Day4-Functions

July 22, 2021

Day 4: Welcome to Functions

Functions are incredible tools that you can write when Python or a package doesn't have what you want to do.

Functions are named blocks of code that are designed to do a specific job. Examples could be doing an equation or plotting data. When you want to perform the task, you call the function instead of writing out all the code inside it.

First you define the function by using `def`, then you name the function some explanatory name to call it later. Then you tell the function what arguments it will take in. Next you tell the function what to do with those arguments.

Goals for the day:

- Define functions
- Create a function that prints outputs
- Create functions that return statements
- Learn how to use a list as an argument
- Learn how to use dictionaries as arguments
- Learn how to pass an arbitrary number of arguments to the function

Functions Learned:

- Define function:

```
def function(input):    print(output)
```

- Set dictionary: `dictionary_name= {key1:value1, key2:value2}`
- Function that accepts any # of positional arguments: `Def function(*args):`
- Function passes arbitrary #of keyword arguments: `Def function(**kwargs):`

1.0 Function that takes in a variable and prints response

We will start with a simple function and then work our way up. The function we will create will take in some **argument** `name` and then return a statement saying hello with the name! This would be helpful if you had many people to say hello to.

We first use `def`, then name our function, and put our argument in parenthesis. Then on the end of our first line, we have a colon. After the colon, we tell Python everything we want to include by starting the line with a tab.

[1]:
```
# define our function
def greet_user(name):
    '''Welcomes the user with input name''' # this is called a docstring and␣
 ↪describes what the function does
    print(f"Welcome {name}!") # this is the f string that we learned yesterday


# then run our new function
greet_user("Katie")
```

```
Welcome Katie!
```

We can now use our function to welcome different people, by inputting different names as the arguments:

[2]:
```
greet_user("Alex") # run the function to greet Alex
greet_user("Zach Efron") # run the function to greet Zach Efron

greet_user(1)
```

```
Welcome Alex!
Welcome Zach Efron!
Welcome 1!
```

On the second line there is a `docstring` which allows you to write the aim for the function. You can call up this documentation using `__doc__` (which is two underscores on each side)

[3]:
```
greet_user.__doc__
```

[3]:
```
'Welcomes the user with input name'
```

You can also call this up by using the built in function `help` to call information up. This also works on other built in functions and functions from packages.

[4]:
```
help(greet_user)
```

```
Help on function greet_user in module __main__:

greet_user(name)
    Welcomes the user with input name
```

We can also use `help` to get information on built in functions, like `len` from yesterday:

[5]:
```
help(len) # get information on the built in function len
```

```
Help on built-in function len in module builtins:
```

```
len(obj, /)
    Return the number of items in a container.
```

2.0 The ordering of arguments in the function is important

We will write an example that uses in multiple arguments, the year, month, and day into a function we name `print_date` and it will return the date for us in scientific notation.

When we call the function, we make sure to insert the year, month, and day in the correct order that we made when we set up the arguments in our function:

[6]:
```
def print_date(year, month, day): # the order is year, month, day
  """Takes in year, month, and day and prints date"""
  joined = str(year) + '-' + str(month) + '-' + str(day) # make each day a␣
 ↪string and then add together
  print(joined)

# lets print out Nov 12th in 2020
print_date(2020, 11, 12) # the order is year, month day
```

```
2020-11-12
```

We can test out what happens if we put them in a different order. For example, month, day, year. The output will not be in the order we intended, which could mess up our data analysis.

[7]:
```
# the November 12, 2020 the American way

print_date(11, 12, 2020) # the order in the function is year, month, day
```

```
11-12-2020
```

We can fix this by being explicit in calling our function, and typing `year = 2020` for example. This way, we are telling our function exactly which value to use for each argument, and it makes our code more readable in the future. This is the best practice for calling functions and is set up so that `keyword = value`:

[8]:
```
print_date(year = 2020, month = 11, day = 12)
```

```
2020-11-12
```

This is also nice, because now the order doesn't matter, because we are explicitly linking the values to the variables, so even if we switch up the order of the variables, the function `print_date` knows what input you're trying to give it.

[9]:
```
print_date(month = 11, day = 12, year = 2020)
```

```
2020-11-12
```

3.0 Saving the output of the function to a variable

Say we want to save this date as something, we might try to call our function `print_date` and save it to a variable `my_date`:

[10]:
```
my_date = print_date(year = 2011, month = 5, day = 10)
print(my_date)
```

```
2011-5-10
None
```

[11]:
```
my_date
```

`my_date` now has the value `None`, which is not what we intended. If we look at what our function is designed to do, we see that it is set to just print out the date. If we want to be able to assign the value to something, we must instead change it to `return` a value.

We're going to change the name of the function to be `return_date` to distinguish it from `print_date`. You can see that it's not that different from our previous function

[12]:
```
def return_date(year, month, day): # the order is year, month, day
  """Takes in year, month, and day and returns date"""
  joined = str(year) + '-' + str(month) + '-' + str(day)
  return(joined)

my_date = return_date(year = 2011, month = 5, day = 10)
print(my_date)
```

```
2011-5-10
```

4.0 Coding check in

Write a function that takes in radius as an argument, `r`, and returns the area of a circle given the radius ($A = \pi r^2$). Python doesn't have a built in value for `pi`, so use `3.1415`.

Advanced: add a second argument to the function that takes in either `"square"` or `"circle"`, which will calculate the area of a circle or a square depending on that argument (using an `if` statement). This function should also tell the user that it doesn't know the shape if it isn't a circle or square.

[13]:
```
### space to put your code here
```

4.1 Check in answer

[14]:
```
def get_area(r):
  """give radius, get area"""
  area = 3.1415* r**2
  return(area)

get_area(5)
```

[14]:
```
78.53750000000001
```

4.2 Check in answer (advanced)

[15]:
```
def get_area(r, type):
  """give radius of circle or square, get area"""

  if type == "circle":
    area = 3.1415* r**2
    return(area)

  elif type == "square":
    area = r*r
    return(area)

  else:
    print(f"{type} is not a shape I know")

get_area(5, type = "square")
```

[15]: 25

5.0 Using a list as an arguement

Here, we will show that a function can take a list as an argument using a `for` loop. We will give a function `zoo_animals` a list of animals, and then the function will tell us the zoo has that animal, by looping through the values in the list.

[16]:
```
def zoo_animals(animal):
  """Takes in a string of animals and prints a string for each animal"""
  for x in animal:
    print(f'The zoo has a {x}')

animals = ["tiger", "lion", "bear"]

zoo_animals(animals)
```

```
The zoo has a tiger
The zoo has a lion
The zoo has a bear
```

6.0 Using dictionaries as arguments

This is a nice way of keeping your code very readable and can allow you to switch between many parameter sets. It would also allow you to input the same parameter sets into multiple functions, without allowing for the possibility of mistyping your arguments. First, we define our function, which takes in name, school, city, and state and returns a statement using an `f string`:

[17]:
```
def student_information(name, school, city, state):
    print(f'{name} goes to the {school} in {city}, {state}')
```

Now, we will create two dictionaries for two students, and then pass them into our student information dictionary.

Note: this is another way to write a dictionary than we did yesterday, by using curly brackets and the convention `{key1:value1, key2:value2}`

[18]:
```
# dictionary for student one
student_1 = {"name": "Katie", "school": "University of Chicago", "city":␣
 ↪"Chicago", "state": "IL"}
# dictionary for student two
student_2 = {"name": "Michelle", "school": "Princeton University", "city":␣
 ↪"Princeton", "state": "NJ"}

# use the function to print out student information for 2 students
# need to use ** to tell Python we are inputting a dictionary
student_information(**student_1)
student_information(**student_2)
```

```
Katie goes to the University of Chicago in Chicago, IL
Michelle goes to the Princeton University in Princeton, NJ
```

This is the same thing as writing it inside the function call, but it would be easier if you need this parameter set for multiple functions

[19]:
```
student_information(name = "Katie", school = "University of Chicago", city =␣
 ↪"Chicago", state = "IL")
```

```
Katie goes to the University of Chicago in Chicago, IL
```

7.0 Coding check in

Write a function called `get_hypotenuse` that takes in the two sides of a right triangle as arguments from a dictionary, and the returns the hypotenuse using the pythagorean theorem. Remember, that equation is:

$$c = \sqrt{a^2 + b^2}$$

Also remember that the square root is the same as raising somethign to the $\frac{1}{2}$.

Test out some different values for `a` and `b`

[20]:
```
## coding check in here
```

7.1 Coding check in answer

[21]:
```
# define our function
def get_hypotenuse (a,b):
  hypot = (a**2 + b**2)**(0.5)
  return(hypot)
```

```
# define our dictionaries
triangle_1 = {"a": 1, "b":2}
triangle_2 = {"a": 15, "b":1}

# use function to get the hypotenuse
print(get_hypotenuse(**triangle_1))
print(get_hypotenuse(**triangle_2))
```

```
2.23606797749979
15.033296378372908
```

8.0 Using `*args` and `**kwargs` in functions

There are certain tricks that you can use when making functions. The first is by using `*args` when you are unsure of how many arguements you are going to pass to the function, say if you're trying to add numbers together, but in diffrent conditions, you might be adding a varying number.

We can do so by using the convention `*args` where before we put our arguments. Inside of that, we first set up a variable, `x` to be 0, and then we are going to add numbers to `x` using a for loop, that will go over each element in args. This `*args` now accepts any number of positional arguements:

[22]:
```
def add_stuff(*args):
  x = 0
  for num in args:
    x = x + num
    print(x) # printing out x each time it updates, to peak inside the function

add_stuff(1,2,3,4,5)
```

```
1
3
6
10
15
```

So, we saw what happened when we put in 5 arguments, now what happens when we put in two:

[23]:
```
add_stuff(100,200)
```

```
100
300
```

The second trick I want to go over is using `**kwargs`. This is basically just telling the function that you are going to pass an arbitrary number of keyword arguments that you haven't define beforehand.

[24]:
```
def intro(**kwargs):

    for key, value in kwargs.items():
        print(f"{key} is {value}")
```

```
intro(Firstname="Katie", Lastname="Dixon", Age=27, Major = "E&E")
```

```
Firstname is Katie
Lastname is Dixon
Age is 27
Major is E&E
```

[25]:
```
intro(Name="Elle Woods", Pet="Bruiser Woods", School="Harvard")
```

```
Name is Elle Woods
Pet is Bruiser Woods
School is Harvard
```

9.0 Extra check in

Write a function that takes in however many arguments (*args) and returns a list of all the squared values. Remember, you have to first define an empty list outside of the for loop to start adding values to it

[26]:
```
#### check in code here
```

9.1 Extra check in answer

[27]:
```
def get_square(*args):

  square = [] # empty list

  for val in args:
    square.append(val**2) # appending squared values for each value in args

  return(square) # return the list of squared values


print(get_square(1,2,3,5,10))
print(get_square(5,10,25))
```

```
[1, 4, 9, 25, 100]
[25, 100, 625]
```

10.0 Extra: arguments and *args

You can also include required keyword arguments and positional arguments, *args, in the same function. As an example, we can write a function where we input the polynomial we want, and then we can input any number of arguments to this function and it will return a list of those values raised to that value:

[28]:
```
def get_poly(polynomial,*args):
```

```
  print(polynomial) # printing the value of the polynomial
  poly_vals = [] # empty list

  for val in args:
    poly_vals.append(val**polynomial) # raising each value in args to that power

  return(poly_vals) # returning the list


get_poly(2,1,2,3,5,10) # the first position will be the polynomial
```

```
2
```

[28]:
```
[1, 4, 9, 25, 100]
```

[29]:
```
get_poly(3, 2, 5, 10)
```

```
3
```

[29]:
```
[8, 125, 1000]
```

Appendix 1.0 Using an `f string` to play a fun game

We can use a dictionary, an `if` statement, and a `f string` to play a game where we guess a number and maybe get a lucky fruit.

We define our function to take in `some_number`, and then we use a dictionary to link numbers to the fruits, the number is the `key` in the dictionary. We then use an `if` statement to see if `some_number` is in the numbers we have in the dictionary. If it is, it will return a fruit using an `f string`. If it is not, it will print that you don't get a fruit.

[30]:
```
# defining our function
def get_my_lucky_fruit(some_number):
  all_fruits=['apple','cherry','banana','strawberry','tomato']
  num_fruit=[1,3,6,7,8]
  num_fruit_dict=dict(zip(num_fruit,all_fruits))
  if some_number in num_fruit:
    my_fruit = num_fruit_dict.get(some_number)
    print (f'You entered {some_number}, your lucky fruit is {my_fruit}')
  else:
    print (f'You entered {some_number}, there is no lucky fruit')

# calling our function
get_my_lucky_fruit(some_number = 4)
```

```
You entered 4, there is no lucky fruit
```