Day-18

Topic: Functions

#30daysofpython

Day-18 of #30daysofpython

Topic: Functions

- def keyword
- simple example of a function
- calling a function with ()
- accepting parameters
- print versus return
- adding in logic inside a function
- multiple returns inside a function
- adding in loops inside a function
- interactions between functions
- Lambda Functions
  - map()
  - filter()
  - *args & *kwargs

Functions

Introduction to Functions

Functions will be one of our main building blocks when we construct larger and larger amounts of code to solve problems.

What is a function?

Formally, a function is a useful device that groups together a set of statements so they can be run more than once. They can also let us specify parameters that can serve as inputs to the functions.

On a more fundamental level, functions allow us to not have to repeatedly write the same code again and again. If you remember back to the lessons on strings and lists, remember that we used a function len() to get the length of a string. Since checking the length of a sequence is a common task you would want to write a function that can do this repeatedly at command.

Functions will be one of most basic levels of reusing code in Python, and it will also allow us to start thinking of program design (we will dive much deeper into the ideas of design when we learn about Object Oriented Programming).

def keyword

In [ ]:

```
def name_of_function(arg1,arg2):
    '''
    This is where the function's Document String (docstring) goes.
    When you call help() on your function it will be printed out.
    '''
    # Do stuff here
    # Return desired result
```

Example

In [5]:

```
def say_helloz():
    print('hello')
```

Calling a function with ()

Call the function:

In [2]:

```
say_hello()
```

```
hello
```

If you forget the parenthesis (), it will simply display the fact that say_hello is a function. Later on we will learn we can actually pass in functions into other functions! But for now, simply remember to call functions with ().

In [7]:

```
say_helloz()
```

```
hello
```

Accepting parameters (arguments)

Let's write a function that greets people with their name.

In [8]:

```
def greeting(name):
    print(f'Hello {name}')
```

In [9]:

```
greeting('Jose')
```

```
Hello Jose
```

Using return

Example: Addition function

In [10]:

```
def add_num(num1,num2):
    return num1+num2
```

In [11]:

```
add_num(4,5)
```

Out[11]:

```
9
```

In [12]:

```
# Can also save as variable due to return
result = add_num(4,5)
```

In [13]:

```
print(result)
```

```
9
```

What happens if we input two strings?

In [14]:

```
add_num('one','two')
```

Out[14]:

```
'onetwo'
```

"What is the difference between *return* and *print*?"

The return keyword allows you to actually save the result of the output of a function as a variable. The print() function simply displays the output to you, but doesn't save it for future use. Let's explore this in more detail

In [15]:

```
def print_result(a,b):
    print(a+b)
```

In [16]:

```
def return_result(a,b):
    return a+b
```

In [17]:

```
print_result(10,5)
```

```
15
```

In [18]:

```
# You won't see any output if you run this in a .py script
return_result(10,5)
```

Out[18]:

```
15
```

But what happens if we actually want to save this result for later use?

In [19]:

```
my_result = print_result(20,20)
```

```
40
```

In [20]:

```
my_result
```

In [21]:

```
my_result1 = return_result(20,20)
```

In [22]:

```
my_result1
```

Out[22]:

```
40
```

In [23]:

```
type(my_result)
```

Out[23]:

```
NoneType
```

In [24]:

```
type(my_result1)
```

Out[24]:

```
int
```

print_result() doesn't let you actually save the result to a variable! It only prints it out, with print() returning None for the assignment!

In [25]:

```
my_result = return_result(20,20)
```

In [26]:

```
my_result
```

Out[26]:

```
40
```

In [27]:

```
my_result + my_result
```

Out[27]:

```
80
```

Adding Logic to Internal Function Operations

So far we know quite a bit about constructing logical statements with Python, such as if/else/elif statements, for and while loops, checking if an item is **in** a list or **not in** a list (Useful Operators Lecture). Let's now see how we can perform these operations within a function.

Check if a number is even

Recall the mod operator % which returns the remainder after division, if a number is even then mod 2 (% 2) should be == to zero.

In [ ]:

```
2 % 2
```

In [ ]:

```
20 % 2
```

In [ ]:

```
21 % 2
```

In [ ]:

```
20 % 2 == 0
```

In [ ]:

```
21 % 2 == 0
```

** Let's use this to construct a function. Notice how we simply return the boolean check.**

In [28]:

```
def even_check(number):
    return number % 2 == 0
```

In [29]:

```
even_check(20)
```

Out[29]:

```
True
```

In [30]:

```
even_check(21)
```

Out[30]:

```
False
```

Check if any number in a list is even

Let's return a boolean indicating if **any** number in a list is even. Notice here how **return** breaks out of the loop and exits the function

In [32]:

```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # Otherwise we don't do anything
        else:
            pass
```

** Is this enough? NO! We're not returning anything if they are all odds!**

In [33]:

```
check_even_list([1,2,3])
```

Out[33]:

```
True
```

In [34]:

```
check_even_list([1,1,1])
```

** VERY COMMON MISTAKE!! LET'S SEE A COMMON LOGIC ERROR, NOTE THIS IS WRONG!!!**

In [35]:

```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # This is WRONG! This returns False at the very first odd number!
        # It doesn't end up checking the other numbers in the list!
        else:
            return False
```

In [41]:

```
# UH OH! It is returning False after hitting the first 1
check_even_list([5,1,4])
```

Out[41]:

```
False
```

** Correct Approach: We need to initiate a return False AFTER running through the entire loop**

In [43]:

```
def check_even_list(num_list):
    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we return True
        if number % 2 == 0:
            return True
        # Don't do anything if its not even
        else:
            pass
    # Notice the indentation! This ensures we run through the entire for loop
    return False
```

In [44]:

```
check_even_list([1,2,3])
```

Out[44]:

```
True
```

In [45]:

```
check_even_list([1,3,5])
```

Out[45]:

```
False
```

Return all even numbers in a list

Let's add more complexity, we now will return all the even numbers in a list, otherwise return an empty list.

In [46]:

```
def check_even_list(num_list):

    even_numbers = []

    # Go through each number
    for number in num_list:
        # Once we get a "hit" on an even number, we append the even number
        if number % 2 == 0:
            even_numbers.append(number)
        # Don't do anything if its not even
        else:
            pass
    # Notice the indentation! This ensures we run through the entire for loop
    return even_numbers
```

In [47]:

```
check_even_list([1,2,3,4,5,6])
```

Out[47]:

```
[2, 4, 6]
```

In [48]:

```
check_even_list([1,3,5])
```

Out[48]:

```
[]
```

Returning Tuples for Unpacking

** Recall we can loop through a list of tuples and "unpack" the values within them**

In [49]:

```
stock_prices = [('AAPL',200),('GOOG',300),('MSFT',400)]
```

In [50]:

```
type(stock_prices)
```

Out[50]:

```
list
```

In [51]:

```
for item in stock_prices:
    print(item)
```

```
('AAPL', 200)
('GOOG', 300)
('MSFT', 400)
```

In [52]:

```
for stock,price in stock_prices:
    print(stock)
```

```
AAPL
GOOG
MSFT
```

In [53]:

```
for stock,price in stock_prices:
    print(price)
```

```
200
300
400
```

Similarly, functions often return tuples, to easily return multiple results for later use.

Let's imagine the following list:

In [54]:

```
work_hours = [('Abby',100),('Billy',400),('Cassie',800)]
```

The employee of the month function will return both the name and number of hours worked for the top performer (judged by number of hours worked).

In [55]:

```
def employee_check(work_hours):

    # Set some max value to intially beat, like zero hours
    current_max = 0
    # Set some empty value before the loop
    employee_of_month = ''

    for employee,hours in work_hours:
        if hours > current_max:
            current_max = hours
            employee_of_month = employee
        else:
            pass

    # Notice the indentation here
    return (employee_of_month,current_max)
```

In [56]:

```
employee_check(work_hours)
```

Out[56]:

```
('Cassie', 800)
```

Interactions between functions

Functions often use results from other functions, let's see a simple example through a guessing game. There will be 3 positions in the list, one of which is an 'O', a function will shuffle the list, another will take a player's guess, and finally another will check to see if it is correct. This is based on the classic carnival game of guessing which cup a red ball is under.

How to shuffle a list in Python

In [57]:

```
example = [1,2,3,4,5]
```

In [58]:

```
from random import shuffle
```

In [59]:

```
# Note shuffle is in-place
shuffle(example)
```

In [60]:

```
example
```

Out[60]:

```
[4, 3, 1, 5, 2]
```

OK, let's create our simple game

In [61]:

```
mylist = [' ','O',' ']
```

In [62]:

```
def shuffle_list(mylist):
    # Take in list, and returned shuffle versioned
    shuffle(mylist)

    return mylist
```

In [65]:

```
mylist
```

Out[65]:

```
[' ', 'O', ' ']
```

In [67]:

```
shuffle_list(mylist)
```

Out[67]:

```
['O', ' ', ' ']
```

In [68]:

```
def player_guess():

    guess = ''

    while guess not in ['0','1','2']:

        # Recall input() returns a string
        guess = input("Pick a number: 0, 1, or 2:  ")

    return int(guess)
```

In [69]:

```
player_guess()
```

```
Pick a number: 0, 1, or 2:  1
```

Out[69]:

```
1
```

Now we will check the user's guess. Notice we only print here, since we have no need to save a user's guess or the shuffled list.

In [70]:

```
def check_guess(mylist,guess):
    if mylist[guess] == 'O':
        print('Correct Guess!')
    else:
        print('Wrong! Better luck next time')
        print(mylist)
```

Now we create a little setup logic to run all the functions. Notice how they interact with each other!

In [71]:

```
# Initial List
mylist = [' ','O',' ']

# Shuffle It
mixedup_list = shuffle_list(mylist)

# Get User's Guess
guess = player_guess()

# Check User's Guess
#-----------------------
# Notice how this function takes in the input
# based on the output of other functions!
check_guess(mixedup_list,guess)
```

```
Pick a number: 0, 1, or 2:  2
Wrong! Better luck next time
['O', ' ', ' ']
```

Lambda Expressions, Map, and Filter

In [1]:

```
# Syntax: lambda arguments : expression

x = lambda a : a+10

print (x(5))
```

```
15
```

In [2]:

```
x = lambda a,b : a + b

print (x(5,6))
```

```
11
```

In [3]:

```
x = lambda a,b,c : a + b + c

print (x(10,11,12))
```

```
33
```

In [4]:

```
def prod_n(num):
    return lambda a: a*num

output_triple=prod_n(3)

output_double=prod_n(2)

print(output_triple(2))
```

```
6
```

map function

In [9]:

```

lst=[2,4,6,8,10]

final_lst = list(map(lambda x: x**2, lst))

print(final_lst)
```

```
[4, 16, 36, 64, 100]
```

In [10]:

```
lst_animals=['dog','cat','rabbit']

upper_list = list(map(lambda a: str.upper(a),lst_animals))

print (upper_list)
```

```
['DOG', 'CAT', 'RABBIT']
```

filter function

In [11]:

```
def check_even(num):
    return num % 2 == 0
```

In [12]:

```
nums = [0,1,2,3,4,5,6,7,8,9,10]
```

In [13]:

```
filter(check_even,nums)
```

Out[13]:

```
<filter at 0x226e2bd6a60>
```

In [14]:

```
list(filter(check_even,nums))
```

Out[14]:

```
[0, 2, 4, 6, 8, 10]
```

In [15]:

```
final_evenlist= list(filter (lambda x : x%2==0, nums))
print(final_evenlist)
```

```
[0, 2, 4, 6, 8, 10]
```

In [16]:

```
from functools import reduce
list_n=[10,20,30,40,50,60]

sum_n = reduce((lambda x,y: x+y), list_n)
print (sum_n)
```

```
210
```

lambda expression

In [17]:

```
lambda num: num ** 2
```

Out[17]:

```
<function __main__.<lambda>(num)>
```

In [18]:

```
list(map(lambda num: num ** 2, my_nums))
```

Out[18]:

```
[1, 4, 9, 16, 25]
```

In [19]:

```
list(filter(lambda x: x%2==0,lst))
```

Out[19]:

```
[2, 4, 6, 8, 10]
```

*args and **kwargs

*args

In [20]:

```
def myfunc(a=0,b=0,c=0,d=0,e=0):
    return sum((a,b,c,d,e))*.05

myfunc(40,60,20)
```

Out[20]:

```
6.0
```

In [21]:

```
def myfunc(*args):
    return sum(args)*.05

myfunc(40,60,20)
```

Out[21]:

```
6.0
```

In [22]:

```
def myfunc(*b):
    return sum(b)*.05

myfunc(40,60,20)
```

Out[22]:

```
6.0
```

In [26]:

```
def my_func1(*args):

    total=0

    for i in args:
        total=total + i

    print(total)


#Positional argument using *args
my_func1(2,3,4,5)

my_func1(2,3)

my_func1(1,5,2,3)
```

```
14
5
11
```

**kwargs

In [24]:

```
def myfunc(**kwargs):
    if 'fruit' in kwargs:
        print(f"My favorite fruit is {kwargs['fruit']}")  # review String Formatting and f-
    else:
        print("I don't like fruit")

myfunc(fruit='pineapple')
```

```
My favorite fruit is pineapple
```

In [25]:

```
myfunc(fruits='')
```

```
I don't like fruit
```