- **SECTION 6: METHODS AND FUNCTIONS, 2 hours 54 mins, 30 parts**
  - **1/29 Methods and the Python Documentation**
    - -> methods
      - built-in objects in Python have methods which can be used
      - **-> you can use OOP to create your own methods**
    - in JN
      - he creates and populates an array
      - array_name.append(4) <- adds 4 to the end
      - .pop() <- removes the last element of the list and saves it as that (this also prints the last element which is removed)
      - **shift tab <- to see a list of the different methods**
      - **help(array_name.insert) <- this returns help on it**
      - **-> there is also Python documentation online**
      - there is a blog post of the things which have improved / been updated in the latest version of Python
        - -> there are secure random numbers (secrets module) <- new in Python 3.6
        - -> **the use of documentation**
        - -> there is also a library reference and FAQs
          - **information about the classes / modules / objects in Python**
          - boolean / tuples / list types etc, collections, modules, file directory access
          - it's not readable
            - -> **stack overflow**
            - -> list operations -> min / max / indexing / slicing
          - there is an official list
            - -> e.g what sort does when you call it on a list
            - -> more information about the methods available on objects
            - -> **the more advanced you get the more answers you will only be able to find in the documentation (in a JN access it via shift tab)**
        - -> graphical interface FAQs

  - **2/29 Introduction to Functions**
    - -> functions
    - -> **repeatable code**
    - -> blocks of code which can be repeated -> without having to re-write it
    - -> **to call the same block of code using one line**
    - -> these allow for more complex solutions to problems
      - control flow
      - loops
    - -> **defining functions becomes more complex -> so this bootcamp is a slow incline to defining functions**
      - datatypes -> then loops and logic -> once you know about functions, you can solve problems which are a lot more complex
      - his advice
        - be patient with yourself
        - take time to practice the material
        - get excited about your skills and start thinking about new personal projects

  - **3/29 def Keyword**
    - -> def keyword
      - this allows the creation of functions
      - -> correct indentation and proper structure

- **-> def name_of_function(): <- define a function**
  - **about**
    - **-> name_of_function <- this is all lowercase 'snake casing' vs camel casing - with all lowercase and underscores in between the words**
    - **-> the ()'s <- take the variable in the argument and pass it into the function**
    - **-> : to show everything inside the function which is indented**
    - **-> '''**
    - **''' <- this is a docstring which explains the function**
  - print("hello")
- -> then to call the function, you type the name of the function
- **-> another example is passing (name) as the argument of the function then -> print("Hello" + name) <- name should be a string, which we concatenate with hello**
- -> return keyword
  - **return <- this allows us to assign the output of the function to a new variable**
    - **(it actually outputs the value, rather than just printing it out)**
  - **-> return num1 + num2 <- you don't need brackets when you use return**
  - -> to print the result and use it elsewhere in the code

- **4/29 Basics of Python Functions**
  - -> functions <- the name of the JN notebook (this includes tuple unpacking)
  - in JN
    - def say_hello(name='Jose'): <- **name is the argument, and the default name is Jose -> without a default then when the function is called without the argument, it will return an error message**
      - about
        - anything inside this indentation is called in the code
        - -> if you put a lot of different print statements in this and then call the function -> all of them get printed out on separate lines
        - -> **to call the function its say_hello() not say_hello <- or else it will just return that this thing is a function**
      - **print(f'Hello {name}') <- this is a string literal**
    - another example
      - **def add_num(num1, num2):**
        - **return num1, num2 <- this prints out the result when the function is called, and the result of that can be set equal to a variable (in comparison to using print)**
          - **-> if you used print and not return then you couldn't set that function call equal to a variable (it's a NoneType), the value isn't returned - something is just printed out**
    - you can use a print and a return statement when you define the function
      - -> then when its called the result is printed and you can set the function call equal to a variable
    - if you need to check the type of the input of the data into the function
      - **-> e.g if you define a function which adds two numbers together -> there is a case where strings are put as arguments into the function instead of numbers**
      - -> **you need to check the type of the data before returning the result (in this case - adding numbers not strings)**

- **5/29 Logic with Python Functions**
  - adding logic into internal function operations
    - -> functions with logic
    - -> checking if a number is even
  - in JN
    - mod operator
      - **the remainder -> if a number mod 2 is 0 then the number is even because its divisible by 2 and has no remainder**
      - **-> 3%2 <- in comparison to maths (where || is the modulus, which is the positive version of a number)**
      - -> 20%2 == 0 <- this returns True, 20 is an even number
    - to put this into a function
      - def even_check(number):
        - result = number % 2 == 0 <- in other words, a boolean which checks if the input is even
        - return result <- then we return the result
        - alt. you can define the function as return number % 2 == 0
    - to use this with a list
      - -> return true is any number is even inside a list
      - -> def check_even_list(num_list):
        - for num in num_list:
          - if number % 2 == 0: <- if the number is even, we return true (if at least one of the numbers in the list are even)
            - **return True <- you can put returns inside a for loop (rather than just at the end of the function which you're defining)**
          - **else:**
            - **pass <- this means 'don't do anything'**
          - so
            - -> we're iterating through the list and if one of the elements in the list is even (it mod 2 is 0), then it returns True
            - **-> this function can have multiple returns**
        - to return False (in the above function)
          - -> you can't put return False under the else loop
          - -> because if there was another even number in the function, it would return False
          - -> so she's put the return False under a separate block of code (indented it later elsewhere)
        - to return all of the even numbers in the list (in the above function)
          - **-> she's defined an empty array ("placeholder" array) -> to hold the list of even numbers**
          - -> then changed the code under the indented even condition -> to append that number to the even list
          - -> **concatenate is to add two thing together, to append is to add something to the end of it**
        - **then he's checking the function on some test arrays**

- **6/29 Tuple Unpacking with Python Functions**
  - defining functions and returning multiple results in Python
  - in JN
    - **he's defined an array of tuples -> [(x,y),(,),(,)] <- essentially coordinates inside an array**

    - for item in stock_prices:
      - print(item) <- printing each of the tuples by iterating through the list
    - another example
      - **for ticker, price in stock_prices: <- this is (x,y)**
    - another example
      - we have tuples of x and y
      - -> we want the y with the maximal x
      - -> def function_name(array_name):
        - #she's initialising the values to return
        - current_max = 0
        - employee_of_month = ''
        - #then we're iterating through the array of tuples and updating the current_max -> and otherwise it's a pass
        - #return
        - return(employee_of_the_month, current_max)
      - -> then she does an example and it prints the maximum
      - -> **if the function outputs two things (e.g return x,y - then you can call the function and write x,y=function_call())**
    - if you are reusing someone elses' function
      - -> e.g if it takes tuples then there can 'not be enough values to unpack' (it will tell you)
      - -> in this case, you can set everything equal to a variable, and then inspect the variable

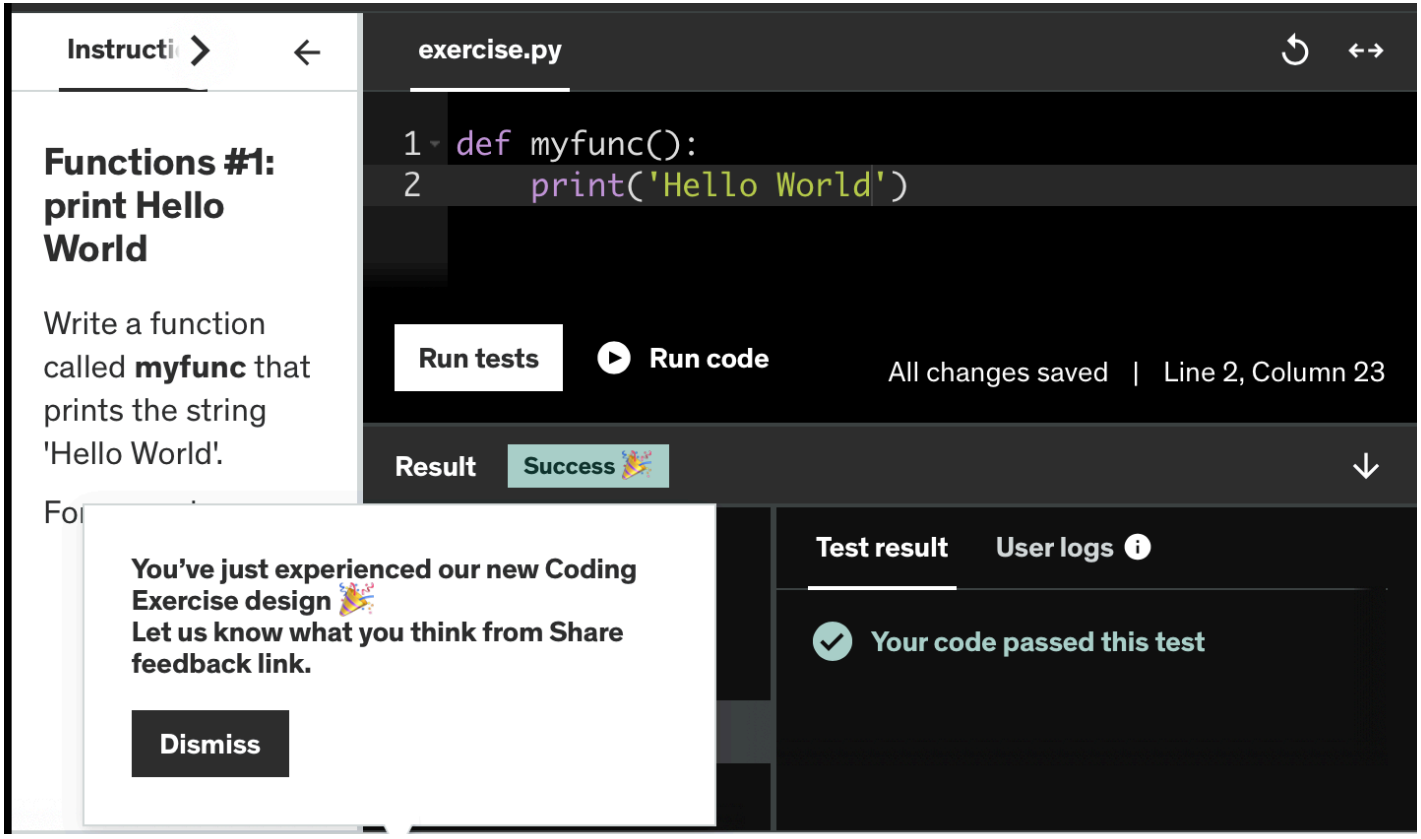
- **7/29 Interactions between Python Functions**
  - -> Python notebooks contain multiple functions interacting with each other
  - -> creating multiple functions to mimick the red ball monte game (a ball under one of three cups) <- so a Python list with two empty strings and then one string with 'o' for the ball
  - -> this is to show multiple functions interacting with each other
  - in JN
    - from random import shuffle <- this is importing shuffle
    - example = [1,2,3,4,5,6,7,8,9,10]
    - shuffle(example) <- this stores example as example all randomly shuffled (it's returning a NoneType, the shuffled list has been stored as example
    - defining the first function <- **when problem solving, you can define multiple functions at each state of the problem solving process, then combine them for the result, which you interpret etc**
      - def shuffle_list(mylist):
        - shuffle(mylist)
        - return mylist <- so we've defined a function which takes the input list and shuffled it (it returns the same list, shuffled)
      - then she's defined a second function for the red ball monte game
        - inside the function
          - guess = ''
          - **while guess not in [,,,]: <- it keeps on asking for a guess, until it's in that list**
            - guess = input("...")
          - -> then return int(guess)
          - -> so this function takes a guess
      - then she defines a third function the two functions
        - -> def check_guess(mylist,guess):

        - if mylist[guess] == 'o':
            - print("Correct")
        - else:
            - print("Wrong")
    - then she combines all three functions into a final function
        - **-> the JN is basically a giant calculator for problem solving -> putting different parts of the problem into different functions makes the problem solving thought process clearer**
        - **-> this allows for more complex problem solving**

- **8/29 Overview of Quick Function Exercises #1-10**
    - -> function defining exercises
    - -> part 1 -> 10 problems / exercises for the syntax of functions
    - -> lecture 26 has useful warmup problems
        - -> https://docs.google.com/document/d/181AMuP-V5VnSorl_q7p6BYd8mwXWBnsZY_sSPA8trfc/edit?usp=sharing <- there are solutions to some problems at this URL
    - -> then there is a JN with problems, with an overview lecture
    - -> the problems increase in difficulty
    - -> then function and methods homework

- **9/29 Quiz 8: Quick Check on Solutions Link**
    - -> there are 10 questions, solutions are here https://docs.google.com/document/d/181AMuP-V5VnSorl_q7p6BYd8mwXWBnsZY_sSPA8trfc/edit?usp=sharing

- **10/29 Coding Exercise 10: Functions #1: print Hello World**



Functions #1: print Hello World

Write a function called **myfunc** that prints the string 'Hello World'.

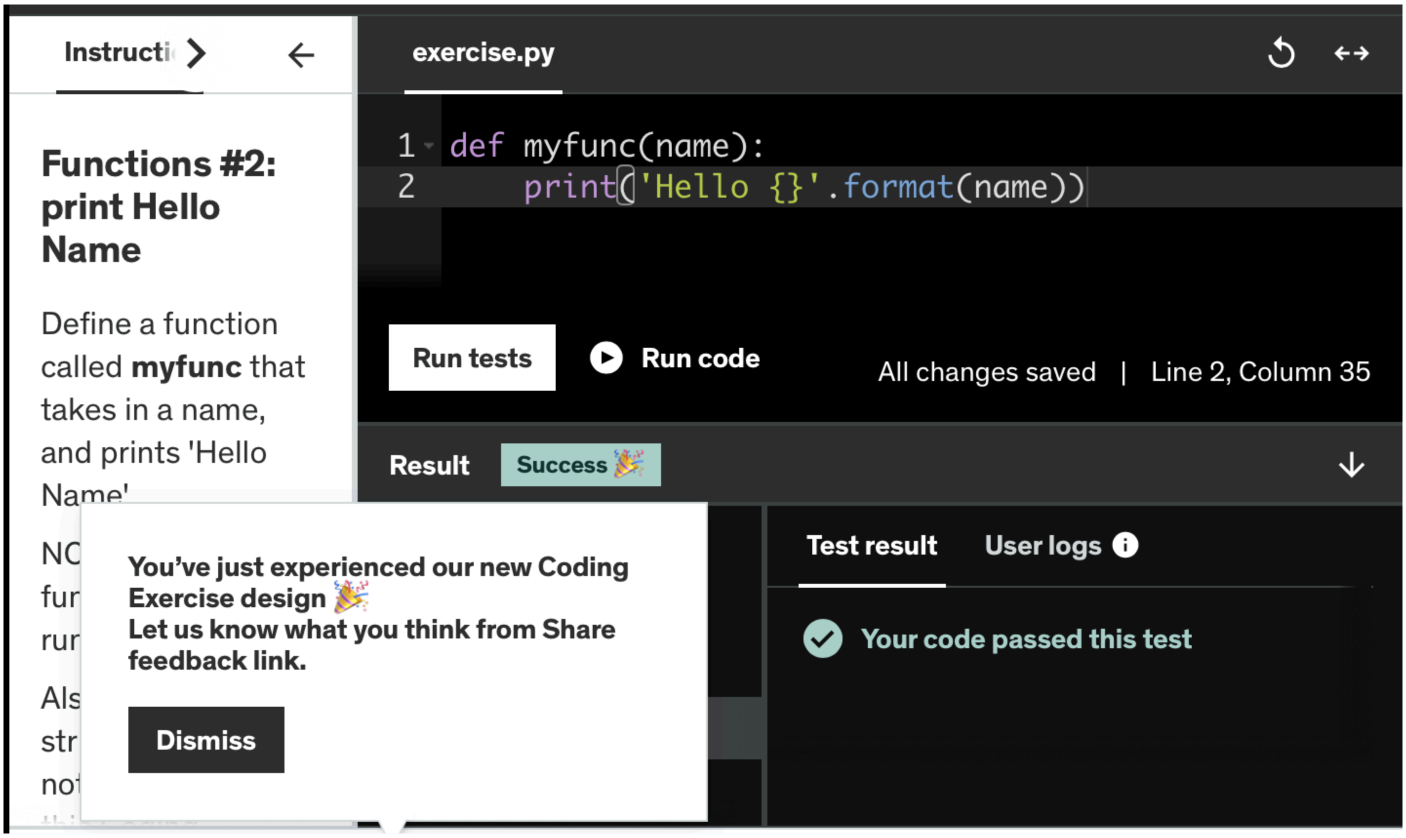
Fo

```python
def myfunc():
    print('Hello World')
```

Your code passed this test

You've just experienced our new Coding Exercise design
Let us know what you think from Share feedback link.

- **11/29 Coding Exercise 11: Functions #2: print Hello Name**
    - we want a function which uses .format to print the name



Functions #2: print Hello Name

Define a function called **myfunc** that takes in a name, and prints 'Hello Name'

NC
fur
rur

Als
str
not

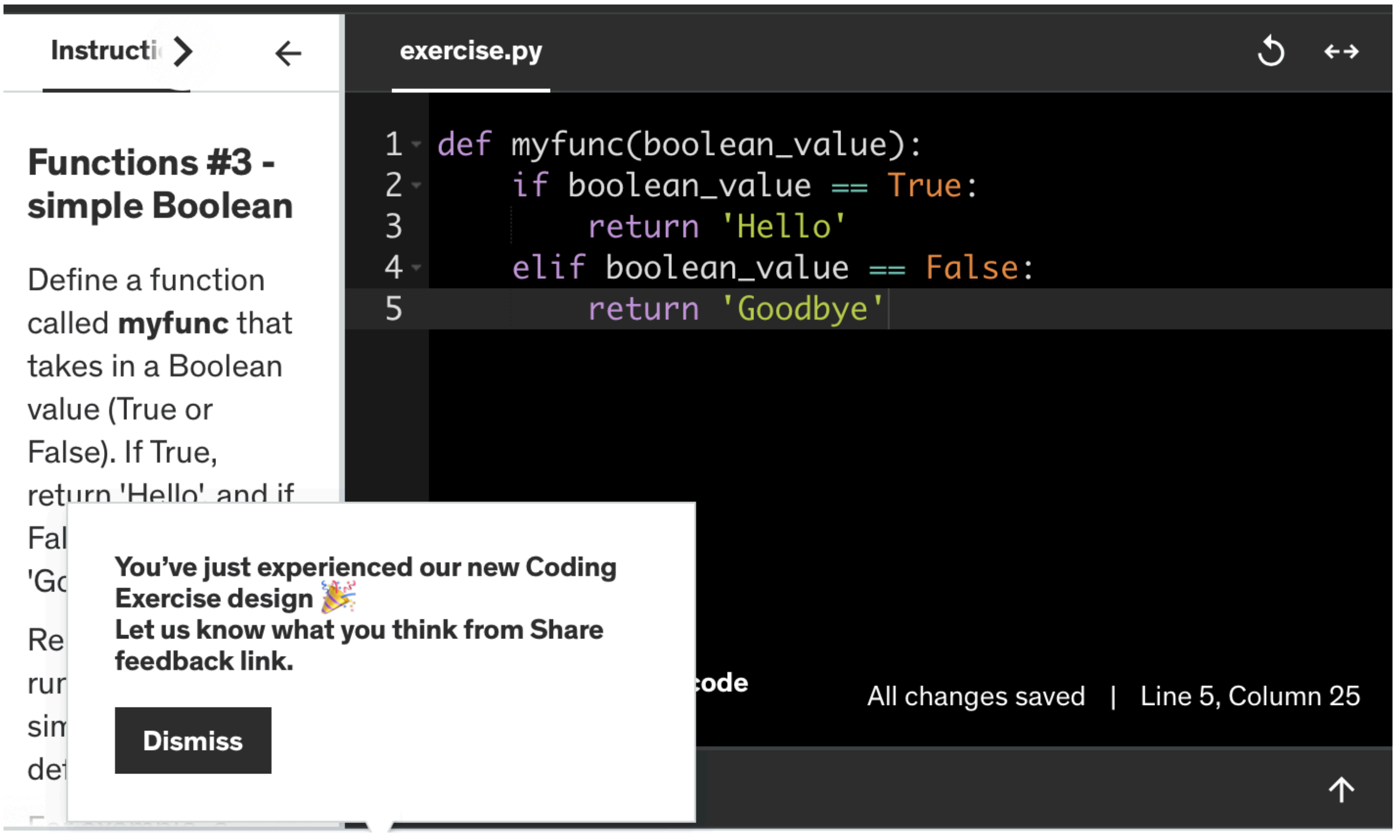
```
def myfunc(name):
    print('Hello {}'.format(name))
```

All changes saved | Line 2, Column 35

Your code passed this test

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

- ○ **12/29 Coding Exercise 12: Functions #3 - simple Boolean**



Functions #3 - simple Boolean

Define a function called **myfunc** that takes in a Boolean value (True or False). If True, return 'Hello' and if Fal
'Gc

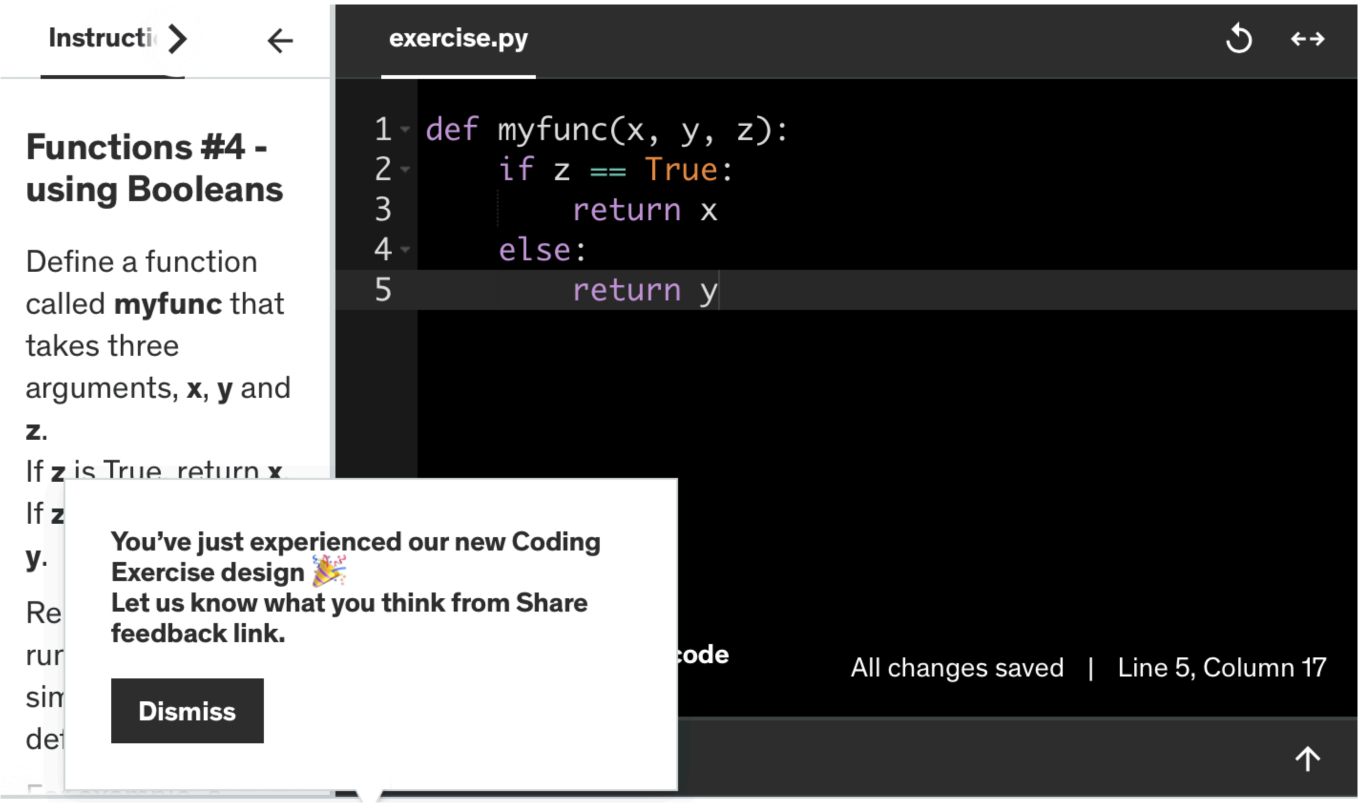
Re
rur
sim
det

```
def myfunc(boolean_value):
    if boolean_value == True:
        return 'Hello'
    elif boolean_value == False:
        return 'Goodbye'
```

All changes saved | Line 5, Column 25

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

- -> we are saying -> if True is entered into the function, then return Hello -> and else Goodbye (assuming that the function takes a boolean as the argument)



Functions #4 - using Booleans

Define a function called **myfunc** that takes three arguments, **x**, **y** and **z**.
If **z** is True, return **x**.
If **z**
y.

Re
rur
sim
def

```python
def myfunc(x, y, z):
    if z == True:
        return x
    else:
        return y
```

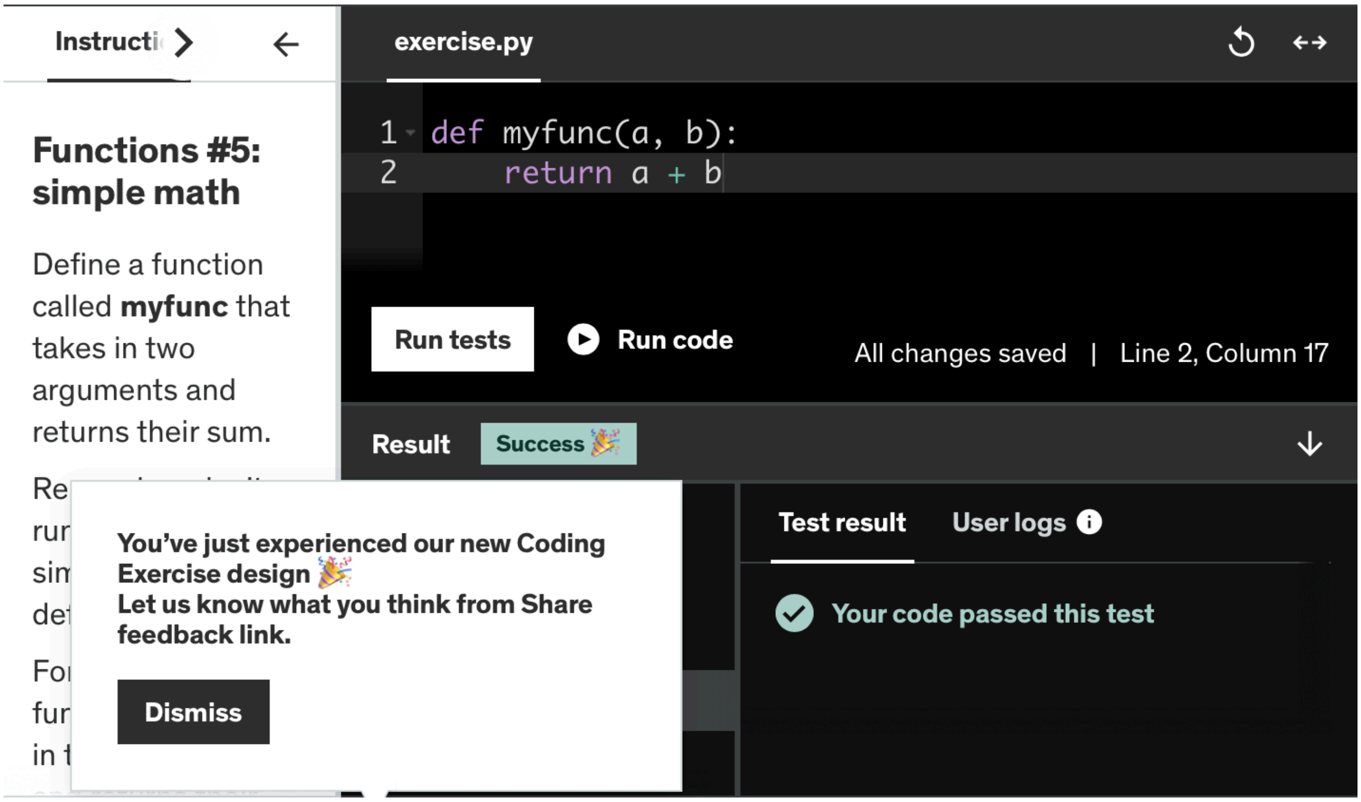
You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

code

All changes saved | Line 5, Column 17

- **13/29 Coding Exercise 13: Functions #4 - using Booleans**
  - -> we have defined a function which takes three arguments
  - -> the the third argument is True, then we return the first argument
  - -> if the third argument is False, then we return the second argument

- **14/29 Coding Exercise 14: Functions #5: simple math**



Functions #5: simple math

Define a function called **myfunc** that takes in two arguments and returns their sum.

Re
rur
sin
det

Fo
fur
in t

```python
def myfunc(a, b):
    return a + b
```

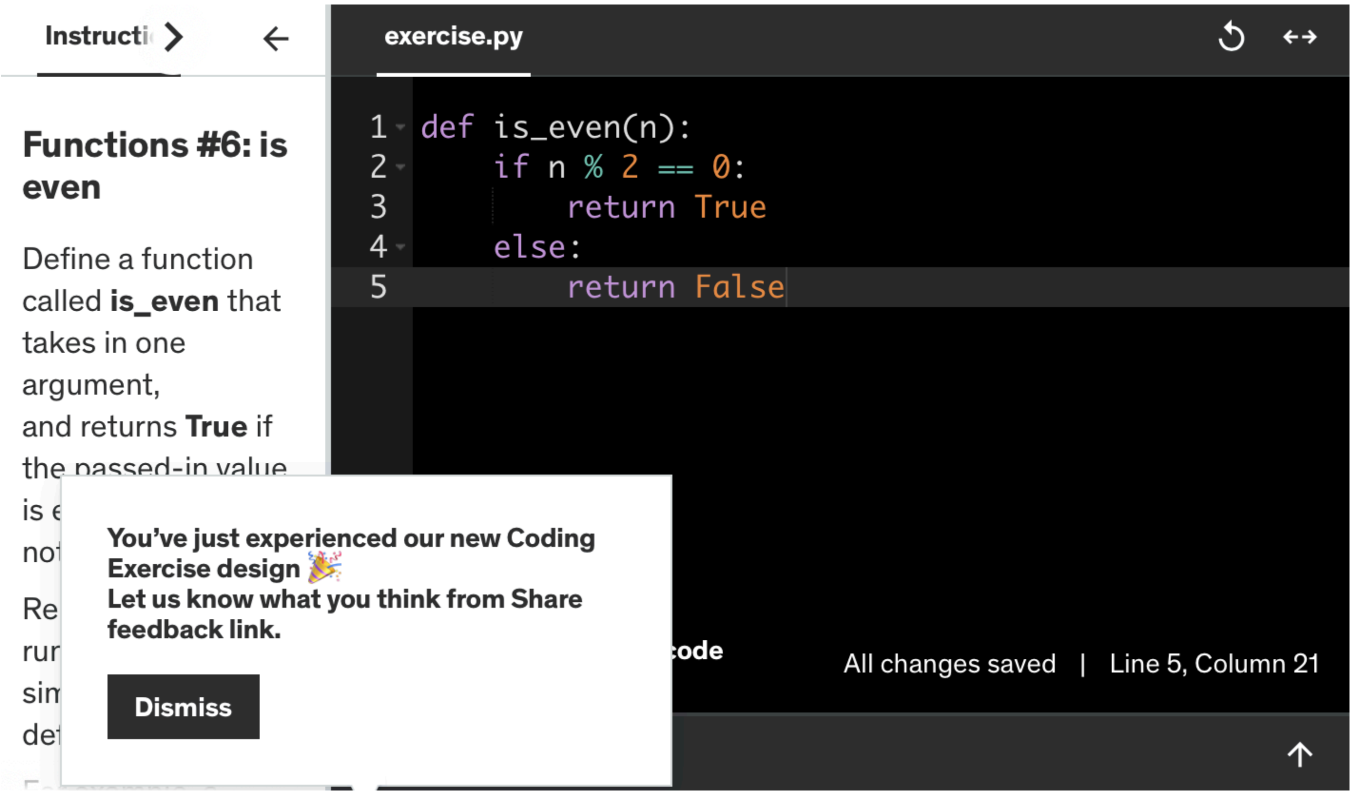
All changes saved | Line 2, Column 17

Your code passed this test

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

  - -> we've defined an adding funciton (which returns the sum of the arguments -> rather than prints them)
  - -> this means we can assign a variable equal to the function when it is called (rather than just using print() in the function definition)

- **15/29 Coding Exercise 15: Functions #6: is even**



Functions #6: is even

Define a function called **is_even** that takes in one argument, and returns **True** if the passed-in value is e

no

Re

rur

sin

def

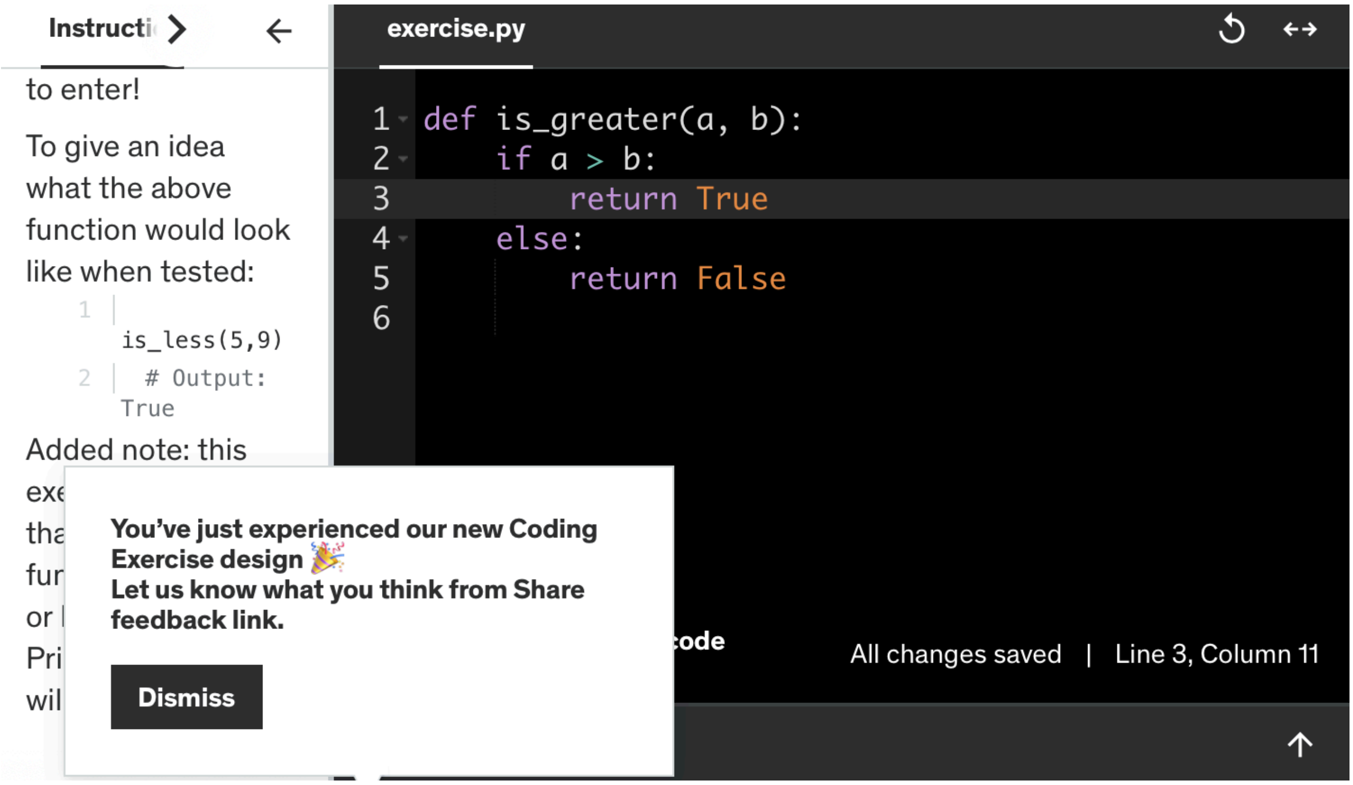
```python
def is_even(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

All changes saved | Line 5, Column 21

- -> we've defined a function which tests if the argument is even or not
- -> if it is even, then it mod (%) 2 is zero - because there is no remainder when it is divided by two -> and in this care, True is returned
- -> otherwise, we return false
- **-> we've used an else rather than an elif statement, because there is only one condition (if we used elif, we could have entered another condition)**

- **16/29 Coding Exercise 16: Functions #7: is greater**



to enter!

To give an idea what the above function would look like when tested:

```
is_less(5,9)
# Output: True
```

Added note: this

exe

tha

fur

or l

Pri

wil

```python
def is_greater(a, b):
    if a > b:
        return True
    else:
        return False

```

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

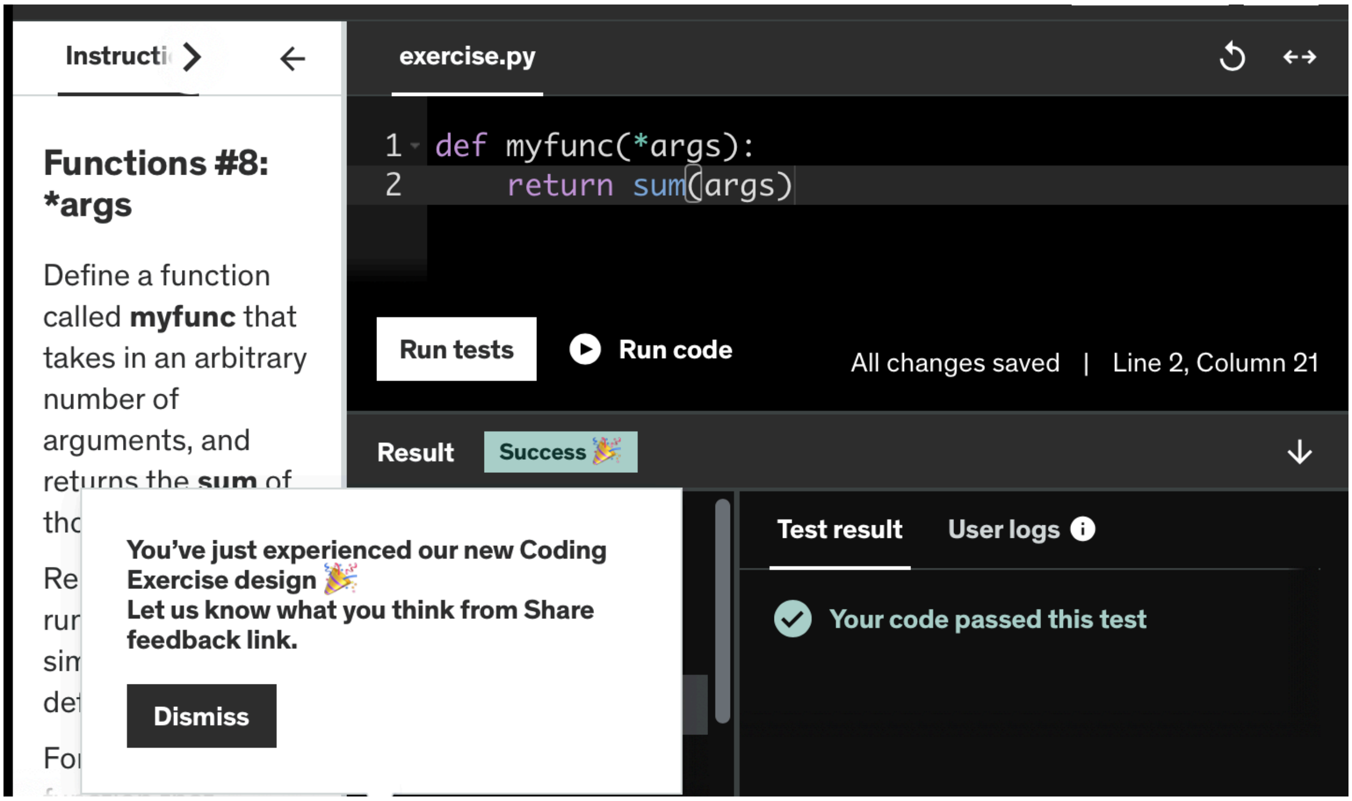
All changes saved | Line 3, Column 11

- -> we've defined a function with two conditions -> if one of the arguments is greater than the other then we return True
- -> we don't use elif as there is only one condition

    - -> the function returns a boolean value

- **17/29 *args and **kwargs in Python**
    - ***args and **kwargs are functional arguments -> arguments and keyword arguments**
    - -> you want to accept an arbitrary number of arguments
    - in JN
        - -> def myfunc(a,b):
            - return 0.05*(a+b)
        - -> a and b are positional arguments
        - **-> if you want a third argument added to the function**
            - **you can define other input parameters (e.g d=0)**
            - **alt., def myfunc(*args):**
                - **-> this treats it as a tuple of arguments**
                - **return sum(args) * 0.05**
            - **myfunc(40,60,....) e.g <- *args is for all arguments (all the arguments which are entered) -> def(*args)**
            - **-> instead of *args, you could also enter *spam, e.g**
                - **then you -> for item in args, e.g you can run a for loop**
    - **for a dictionary of key value pairs when defining a function**
        - **def myfunc(**kwargs) <- for any amount of arguments**
        - -> for 'fruit' in kwargs: <- e.g
            - -> then indented a formal
            - -> print("....{}".format(kwargs['fruit]))
            - **-> then you can have any amount of inputs and it's creating a dictionary**
            - **-> ** creates a dictionary and * doesn't**
        - -> another one is def myfunc(*args, **kwargs):
            - print('I would like {} {}'.format(args[0],kwargs['food']))
            - -> then you can input numbers and then text e.g -> the first ones entered are args and the second is kwargs
            - -> this is useful for outside libraries

- **18/29 Coding Exercise 17: Functions #8: *args**



Functions #8: *args

Define a function called **myfunc** that takes in an arbitrary number of arguments, and returns the **sum** of

tho

Re
rur
sim
de
Fo

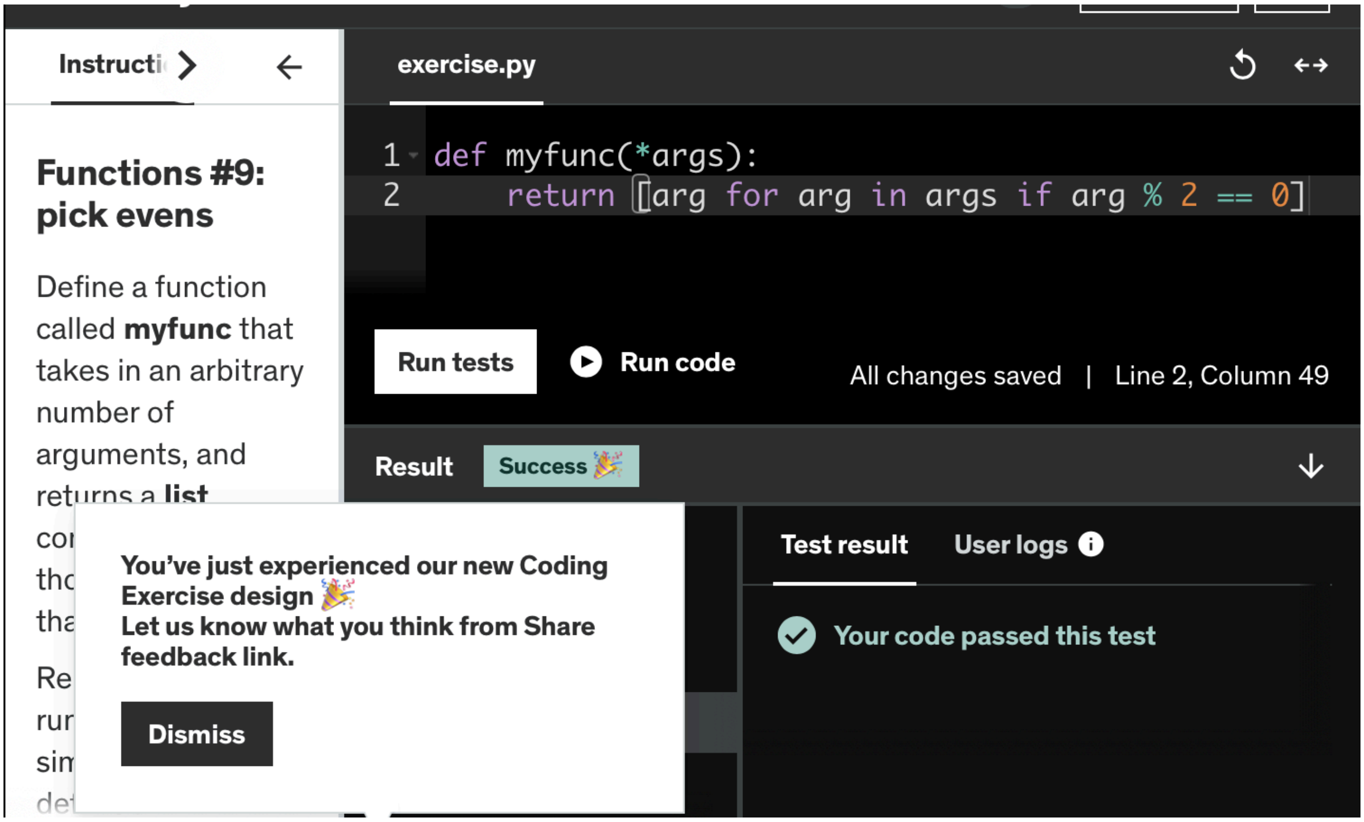
```
def myfunc(*args):
    return sum(args)
```

Your code passed this test

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

  - **-> we've defined a function which takes an unlimited amount of arguments, sums them then returns them**
  - **-> think of * as representing all of the possible arguments -> all of the arguments (there are an unlimited amount of arguments in the definition of this function)**

- **19/29 Coding Exercise 18: Functions #9: pick evens**



Functions #9: pick evens

Define a function called **myfunc** that takes in an arbitrary number of arguments, and returns a **list**

co

th

tha

Re

rur

sim

det

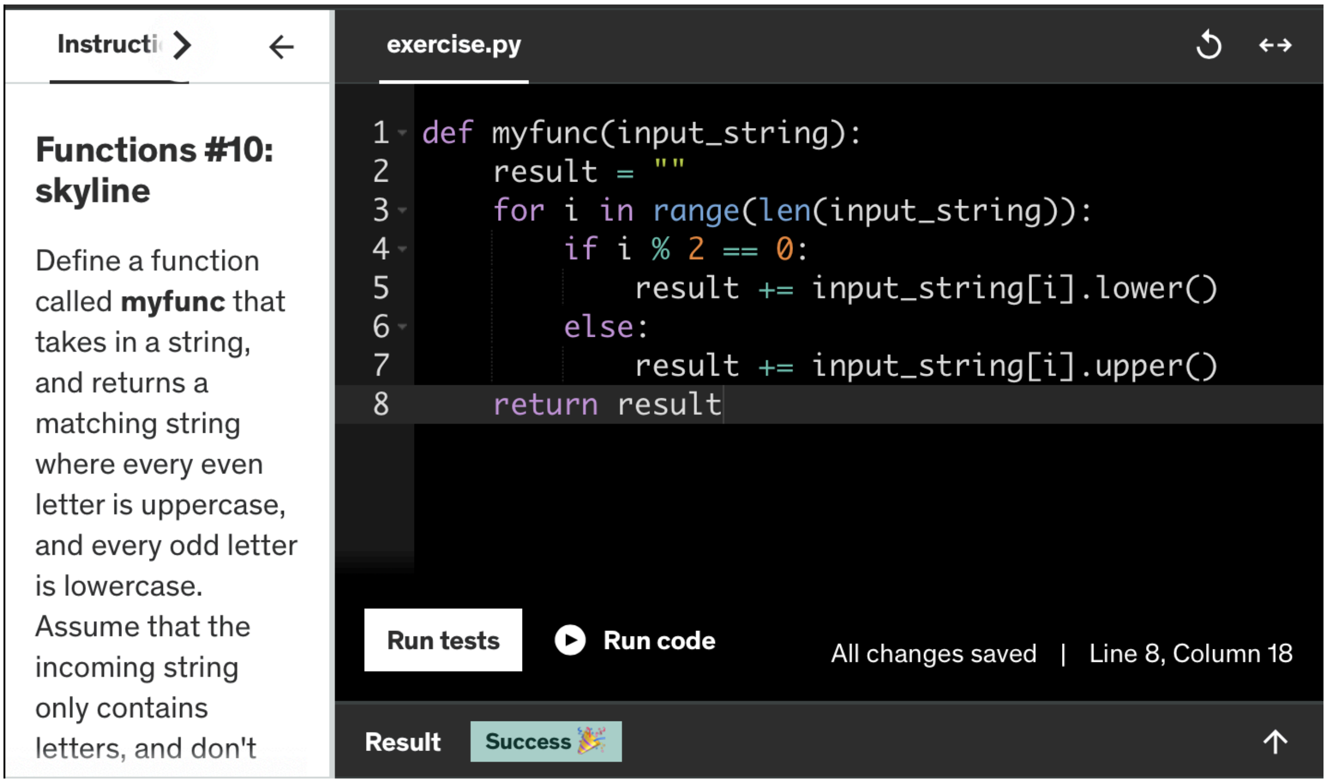
```
def myfunc(*args):
    return [arg for arg in args if arg % 2 == 0]
```

You've just experienced our new Coding Exercise design 🎉
Let us know what you think from Share feedback link.

✓ Your code passed this test

  - -> we've defined a function which takes an unlimited amount of arguments
  - -> then used a list comprehension
  - -> we've done this to return the even numbers in the function (if the number modulo 2 is zero then it's remainder with 2 is 0 and therefore it's even)

- **20/29 Coding Exercise 19: Functions #10: skyline**



Functions #10: skyline

Define a function called **myfunc** that takes in a string, and returns a matching string where every even letter is uppercase, and every odd letter is lowercase. Assume that the incoming string only contains letters, and don't

```
def myfunc(input_string):
    result = ""
    for i in range(len(input_string)):
        if i % 2 == 0:
            result += input_string[i].lower()
        else:
            result += input_string[i].upper()
    return result
```

  - problem solving thought process
    - -> we've initialised a string which we are returning
    - -> then we define a for loop iterating through each character in the string
    - -> if the element in that string is even (i.e it modulo, %, 2 is zero) -> then we are adding it to the result which we later return
      - -> we use i because this is the index of that character in the input string
      - -> we populating the result using the += approach, and doing so using .lower()
    - -> if we are dealing with anything else -> we are using the same process, but adding the uppercase form of the string to the returned result
    - -> this gives us a function which takes a string and outputs the same thing but with all of the even elements as lowercase and other elements as uppercase

- **21/29 Function Practice Exercises - Overview**
  - -> function practice problems
  - -> Python problems
  - -> the more Python you learn, the higher the difficulty of the problems you can solve
  - -> this is a general problem to solve and deciding what the best way to solve it is
  - -> i.e this lecture is example problems for functions
  - in JN
    - -> the methods notebook on the git page for the course
    - -> there is a notebook called function practice exercises
    - -> the problems are in increasing difficulty -> if then condition statements and using iterations in the solutions
    - -> the example
      - a function which returns the lesser function if both are even
      - if one is odd or both are odd, then it returns the larger argument
      - -> the entire idea is - a lot of if else conditions
      - **-> you have to think about all of the different case scenarios when you are solving the problems**

- **22/29 Function Practice Exercises - Solutions**
  - -> this is the solution to the question from the previous video
  - -> we're first checking if both numbers are even -> there is an if a %2 ==0 and b%2==0:
    - **this comes up a lot in questions, the condition for an even number is a%2==0**
    - -> so we are saying -> if both the numbers are even - do this
    - -> else - do this
    - -> **and nesting the if statements**
      - **-> a lot of it is thinking about the different cases for what the inputs to the functions would be**
      - **-> writing the function**
      - **-> then testing the function with different possible inputs, and cleaning up the code**
        - **-> she's replacing sections of code with a min function (with an entire function) - rather than repeating sections of the code**
        - **-> she's also written return in the code -> rather than storing the result then returning it at the end**
        - **-> the efficiency of solutions (in as few lines as possible while it still being understandable)**
  - -> another question
    - -> takes a two word string and returns True if they begin with the same letter
    - **-> build out the logic and then make it (the solution) more compact**

        - -> we take the two input words and split them -> text.split()
        - -> then return the_first[0] == the_second[0]
        - **-> she's also thinking about the different possible inputs to the function (e.g if a character is lowercase or uppercase), and we are also assuming the input has a certain datatype**
    - -> another question
        - -> true if one of the numbers is 20, or if both of them sum to 20
        - -> **we have a set of conditions in the question, and then in the solution we are converting the conditions into code (like in maths when you convert the text in the question into simultaneous equations -> we are coding the problem)**
        - **-> once we have coded the problem -> then she is making the solution more compact / concise -> return (....) or (....) or (....) <- we are checking for three different boolean conditions**
            - **you can have ultra compact code or the same thing over a lot of separate if etc else conditions**

- **23/29 Function Practice - Solutions Level One**
    - -> this is going over the solutions to the previous JN again
    - -> example question
        - we want a function which capitalises the first and fourth letters of a name
        - **-> she's broken up the input string, changed the cases of the elements and then added them all together in a return statement**
        - -> adding the strings is the same as concatenating them
        - **-> then she's writing out the same solution in a more compact notation**
    - -> another example question
        - -> we want the function to return the same word / list spelt backwards
        - **-> wordlist = text.split() <- make a list of the string (each word int he input string to the function is the element of a list)**
        - **-> then return wordlist[::-1] <- go through the entire thing and spell it backwards**
        - **-> another approach is to use '--'.join(mylist) <- this puts a space between every element in the list -> it's joining them together (join), separated by --'s**
    - -> another example question
        - **abs(num) <- this returns the absolute number (positive version)**
        - -> return (abs(100-n) <= 10) or (abs(200-n)<=10)
            - -> so is the difference between the number and 100 or 200 less than or equal to 10
            - **-> the return statement from a function can be a boolean condition (this makes the solution more compact)**

- **24/29 Function Practice - Solutions Level Two**
    - -> level 2 solutions to example problems
    - -> we have a list of integers and want to return True if we have a 3 next to another 3 in the array
    - -> she's defined a loop -> Python is zero indexed, so we're going from 0 to the length of the array-1
        - we're going through a for loop
            - -> for each element in the for loop, we're checking that element and the element at the index (i+1) to see if they are both true
            - -> the other thing is, we have two return statements -> one is True and the other is False

        - -> **you don't have to have one return statement at the end of the the entire function definition**
    - -> another example
        - given a string return another string -> the would return ttthhheee e.g
        - -> this is done via for char in text -> and then populating an empty string
        - -> result += char*3 <- you are adding to that result
    - -> another example
        - blackjack
        - the blackjack card game
        - -> three integers between 1 and 11 -> there are three different conditions
        - -> the thought process is the same
            - **given the conditions in the question -> you put the conditions in the question into code**
            - **-> and for each of those conditions, she's written a return statement**
            - **-> then at the end gone back to make the entire thing more compact**
            - -> **the notation she's used is sum([a,b,c]) <- rather than using a loop to add all the elements together**
            - -> **the difference between an elif statement and an else statement is that elif statements take boolean statements / logic conditions for control flow**
    - -> another example
        - -> return the sum of the numbers in the array, but ignoring the numbers which are after a 6 in the array
        - -> she's first initialised a counter at 0 and a boolean
        - -> she's then iterated through the elements in the array -> and added break conditions wherever the numbers were 6
            - -> using !- in this example
            - -> she's also used add and while not statements
            - -> the break is only connected to the while loop it's in

- **25/29 Function Exercise Solutions - Challenge Problem**
    - -> more challenge problem examples
    - -> one example
        - function which takes a list of integers and returns True if it contains 0, 0, and 7 in the array
        - -> she's defined an array inside the function - iterating through that [0,0,7] list is popping the numbers off that list if they appear in the input string
        - -> and the case that all the numbers have been popped off -> then the input array involves 0,0,7 -> in which case we define a return True statement
        - -> the argument of the pop function is the index of the element which is popped of in the list
        - so she's iterating through the list and then if the element in the list matches the one in the list with 0,0,7 in it and then it pops off the elements in that list -> until there are none left (if there are none left)
        - -> use logic to find the consecutive numbers
    - -> another example
        - write a function which returns the number of prime numbers up to and including the argument of the function
        - def count_primes(num):
            - if num < 2:
                - return 0 < this is the check for the 0 or 1 input
                    - -> **the thought process is to think about each of the possible inputs to the function**

- **-> and then structure the code according to the case scenarios**
- -> in other words, if the input < 2 then there are no primes counting up to the number
- -> then she's creating another block of code
    - an array to store the prime numbers and populate it with
    - then a counter going up to the input number
    - she's made a counter going up to the input number
    - -> then checked if there is a prime number using modulo
    - -> if then added it to the primes list (array)
    - -> if the number itself is prime, then she's added a block of code before that which breaks the iteration
- **-> the Euler project has problems similar to this**
- -> functions which print out results rather than return them

- **26/29 Lambda Expressions, Map, and Filter Functions**
    - **-> lamda expressions map and filter**
        - **-> these are expressions to create anonymous (one time use) functions**
        - **-> like Einstein’s lambda fudge**
    - -> in JN
        - **map <- then shift tab for the documentation, it expects func and then *iterables (* is for any amount of them)**
        - -> def square(num):
            - return num**2
        - my__nums = [1,2,3,4,5]
            - we want to apply the square function to every item in the list -> **you could use the map function or a for loop**
            - **map(square,my_nums) <- use a function which works for a single number, but instead maps it to an entire array of values**
        - another example
            - **map is to - in this case, map a function which works on a single number to an entire array - it's saying - apply this function to an entire array of values**
            - -> def splicer(mystring):
                - if len(mystring)%2 ==0: <- then there are an even number of characters in the string
                    - return even
                - else:
                    - return mystring[0]
            - -> then she's testing the function on a list of strings
            - -> so we have a function which works on a single string
            - **-> then she uses map(name_of_function, array_name) <- and it runs the function on an entire array, without doing a loop**
        - **the filter function**
            - returns an iterator yielding the items of an iterable
            - -> def check_even(num):
                - return num%2 ==0
                - -> **she's defined a function which returns True of False**
            - -> then defined an array of numbers
            - -> **filter -> we have one function which takes a number and either returns true of false**
                - **list(filter(function_name, array_name))**
                - **-> this returns the elements in the list which return True when they are**

passed through the function

    - converting a function step by step into a lambda function
      - -> def square(num):
        - result = num**2
        - return result
      - -> it's a function which squares the input
      - -> to turn it into a lambda expression / function
      - -> she's simplified the function
        - def square(num): return num**2
        - -> we don't give it a name
        - **-> lambda num: num**2 <- this is a lambda expression**
        - **-> with the 'map' -> this is where lambda expressions are useful**
        - **-> list(map(lambda num: num**2, mynums))**)
        - -> it's a similar idea with the filter function.
          - filter(filter(lambda num:num%2 ==0,mynums))
    - list(map(lambda x:x[::-1],names))
      - -> not every function can be translated to a lambda expression
      - -> you need to be able to understand / read it when you come back to the code later

- **27/29 Nested Statements and Scope**
  - -> nested statements and scope
    - how to write functions
    - how Python deals with the variable names we assign
      - **variable names are called in name space**
  - in JN
    - x=25
    - def printer():
      - x = 50
      - return x <- **if you call the function the value of x which is the global variable is printed (not the x which is stated in the local / definition of the function)**
      - **-> scope allows Python to have a set of rules to decide which variables we are referencing in the code -> LEGB - local enclosing global built-in <- this is the order in which values are called in the name space**
      - -> this applies if e.g you have a function in a function
      - -> built-in are highlighted in certain colours
    - -> local variables example
      - in a lambda function
      - -> you need to list(the lambda function in here)
    - -> def greet():
      - name = 'Sammy'
      - def hello():
        - print('Hello ' +name)
      - hello()
    - greet()
    - -> so
      - -> we set the variable name equal to Sammy
      - -> then inside the hello function we are printing out the local variable name
      - -> the local variable is defined in the function definition
      - -> name is defined within that function
      - **-> she's commented out one of the lines of code and one of the variable names has been replaced with the next in LEGB -> the global value of that**

            variable rather than the local is now being used (in the "global namespace")
        - -> if x has multiple values in the JN -> then the one it uses when x is called is in order of LEGB
    - -> another example
        - **if you call a function with x as the argument of that function -> it doesn't change the global value of x in the JN -> this is scope (scope of the variable name in the function)**
        - **-> you can declare in the definition of a function -> global x = 100 (then if you change it later in the function to a variable local to the function, the value of x used in the definition of the function is a local one - and everywhere else it's the global value of x (which was assigned this value in the definition of the function in this example)**
        - -> when you use the scripts again and again if the function definitions involve global assignments to variables ->v this can cause errors if the function is used again and again

- **28/29 Methods and Functions Homework Overview**
    - -> testing skills with methods and functions
    - JN
        - the first function one which calculates the volume of the sphere given its radius
        - -> the next cell is **one which tests that the function works after it's defined**
        - -> the next question
            - define a function to check if a number is in a given range which returns a boolean or a statement
        - -> the next
            - a function which inputs a string and returns the number of upper and lowercase letters in that input
            - .isupper() and .islower()
        - -> the next
            - -> this takes a list and removes the repeats
            - -> sets are for the unique elements in a list (but we want an array returned)
        - -> the next question
            - -> one which inputs an array and timses all of the numbers in the array together
        - -> next
            - -> write a Python function which checks if the input string is the same thing spelt backwards <- iterate through the entire thing backwards
            - -> you can use the .replace() method
        - -> next
            - -> a Python function to check weather a string is a pangram or not
            - -> they are making you search this (for research skills) on stack overflow
            - -> a word or sentence with every letter of the alphabet at least once
                - string.ascli_lowercase
            - -> assume there is no punctuation passed into the string
                - **we are making assumptions about the inputs all being in a certain form**

- **29/29 Methods and Functions Homework - Solutions**
    - in JN
        - put an equation into code e.g -> you can import pi from import math, from math import pi
        - -> **you can put the equation and return on the same line**

- -> in another example
    - for item in range(0,5)L
        - print(item)
- **-> 5 in range (1,10) <- this is asking if 5 is in that range of numbers**
    - -> so to put this into a function, it's
    - def ran_check(num, low, high):
        - num in range(low, high+1) <- you can also put this entire thing into an if / else block. **She's printed the result in an f string literal**
- -> a Python function which returns the number of upper and lowercase elements in the string
    - -> we need a loop in there
    - -> she's initialised two counters -> one for the number of upper case strings in the input string and a similar one for the number of lowercase strings in it
    - -> we're iterating through the characters in the input string
        - if it's .upper() <- then we're increasing the uppercase counter by 1, it is itself +=1
    - -> then she's printing out the results in an f string literal
        - **and testing the function**
    - **-> you can also set up all of the initialised variables into a dictionary (e.g d={'upper':0,'lower':0}**
- -> a function which takes a string and removes the repeated elements in it
    - def unique)list(list):
        - return list(set(list)) <- **sets remove the repeated elements**
    - another way of doing this is
        - x = []
        - for number in list:
            - if number not in seen_numbers:
                - seen_numbers.append(number)
        - return seen_numbers <- if seen numbers isn't in the list then it hasn't been repeated
- -> another example
    - def multiply(numbers):
        - total = 1
        - for num in numbers:
            - total = total * num
        - return total <- we're taking an array of numbers, iterating through them and then retuning the total product
        - -> **then she's testing the function for different use cases again once it's been written**
- -> a function to check if the string is itself written backwards (palindrome)
    - checking is the string == itself spelt backwards
    - she's done s = s.replace(' ','')
        - -> the first stage is to remove the spaces
        - **-> s == s[::-1] <- we are checking if s is itself backwards (step size going back wards, stepping -1)**
        - -> then return s == s[::-1]
- -> another one is a Python function to check is a string is a pangram or not
    - a word or sentence containing every letter of the alphabet at least once
    - -> it should return for 'the quick brown fox jumps over the lazy dog'
    - -> she creates a set of the entire alphabet
    - -> removes spaces from the input string
    - -> converts the input string into all lowercase strings

- -> **a lot of it is first making sure the input is in the right format**
- -> making that into a set -> so we have no repeats
- -> the set of the alphabet
- -> checking is the alphabet set is equal to the input
- -> remove any spaces from the input string
- -> converting everything into lowercase
- -> **she's converted both of them into sets, and then is comparing them)**
- -> another way to visualise it is through print statements
- -> debugging / understanding what is going on in larger functions