Pierian-Data / Complete-Python-3-Bootcamp Public

Code Issues 95 Pull requests 74 Actions Projects Security Insights

master

Complete-Python-3-Bootcamp / 03-Methods and Functions / .ipynb_checkpoints / 02-Functions-checkpoint.ipynb

Go to file ...

Pierian-Data added image link to all notebooks

Latest commit c6d506f on Jun 25, 2020 History

1 contributor

1381 lines (1381 sloc) 29.2 KB

Raw Blame

Functions

Introduction to Functions

This lecture will consist of explaining what a function is in Python and how to create one. 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).

Why even use functions?

Put simply, you should use functions when you plan on using a block of code multiple times. The function will allow you to call the same block of code without having to write it multiple times. This in turn will allow you to create more complex Python scripts. To really understand this though, we should actually write our own functions!

Function Topics

- 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
- tuple unpacking
- interactions between functions

def keyword

Let's see how to build out a function's syntax in Python. It has the following form:

In [1]:
```
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
```

We begin with `def` then a space followed by the name of the function. Try to keep names relevant, for example len() is a good name for a length() function. Also be careful with names, you wouldn't want to call a function the same name as a built-in function in Python (such as len).

Next come a pair of parentheses with a number of arguments separated by a comma. These arguments are the inputs for your function. You'll be able to use these inputs in your function and reference them. After this you put a colon.

Now here is the important step, you must indent to begin the code inside your function correctly. Python makes use of *whitespace* to organize code. Lots of other programing languages do not do this, so keep that in mind.

Next you'll see the docstring, this is where you write a basic description of the function. Using Jupyter and Jupyter Notebooks, you'll

Next you'll see the docstring, this is where you write a basic description of the function. Using Jupyter and Jupyter Notebooks, you'll be able to read these docstrings by pressing Shift+Tab after a function name. Docstrings are not necessary for simple functions, but it's good practice to put them in so you or other people can easily understand the code you write.

After all this you begin writing the code you wish to execute.

The best way to learn functions is by going through examples. So let's try to go through examples that relate back to the various objects and data structures we learned about before.

Simple example of a function

In [4]:

```python
def say_hello():
    print('hello')
```

Calling a function with ()

Call the function:

In [5]:

```python
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]:

```python
say_hello
```

Out[7]:

```
<function __main__.say_hello>
```

Accepting parameters (arguments)

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

In [1]:

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

In [2]:
```
greeting('Jose')
```

```
Hello Jose
```

Using return

So far we've only seen print() used, but if we actually want to save the resulting variable we need to use the **return** keyword.

Let's see some example that use a `return` statement. `return` allows a function to *return* a result that can then be stored as a variable, or used in whatever manner a user wants.

Example: Addition function

In [6]:
```
def add_num(num1,num2):
    return num1+num2
```

In [7]:
```
add_num(4,5)
```

Out[7]:
```
9
```

In [8]:
```
# Can also save as variable due to return
result = add_num(4,5)
```

In [9]:
```
print(result)
```

```
9
```

What happens if we input two strings?

In [10]:
```
add_num('one','two')
```

```
Out[10]: 'onetwo'
```

Very Common Question: "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 [1]:
def print_result(a,b):
    print(a+b)
```

```
In [2]:
def return_result(a,b):
    return a+b
```

```
In [3]:
print_result(10,5)
```

```
15
```

```
In [4]:
# You won't see any output if you run this in a .py script
return_result(10,5)
```

```
Out[4]: 15
```

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

```
In [5]:
my_result = print_result(20,20)
```

```
40
```

```
In [6]:
my_result
```

```
In [7]:
type(my_result)
```

```
Out[7]: NoneType
```

Be careful! Notice how 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 [8]: my_result = return_result(20,20)
```

```
In [9]: my_result
```

```
Out[9]: 40
```

```
In [10]: my_result + my_result
```

```
Out[10]: 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 [11]: 2 % 2
```

```
Out[11]: 0
```

In [12]:
```
20 % 2
```
Out[12]:
```
0
```
In [14]:
```
21 % 2
```
Out[14]:
```
1
```
In [15]:
```
20 % 2 == 0
```
Out[15]:
```
True
```
In [16]:
```
21 % 2 == 0
```
Out[16]:
```
False
```
** Let's use this to construct a function. Notice how we simply return the boolean check.**

In [18]:
```
def even_check(number):
    return number % 2 == 0
```
In [19]:
```
even_check(20)
```
Out[19]:
```
True
```
In [21]:
```
even_check(21)
```
Out[21]:
```
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 [25]:

```
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 [26]:

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

Out[26]:

```
True
```

In [27]:

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

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

In [28]:

```
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 [30]:

```
# UH OH! It is returning False after hitting the first 1
check_even_list([1,2,3])
```

Out[30]:

```
False
```

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

In [31]:
```
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 [32]:
```
check_even_list([1,2,3])
```

Out[32]:
```
True
```

In [34]:
```
check_even_list([1,3,5])
```

Out[34]:
```
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 [35]:
```
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 [36]:

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

Out[36]:

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

In [37]:

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

Out[37]:

```
[]
```

Returning Tuples for Unpacking

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

In [38]:

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

In [39]:

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

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

In [41]:

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

```
AAPL
GOOG
MSFT
```

In [42]:

```
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 [46]:
```
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 [47]:
```
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 [48]:
```
employee_check(work_hours)
```

Out[48]:
```
('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 [8]:
```
example = [1,2,3,4,5]
```

In [9]:
```
from random import shuffle
```

In [10]:
```
# Note shuffle is in-place
shuffle(example)
```

In [11]:
```
example
```

Out[11]:
```
[3, 1, 4, 5, 2]
```

OK, let's create our simple game

In [12]:
```
mylist = [' ','O',' ']
```

In [13]:
```
def shuffle_list(mylist):
    # Take in list, and returned shuffle versioned
    shuffle(mylist)

    return mylist
```

In [14]:
```
mylist
```

Out[14]:
```
[' ', 'O', ' ']
```

In [15]:
```
shuffle_list(mylist)
```

```
Out[15]: [' ', ' ', 'O']
```

```
In [18]:
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 [24]:
player_guess()
```

```
Pick a number: 0, 1, or 2:  1
Out[24]: 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 [22]:
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 [23]:
# Initial List
mylist = [' ','O',' ']

# Shuffle It
mixedup_list = shuffle_list(mylist)

# Get User's Guess
guess = player_guess()
```

```
# Check User's Guess
#-------------------------
```