Section 3

Functions

Mohammed Asir Shahid

2021-07-30

Contents

1 Pythons Built-In Functions

Python comes with many built in functions such as print, input, and len which we have already used. Python also comes with Modules known as the Standard Library. For example, we have the Math module containing mathematics functions, the random module containing random number functions, etc. In order to use functions from these modules, we need to import them as follows:

```
import random

print(random.randint(1,10))
```

```
5
```

The above randint function gives us a random number between the given integers. We need to preface the randint function by "random" since it is inside of the random module. It is not a built-in function, so Python will not

find it without first calling the random module. Python's standard library has many such modules that can be imported via the import statement.

We can also import statements in a different way "from random import *" which removes the need to put in "random" before calling the function. However, this can decrease readibility as you do not know which module the function comes from.

```
from random import *

print(randint(1,10))
```

```
7
```

Sometimes you want to terminate a program early. There is a function for this in the "sys" module. The "sys.exit" function.

```
import sys

print("Hello")

sys.exit()

print("Goodbye")
```

```
Hello
```

As we can see above, the "Goodbye" string was not printed. This is due to the fact that we used the exit function and terminated the program early.

While Python comes with several modules in the standard library, we can also install new modules using the pip program.

```
pip install pyperclip
```

The pyperclip module contains 2 functions, the copy and paste functions which can be used to copy and paste text.

```
import pyperclip

pyperclip.copy("Hello world!")
print(pyperclip.paste())
```

```
Hello world!
```

2 Writing Your Own Functions

A function is like a mini program inside of a program containing code that runs when the function is called.

```
def hello():
    print("Howdy!")
    print("Howdy!!!")
    print("Hello there.")

hello()
hello()
hello()
```

```
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
Howdy!
Howdy!!!
Hello there.
```

When a function is defined using the "def" statement, the code inside of it is not executed. The code inside the function is only executed when the function is called.

One of the benefits of functions is that it lets you avoid duplicating code. Duplication can be an issue because when you find a bug in the code, you

need to make sure you fix it everywhere. With functions, you can just fix the function itself.

Our functions can also contain arguments that our function can use, for example the argument given in a "print" or "len" function.

```
def hello(name):
    print("Hello " + name)

hello("Alice")
hello("Bob")
```

```
Hello Alice
Hello Bob
```

Above when the "hello" function is called, the argument is passed into the function and used as the name variable.

```
def plusOne(number):
    return number+1


newNumber=plusOne(5)

print(newNumber)
```

```
6
```

What does the print function return?

The function returns a special value of a data type "None". It represents a lack of a data type.

```
spam=print()

print(type(spam))
```

```
<class 'NoneType'>
```

We can take away from this the fact that every function call has some sort of return value, including the print function. When a function does not have a return statement, the return value defaults to "None", as in the print function.

Some functions have keyword arguments. These can be optional arguments, for example the print function adds a new line after you call the function. However, this can be changed as seen below. We can also choose what separates the separating character between the arguments.

```
print("Hello")
print("World")


print("Hello", end="")
print("World")


print("cat", "dog", "mouse")


print("cat", "dog", "mouse", sep="ABC")


Hello
World
HelloWorld
cat dog mouse
catABCdogABCmouse
```

3 Global and Local Scopes

Variables inside of a function can have the same name as variables inside of a function. This is due to the fact that some variables and parameters have a local scope and others have a global scope. Variables that are assigned inside of a function exist inside of the function's local scope while variables that are assigned outside of functions exist in the program's global scope which

means they can be accessed from anywhere in the program. Any given line in the program is either in the global scope or inside of a local scope.

Scopes can be thought of containers for variables. All variables existing in the global scope are global variables while variables existing in a local scope are local variables.

A global scope for global variables is created when the program starts and is destroyed when the program ends while a local scope for local variables is created when the function is called and ends when the function returns.

There are a few reasons why local and global scopes matter.

1. Code in a global scope can't use local variable.

   Let's look at the following example:

```
def spam():
    eggs=99

spam()
print(eggs)
```

This looks like it should work, however it returns an error saying that eggs is not defined.

When we call the spam function, it creates the local scope. However, the eggs variables does not print. This is because after the spam function runs and returns, the local scope is destroyed. Thus we can't use the local variable in our global scope.

1. Code in a local scope can use global variables.

```
def spam():
    print(eggs)

eggs=42

spam()
```

```
42
```

As we can see above, first the spam function is assigned, then we define the eggs variable in the global scope, then the spam function is called. Now since Python does not see any local variables named eggs, it will check and see if there are any global variables named eggs, it will use that and print it out.

This eggs variable here is a global variable that is being read from a local scope. Python distinguishes based on where the variable is assigned.

```
def spam():
    eggs="Hello"
    print(eggs)

eggs=42

spam()
```

```
Hello
```

Above we can see that Python will prioritize local variables.

What if we want to change the global variable from inside of the local scope? We can do the following:

```
def spam():
    global eggs
    eggs="Hello"
    print(eggs)

eggs=42

spam()
print(eggs)
```

```
Hello
Hello
```

1. Code in one function's local scope can't use variables in another local scope.

```
def spam():
    eggs=99
    bacon()
    print(eggs)

def bacon():
    ham=101
    eggs=0

spam()
```

```
99
```

The above code prints out 99, which is the value of the eggs variable inside of the spam function.

When bacon is called in the spam function, we know that the eggs variable inside of the bacon function is different than the eggs variable inside of the spam function. When the bacon function runs, it creates the local scope, assigns values to the ham and eggs variables, and then returns which destroys the local scope. Then the spam function continues and prints out eggs with a value of 99.

1. You can use the same name for variables given that they are in a different scope.

   This is self explanatory.

So why do we need to have local and global scopes in the first place? Why not just have everything as a global variables?

The benefit that local variables provide is that they separate code from the rest of the program. This helps with debugging. If something is wrong in the global scope because of a bad variable, you only need to check the global scope for issues. If something is wrong in a function due to a bad variable, you only need to check the local scope of the function.