Functions

Objectives:

- Describe what a function is and how they are useful
- Explain exactly what the return keyword does and some of the side effects when using it
- Add parameters to functions to output different data
- Define and diagram how scope works in a function
- Add keyword arguments to functions

What is a function?

- A process for executing a task
- It can accept input and return an output
- Useful for executing similar procedures over and over

Why use functions?

- Stay **DRY** — **D**on't **R**epeat **Y**ourself
- Clean up and prevent code duplication
- "Abstract away" code for other users
    - Imagine if you had to rewrite the "print()" function for every program you wrote

Defining Functions:

The structure of a function looks like this:

```
def name_of_function():
```

Typically for functions, it is all lowercase. After creating a function, it will run when it is invoked or called. i.e.

```
def say_hi():
    print('hi!')

say_hi()        #prints 'hi!'
```

Your First Function

Define a function named `make_noise` that prints "THE CROWD GOES WILD" (make sure you spell it exactly the same). Execute it once at the bottom of your file.

exercise.py

```
#Define your make_noise function below
def make_noise():
    print('THE CROWD GOES WILD')

#Then, call make_noise once:
make_noise()
```

Line 6, Column 13 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

Returning Values from Functions:

- Exits the function
- Outputs whatever value is placed after the return keyword
- Pops the function off of the call stack

Super Quick Return Exercise

Write a function called speak_pig that returns 'oink'. Yup, that's it.

exercise.py

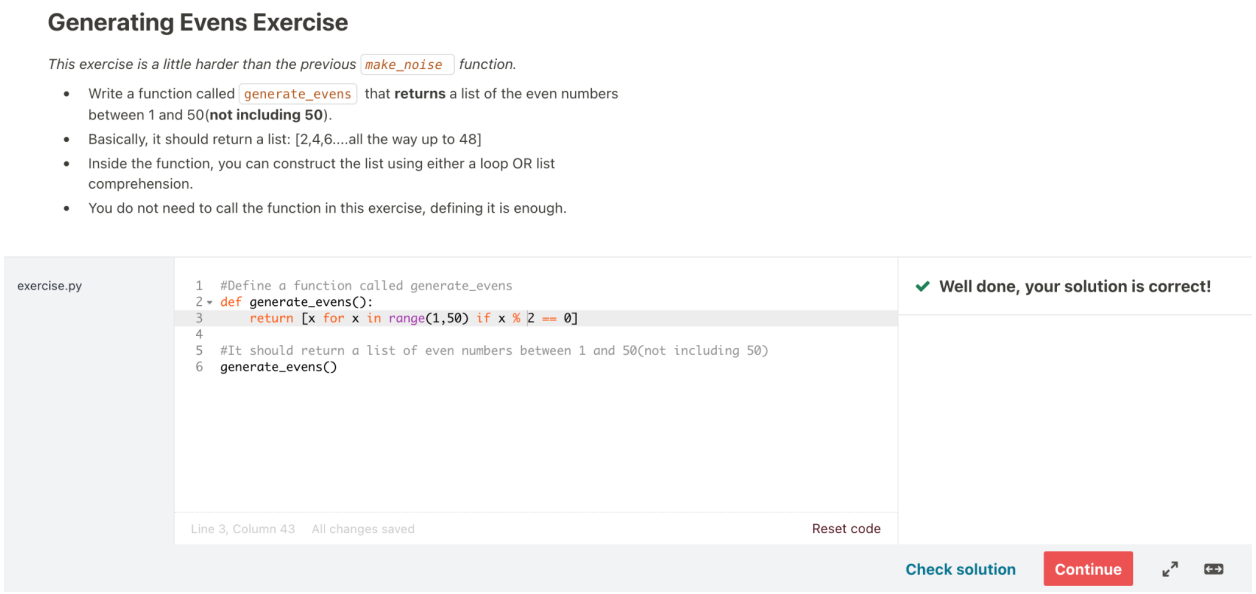
```
def speak_pig():
    return 'oink'

speak_pig()
```

Line 4, Column 12 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue



Parameters:

- Variables that are passed to a function—think of them as placeholders that get assigned when you call the function

Naming Parameters:

- When it comes to naming parameters, it's important to have parameters with names that actually make sense and are semantic. By doing this, it helps the code much easier to understand

Parameters vs Arguments:

- A **parameter** is a variable in a method definition
- When a method is called, the **arguments** are the data you pass into the method's **parameters**
- **Parameter** is a variable in the declaration of the function
- **Argument** is the actual value of this variable that gets passed to the function

Yell Function Exercise

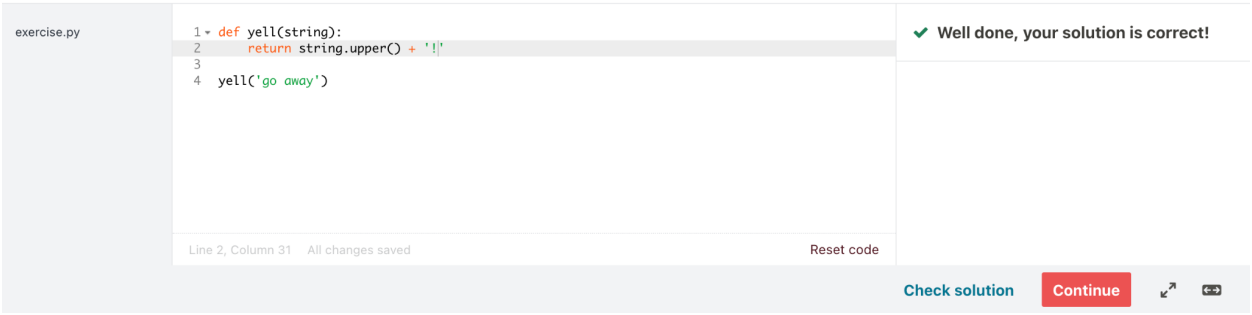
Implement a function `yell` which accepts a single string argument. It should **return(not print)** an uppercased version of the string with an exclamation point aded at the end. For example:

`yell("go away")` # "GO AWAY!"

`yell("leave me alone")` # "LEAVE ME ALONE!"

You do not need to call the function to pass the tests.

Remember, that currently you can't use f-strings in Udemy coding challenges, so either use string concatenation or the format() method.



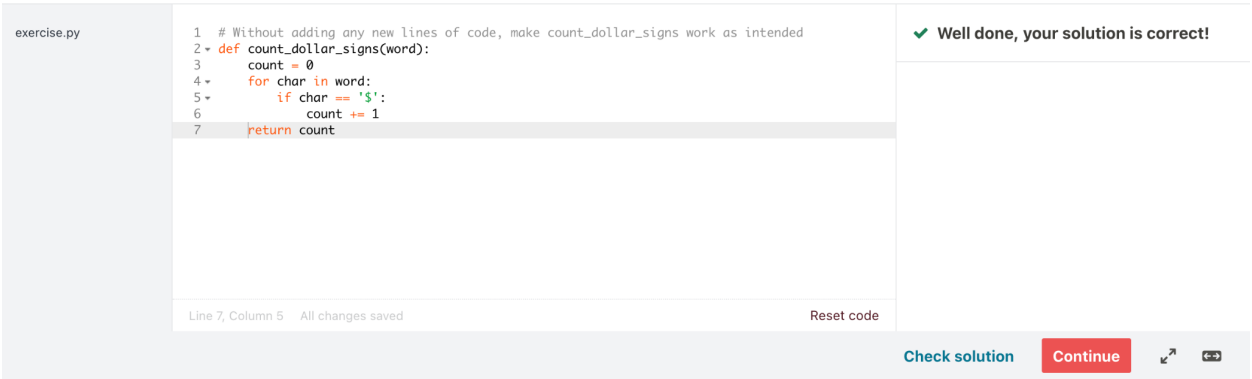
Common Return Mistakes:

- Returning the result too soon. This can happen in a conditional or a loop, or sometimes all over the place.
    - To prevent this from happening, it's important to make sure that the indentation is correct
- Unnecessary 'else'—it's important to make the code look cleaner by using the return function properly

Fix This Function!

The pre-written `count_dollar_signs` function is broken. It's supposed to return the number of $ characters in a given string. For example: `count_dollar_signs("$uper $ize")` should return `2`. But for some reason, the function always returns either 0 or 1. What's going on?

Without adding any new lines (just move existing code around), make it work as intended.



Default Parameters:

- If there is no argument specified for the parameter, we can create a default like this: i.e.

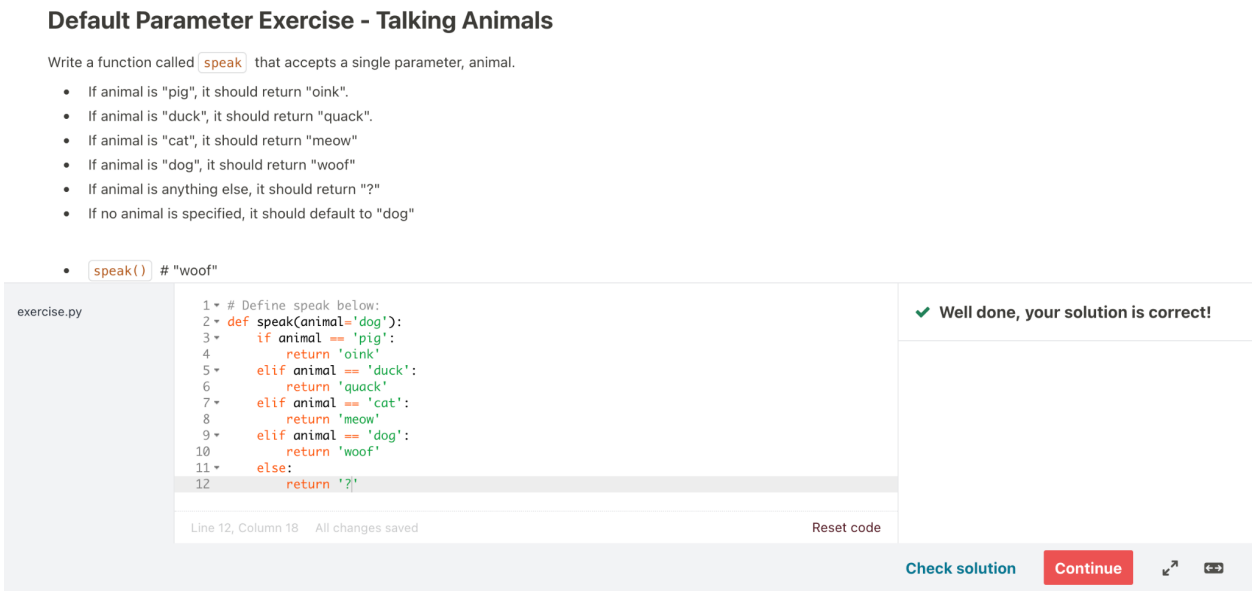
```
def exponent(num, power=2):        # in case if no power is specified, 2 is default
        return num ** power
```

Why have default parameters?

- Allows you to be more defensive
- Avoids errors with incorrect parameters
- More readable examples

What can default parameters be?

- Anything! Functions, lists, dictionaries, strings, booleans—all of the above!
- It's important that default parameters are at the end



Keyword Arguments:

Why use keyword arguments?

- You may no see the value now, but it's useful when passing a dictionary to a function and unpacking it's values
- A little more flexibility

They are different from default params—when you define a function and use an = you are setting a default parameter.
When you invoke a function and use an = you are making a keyword argument.

Scope:

Where our variables can be accessed!

- Variables created in functions are scoped in that function

i.e.

```
# Global variable
        instructor = 'Colt'

        def say_hello():
                return f'Hello {instructor}'

# Local variable
        def say_hello():
                instructor = 'Colt'
                return f'Hello {instructor}'

        say_hello()

        print(instructor)       # NameError because variable is within the function and not
                                  accessible outside
```

- Global—lets us reference variables that were originally assigned on the global scope
- Nonlocal—lets us modify a parent's variables in a child (aka nested) function

You will not find yourself using the *global* or *nonlocal* keyword frequently—however, it is essential to understand for scope.

Documenting Functions:

It is essential to utilize documenting functions when it comes to writing complex functions. They can easily be accessed by using the '__doc__' method.

Recap:

- Functions are procedures for executing code. They accept inputs and return outputs when the return keyword is used
- To create inputs, we make parameters which can have default values, we call those default parameters
- variables defined inside of functions are scoped to that function—watch out for that!
- When invoking a function, we can pass in keyword arguments in any order, we'll see this more later!
- Be careful to not return too early in your conditional logic and refactor when you can to remove unnecessary conditional logic. Make sure you don't return in a loop too early as well!

Product

Write a function called product that accepts two parameters and returns the product of the two parameters (multiply them together)

exercise.py

```
'''
product(2,2) # 4
product(2,-2) # -4
'''

# define product below:
def product(num1, num2):
    return num1 * num2
```

Line 8, Column 23 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

return_day

Write a function called **return_day**. this function takes in one parameter (a number from 1-7) and returns the day of the week (1 is Sunday, 2 is Monday, 3 is Tuesday etc.). If the number is less than 1 or greater than 7, the function should return None

Hint: store the days of the week in a list (or dict using numbers as keys).

exercise.py

```
'''
return_day(1) # "Sunday"
return_day(2) # "Monday"
return_day(3) # "Tuesday"
return_day(4) # "Wednesday"
return_day(5) # "Thursday"
return_day(6) # "Friday"
return_day(7) # "Saturday"
return_day(41) # None
'''

def return_day(num):
    days = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
    if num > 0 and num <= len(days):
        return days[num-1]
    else:
        return None
```

Line 17, Column 20 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

last_element

Write a function called **last_element**. This function takes in one parameter (a list) and returns the last value in the list. It should return None if the list is empty.

exercise.py

```
'''
last_element([1,2,3]) # 3
last_element([]) # None
'''

def last_element(list):
    if list != []:
        x = len(list)
        return list[x-1]
    else:
        return None
```

Line 11, Column 20 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

number_compare

Write a function called **number_compare**. This function takes in two parameters (both numbers). If the first is greater than the second, this function returns "First is greater" If the second number is greater than the first, the function returns "Second is greater" Otherwise the function returns "Numbers are equal"

exercise.py

```
'''
number_compare(1,1) # "Numbers are equal"
number_compare(1,0) # "First is greater"
number_compare(2,4) # "Second is greater"
'''

def number_compare(num1, num2):
    if num1 > num2:
        return 'First is greater'
    elif num2 > num1:
        return 'Second is greater'
    else:
        return 'Numbers are equal'

```

Line 13, Column 34 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

single_letter_count

Write a function called **single_letter_count**. This function takes in two parameters (two strings). The first parameter should be a word and the second should be a letter. The function returns the number of times that letter appears in the word. The function should be case insensitive (does not matter if the input is lowercase or uppercase). If the letter is not found in the word, the function should return 0.

Hint: take advantage of count() method

exercise.py

```
'''
single_letter_count("Hello World", "h") # 1
single_letter_count("Hello World", "z") # 0
single_letter_count("HelLo World", "l") # 3
'''

#define single_letter_count below:
def single_letter_count(string, letter):
    return string.lower().count(letter.lower())
```

Line 8, Column 31 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

multiple_letter_count

Write a function called **multiple_letter_count**. This function takes in one parameter (a string) and returns a dictionary with the keys being the letters and the values being the count of the letter. Hint: use a dictionary comprehension and count().

Here's how it should work:

```
multiple_letter_count("awesome")   # {'a': 1, 'e': 2, 'm': 1, 'o': 1, 's':
```

exercise.py

```
'''
multiple_letter_count("awesome") # {'a': 1, 'e': 2, 'm': 1, 'o': 1, 's': 1, 'w': 1}
'''

# flesh out multiple_letter count:
def multiple_letter_count(string):
    return {letter: string.count(letter) for letter in string}
```

Line 7, Column 63 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

list_manipulation

Write a function called **list_manipulation**. This function should take in four parameters (a list, command, location and value).

- If the command is "remove" and the location is "end", the function should remove the last value in the list and return the value removed
- If the command is "remove" and the location is "beginning", the function should remove the first value in the list and return the value removed
- If the command is "add" and the location is "beginning", the function should add the value (fourth parameter) to the beginning of the list and return the list
- If the command is "add" and the location is "end", the function should add the value (fourth parameter) to the end of the list and return the list

exercise.py

```
'''

def list_manipulation(list1, command, location, value=None):
    if(command == "remove" and location == "end"):
        return list1.pop()
    elif(command == "remove" and location == "beginning"):
        return list1.pop(0)
    elif(command == "add" and location == "beginning"):
        list1.insert(0,value)
        return list1
    elif(command == "add" and location == "end"):
        list1.append(value)
        return list1

```

Line 18, Column 21 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

is_palindrome

Write a function called **is_palindrome**. A Palindrome is a word, phrase, number, or other sequence of characters which reads the same backward or forward. This function should take in one parameter and returns `True` or `False` depending on whether it is a palindrome. As a bonus, allow your function to ignore whitespace and capitalization so that `is_palindrome('a man a plan a canal Panama')` returns `True`.

exercise.py

```
'''
is_palindrome('testing') # False
is_palindrome('tacocat') # True
is_palindrome('hannah') # True
is_palindrome('robert') # False
is_palindrome('amanaplanacanalpanama') # True
'''

def is_palindrome(palindrome):
    stripped = palindrome.replace(' ', '')
    return stripped == stripped[::-1]

```

Line 11, Column 38 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

frequency

Write a function called **frequency**. This function accepts a list and a search_term (this will always be a primitive value) and returns the number of times the search_term appears in the list.

exercise.py

```
'''
frequency([1,2,3,4,4,4], 4) # 3
frequency([True, False, True, True], False) # 1
'''

def frequency(freq, search):
    return freq.count(search)
```

Line 7, Column 29 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

multiply_even_numbers

Write a function called **multiply_even_numbers**. This function accepts a list of numbers and returns the product of all even numbers in the list.

exercise.py

```
'''
multiply_even_numbers([2,3,4,5,6]) # 48
'''

def multiply_even_numbers(lst):
    total = 1
    for val in lst:
        if val % 2 == 0:
            total = total * val
    return total
```

Line 6, Column 5 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

capitalize

Write a function called **capitalize**. This function accepts a string and returns the same string with the first letter capitalized. You may want to use slices to help you out.

```
capitalize("jamaica") # "Jamaica"
capitalize("chicken") # "Chicken"
```

exercise.py

```
'''
capitalize("tim") # "Tim"
capitalize("matt") # "Matt"
'''

def capitalize(string):
    return string.capitalize()
```

Line 7, Column 31 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

compact

Write a function called **compact**. This function accepts a list and returns a list of values that are truthy values, without any of the falsey values.

```
compact([0,1,2,"",[], False, {}, None, "All done"])     # [1,2, "All done"]
```

exercise.py

```
'''
compact([0,1,2,"",[], False, {}, None, "All done"]) # [1,2, "All done"]
'''

def compact(l):
    return [val for val in l if val]
```

Line 7, Column 5 All changes saved Reset code

✔ Well done, your solution is correct!

Check solution Continue

Objectives:

- Use the * and ** operators as parameters to a function and outside of a function
- Leverage dictionary and tuple unpacking to create more flexible functions

*args:

- A special operator we can pass to functions
- Gathers remaining arguments as a tuple
- This is just a parameter—you can call it whatever you want

Rather than passing in multiple parameters within the function, we could use *args instead:

```
# Without *args
def add(a, b, c, d):
        return a+b+c+d

# With *args
def add(*args):
        result = 0
        for x in args:
                result += x
        return total
```

*args Exercise: The Purple Test

Define a function `contains_purple` that accepts **any number of arguments**. It should return `True` if any of the arguments are "purple" (all lowercase). Otherwise, it should return `False`. For example:

`contains_purple(25, "purple")` #True

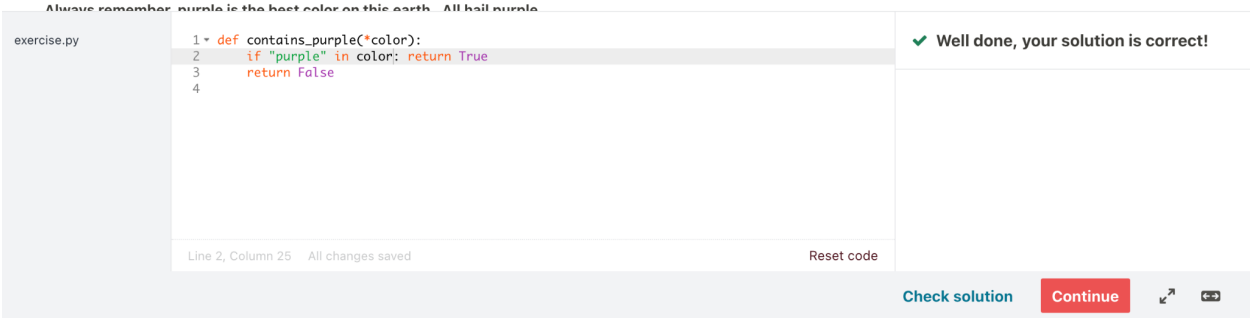
`contains_purple("green", False, 37, "blue", "hello world")` #False

`contains_purple("purple")` #True

`contains_purple("a", 99, "blah blah blah", 1, True, False, "purple")` #True

`contains_purple(1,2,3)` #False

Always remember, purple is the best color on this earth. All hail purple.



*kwargs:

- Keyword argument
- A special operator we can pass to functions
- Gether remaining keyword arguments as a dictionary
- This is just a parameter—we can call it whatever we want

**kwargs Exercise

Note: for this exercise, **make use of **kwargs**. No default parameters allowed!

Write a function called `combine_words` which accepts a single string called word and any number of additional key word arguments. If a prefix is provided, return the prefix followed by the word. If a suffix is provided, return the word followed by the suffix. If neither is provided, just return the word. It might sound confusing, but the examples should make this a lot clearer!

`combine_words("child")` *#'child'*

`combine_words("child", prefix="man")` *#'manchild'*

`combine_words("child", suffix="ish")` *#'childish'*

Parameter Ordering:

- parameters
- *args
- default parameters
- **kwargs

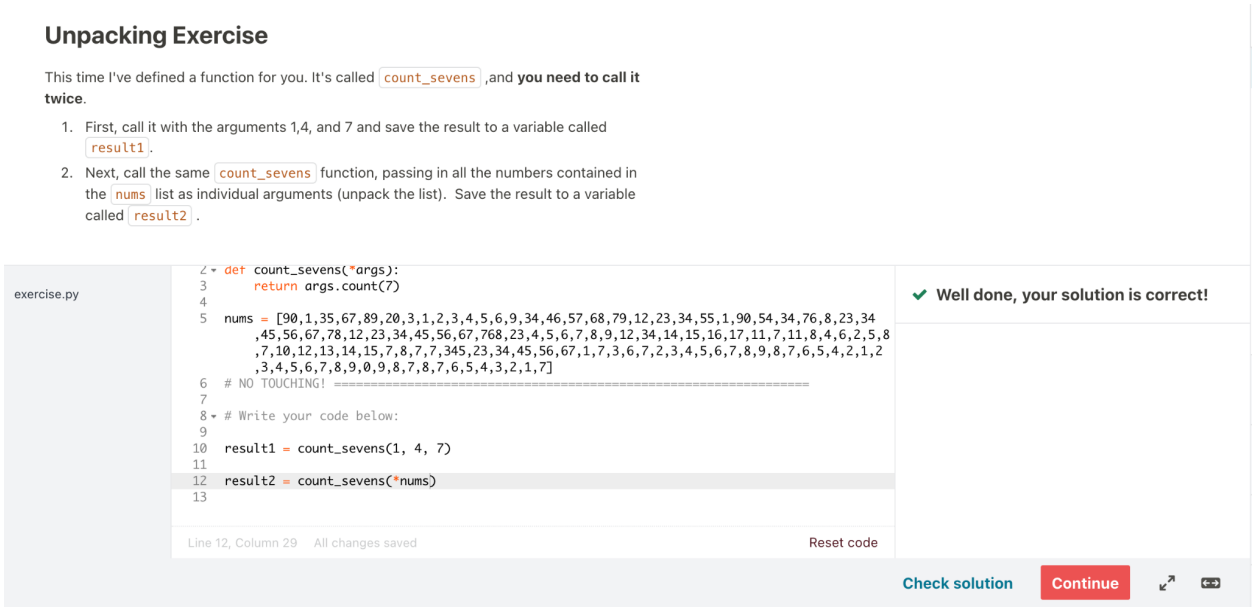
Using * as an Argument—Argument Unpacking:

- We can use * as an argument to a function to "unpack" values
- It will unpack values from a tuple

i.e.

```
def sum_all_values(*args):
        print(args)
        total = 0
        for num in args:
                total += num
        print(total)

nums = (1, 2, 3, 4, 5, 6)
sum_all_values(*nums)
```



Using ** as an Argument—Dictionary Unpacking:

- Similarly to using * as an argument, it will unpack the values as well.
- It will unpack the values from a dictionary