are strings, you would not need to use this function).

```
my_numbers = [1,2,4,6,76,8,9,9]

with open("testw9.txt",'w',encoding = 'utf-8') as f:
   for number in my_numbers:
   #the elements of my_numbers are written line-by-line to testw9.txt as strings:
       f.write(str(number)+"\n")
```

If one would like to read the file in and convert it to the same format as `my_numbers` was (with integers), the following code could be used with the `int()` function:

```
with open("testw9.txt",'r',encoding = 'utf-8') as f:
    data = f.read().splitlines()

print(data)
print("Type of the list elements:", type(data[0])) #still strings!

new_data = []
for i in data:
    new_data.append(int(i))

print(new_data)
print("Type of the new list elements:", type(new_data[0])) #integers!
#the short version with list comprehension without creating a new variable name:
#data = [int(i) for i in data]
```

There are many alternative ways to read and write files, and there are some really short and convenient expressions using list comprehensions and string methods. You must use the `with` keyword to handle files, but it is up to you what specific method you choose (even if we did not cover it during the class), as long as you follow the exercise descriptions and your output is correct. Make sure you can:

1. Write a string to a file
2. Write a list to a file (with strings and integers), one element per line
3. Read in a file as a string
4. Read in a file to a list, one line - one list element (with and without newlines)