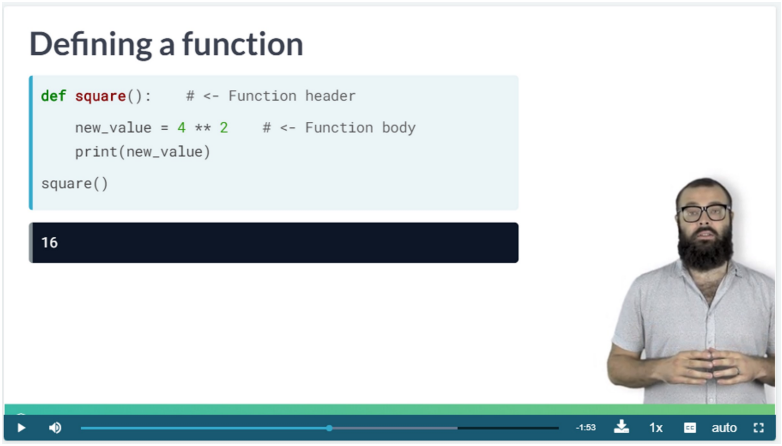
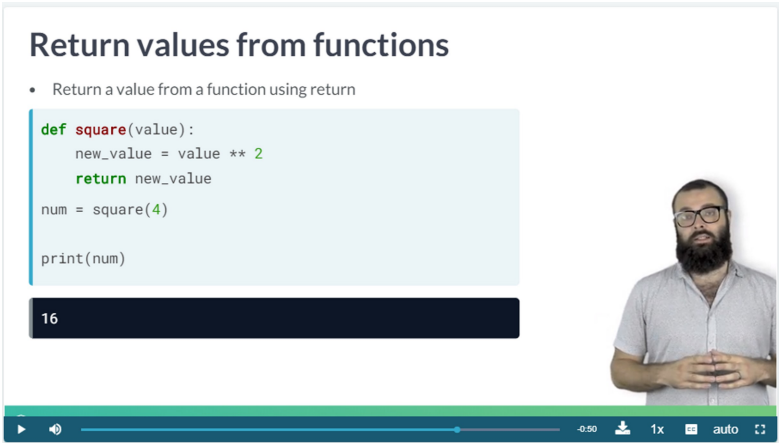
03. Python Data Science Toolbox (Part 1)

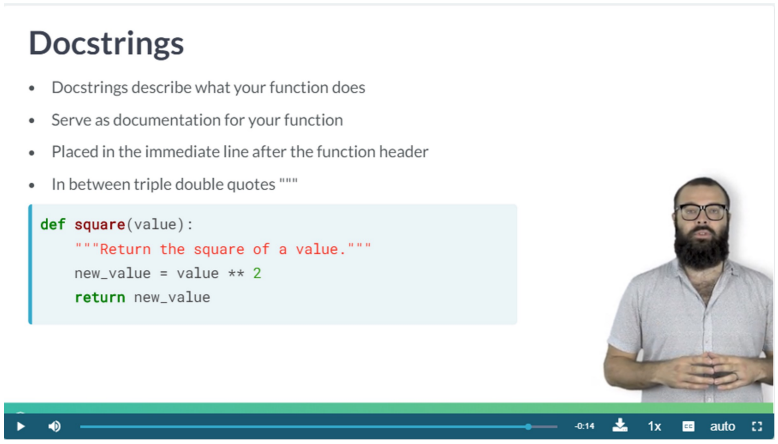
02 January 2020 15:42

1. Writing your own functions

1.1 Strings in Python







In the video, you learned of another standard Python datatype, **strings**. Recall that these represent textual data. To assign the string 'DataCamp' to a variable company, you execute:

```
company = 'DataCamp'
```

You've also learned to use the operations + and * with strings. Unlike with numeric types such as ints and floats, the + operator *concatenates* strings together, while the * concatenates multiple copies of a string together. In this exercise, you will

use the + and * operations on strings to answer the question below. Execute the following code in the shell:

```
object1 = "data" + "analysis" + "visualization"
object2 = 1 * 3
object3 = "1" * 3
In [1]: "1" * 3
Out[1]: '111'
```

What are the values in object1, object2, and object3, respectively?

Possible Answers

- [ ] object1 contains "data + analysis + visualization", object2 contains "1*3", object3 contains 13.
- [ ] object1 contains "data+analysis+visualization", object2 contains 3, object3 contains "13".
- [x] object1 contains "dataanalysisvisualization", object2 contains 3, object3 contains "111".

1.2 Recapping built-in functions

In the video, Hugo briefly examined the return behavior of the built-in functions print() and str(). Here, you will use both functions and examine their return values. A variable x has been preloaded for this exercise. Run the code below in the console. Pay close attention to the results to answer the question that follows.

- Assign str(x) to a variable y1: y1 = str(x)
- Assign print(x) to a variable y2: y2 = print(x)
- Check the types of the variables x, y1, and y2.

What are the types of x, y1, and y2?

Possible Answers

- [ ] They are all str types.
- [ ] x is a float, y1 is an float, and y2 is a str.
- [x] x is a float, y1 is a str, and y2 is a NoneType.
- [ ] They are all NoneType types.

1.3 Write a simple function

In the last video, Hugo described the basics of how to define a function. You will now write your own function!

Define a function, shout(), which simply prints out a string with three exclamation marks '!!!' at the end. The code for the square() function that we wrote earlier is found below. You can use it as a pattern to define shout().

```
def square():
    new_value = 4 ** 2
    return new_value
```

Note that the function body is indented 4 spaces already for you. Function bodies need to be indented by a consistent number of spaces and the choice of 4 is common.

*This course touches on a lot of concepts you may have forgotten, so if you ever need a quick refresher, download the **Python for data science Cheat Sheet** and keep it handy!*

Instructions:

- Complete the function header by adding the appropriate function name, shout.
- In the function body, **concatenate** the string, 'congratulations' with another string, '!!!'. Assign the result to shout_word.
- Print the value of shout_word.
- Call the shout function.

```
# Define the function shout
def shout():
    """Print a string with three exclamation marks"""
    # Concatenate the strings: shout_word
    shout_word = "congratulations" + "!!!"
```

```
    # Print shout_word
    print(shout_word)

# Call shout
shout()
```

1.4 Single-parameter functions

Congratulations! You have successfully defined *and* called your own function! That's pretty cool.

In the previous exercise, you defined and called the function shout(), which printed out a string concatenated with '!!!'. You will now update shout() by adding a *parameter* so that it can accept and process any string *argument* passed to it. Also note that shout(word), the part of the *header* that specifies the function name and parameter(s), is known as the *signature* of the function. You may encounter this term in the wild!

Instructions:

- Complete the function header by adding the parameter name, word.
- Assign the result of concatenating word with '!!!' to shout_word.
- Print the value of shout_word.
- Call the shout() function, passing to it the string, 'congratulations'.

```
def shout(word):
    """Print a string with three exclamation marks"""
    # Concatenate the strings: shout_word
    shout_word = word + '!!!'

    # Print shout_word
    print(shout_word)

# Call shout with the string 'congratulations'
shout("congratulations")
```

1.5 Functions that return single values

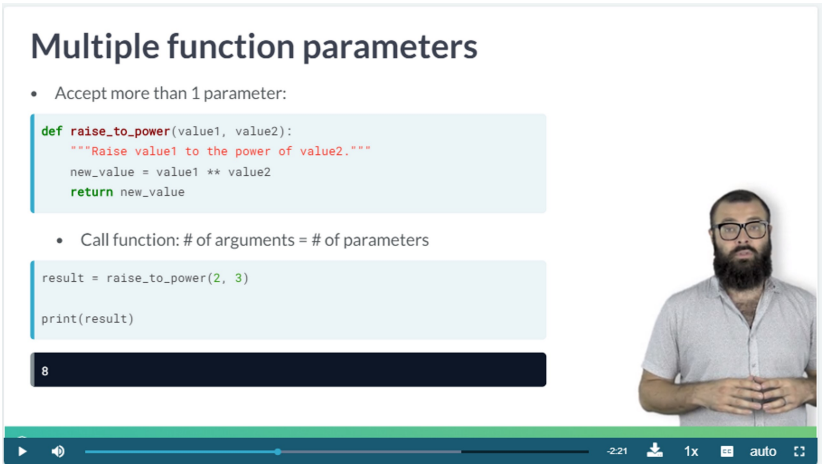
You're getting very good at this! Try your hand at another modification to the shout() function so that it now *returns* a single value instead of printing within the function. Recall that the return keyword lets you return values from functions. Parts of the function shout(), which you wrote earlier, are shown. Returning values is generally more desirable than printing them out because, as you saw earlier, a print() call assigned to a variable has type NoneType.

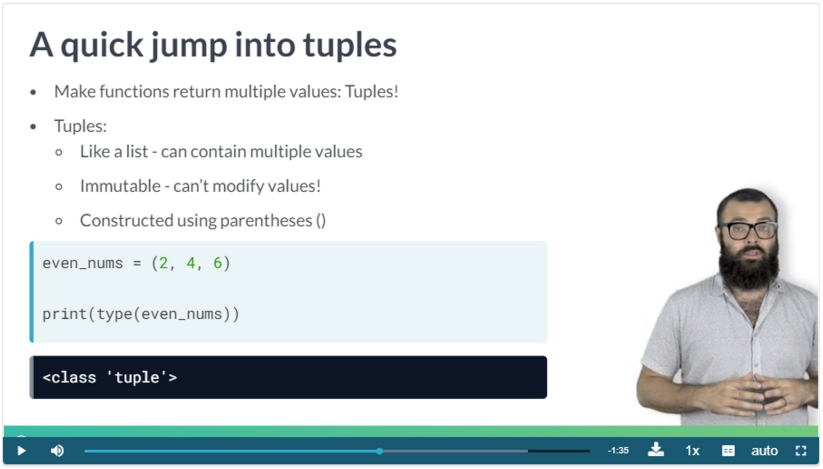
Instructions:

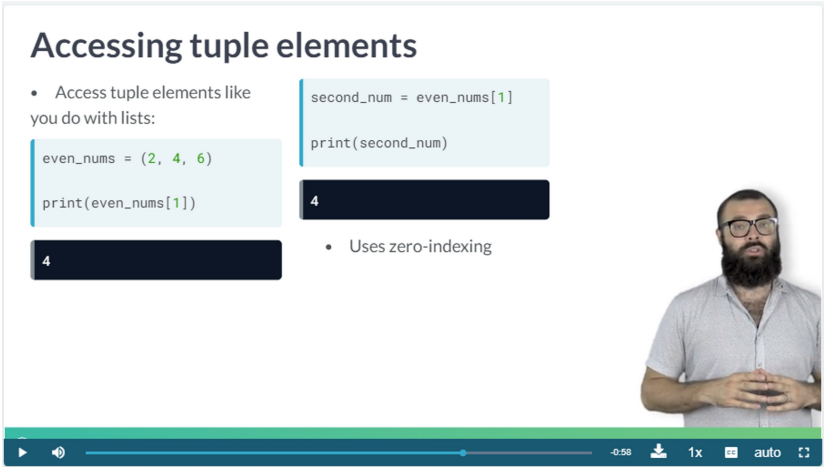
- In the function body, concatenate the string in word with '!!!' and assign to shout_word.
- Replace the print() statement with the appropriate return statement.
- Call the shout() function, passing to it the string, 'congratulations', and assigning the call to the variable, yell.
- To check if yell contains the value returned by shout(), print the value of yell.

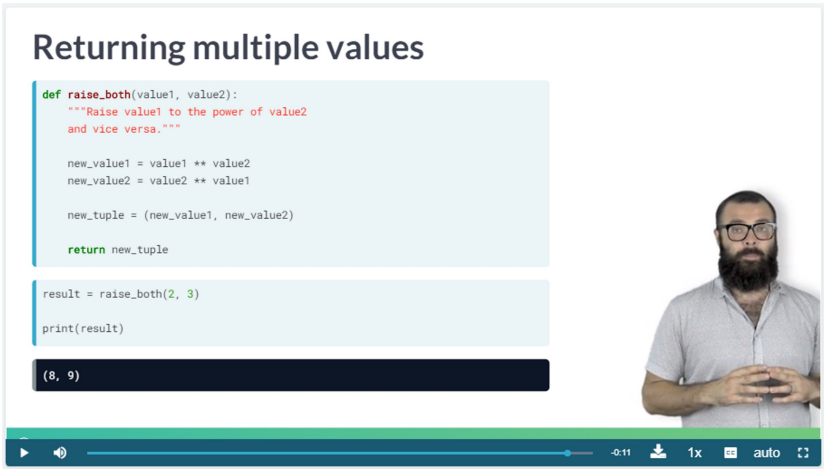
```
# Define shout with the parameter, word
def shout(word):
    """Return a string with three exclamation marks"""
    # Concatenate the strings: shout_word
    shout_word = word + "!!!"

    # Replace print with return
    return(shout_word)

# Pass 'congratulations' to shout: yell
yell = shout("congratulations")

# Print yell
print(yell)
```

1.6 Functions with multiple parameters









Hugo discussed the use of multiple parameters in defining functions in the last lecture. You are now going to use what you've learned to modify the shout() function further.
Here, you will modify shout() to accept two arguments. Parts of the function shout(), which you wrote earlier, are shown.

Instructions:

- Modify the function header such that it accepts two parameters, word1 and word2, in that order.
- Concatenate each of word1 and word2 with '!!!' and assign to shout1 and shout2, respectively.
- Concatenate shout1 and shout2 together, in that order, and assign to new_shout.
- Pass the strings 'congratulations' and 'you', in that order, to a call to shout(). Assign the return value to yell.

```
# Define shout with parameters word1 and word2
def shout(word1, word2):
    """Concatenate strings with three exclamation marks"""
    # Concatenate word1 with '!!!': shout1
    shout1 = word1 + "!!!"

    # Concatenate word2 with '!!!': shout2
    shout2 = word2 + "!!!"

    # Concatenate shout1 with shout2: new_shout
    new_shout = shout1 + shout2

    # Return new_shout
    return new_shout

# Pass 'congratulations' and 'you' to shout(): yell
yell = shout("congratulations", "you")

# Print yell
print(yell)
```

1.7 A brief introduction to tuples

Alongside learning about functions, you've also learned about tuples! Here, you will practice what you've learned about tuples: how to construct, unpack, and access tuple elements. Recall how Hugo unpacked the tuple even_nums in the video:

```
a, b, c = even_nums
```

A three-element tuple named nums has been preloaded for this exercise. Before completing the script, perform the following:

- Print out the value of nums in the IPython shell. Note the elements in the tuple.
- In the IPython shell, try to change the first element of nums to the value *2* by doing an assignment: nums[0] = 2. What happens?

```
nums[0]=2
TypeError: 'tuple' object does not support item assignment
```

Instructions:

- Unpack nums to the variables num1, num2, and num3.
- Construct a new tuple, even_nums composed of the same elements in nums, but with the 1st element replaced with the value, *2*.

```
# Unpack nums into num1, num2, and num3
num1, num2, num3 = nums

# Construct even_nums
even_nums = (2, num2, num3)
```

1.8 Functions that return multiple values

In the previous exercise, you constructed tuples, assigned tuples to variables, and unpacked tuples. Here you will return multiple values from a function using tuples. Let's now update our shout() function to return multiple values. Instead of returning just one string, we will return two strings with the string !!! concatenated to each.

Note that the return statement return x, y has the same result as return (x, y): the former actually packs x and y into a tuple under the hood!

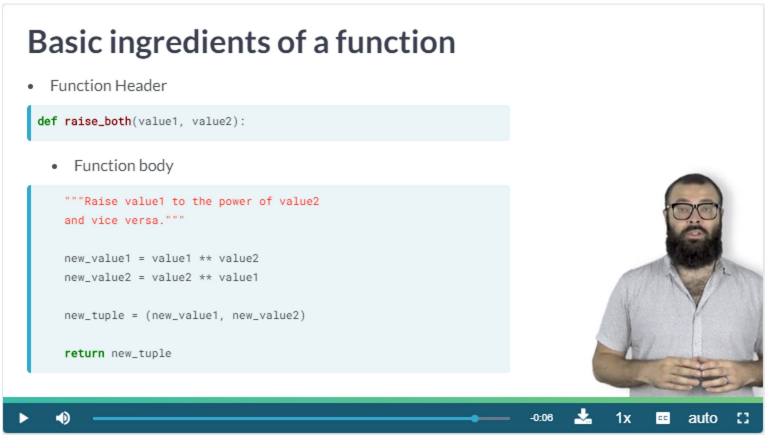
Instructions:

- Modify the function header such that the function name is now shout_all, and it accepts two parameters, word1 and word2, in that order.
- Concatenate the string '!!!' to each of word1 and word2 and assign to shout1 and shout2, respectively.
- Construct a tuple shout_words, composed of shout1 and shout2.
- Call shout_all() with the strings 'congratulations' and 'you' and assign the result to yell1 and yell2 (remember, shout_all() returns 2 variables!).

```
# Define shout_all with parameters word1 and word2
def shout_all(word1, word2):

    # Concatenate word1 with '!!!': shout1
    shout1 = word1 + "!!!"

    # Concatenate word2 with '!!!': shout2
    shout2 = word2 + "!!!"

    # Construct a tuple with shout1 and shout2: shout_words
    shout_words = (shout1, shout2)

    # Return shout_words
    return shout_words

# Pass 'congratulations' and 'you' to shout_all(): yell1, yell2
yell1, yell2 = shout_all("congratulations", "you")
```

```
# Print yell1 and yell2
print(yell1)
print(yell2)
```

1.9 Bringing it all together (1)



You've got your first taste of writing your own functions in the previous exercises. You've learned how to add parameters to your own function definitions, return a value or multiple values with tuples, and how to call the functions you've defined.

In this and the following exercise, you will bring together all these concepts and apply them to a simple data science problem. You will load a dataset and develop functionalities to extract simple insights from the data.

For this exercise, your goal is to recall how to load a dataset into a DataFrame. The dataset contains Twitter data and you will iterate over entries in a column to build a dictionary in which the keys are the names of languages and the values are the number of tweets in the given language. The file tweets.csv is available in your current directory.

Be aware that this is real data from Twitter and as such there is always a risk that it may contain profanity or other offensive content (in this exercise, and any following exercises that also use real Twitter data).

Instructions:

- Import the pandas package with the alias pd.
- Import the file 'tweets.csv' using the pandas function read_csv(). Assign the resulting DataFrame to df.
- Complete the for loop by iterating over col, the 'lang' column in the DataFrame df.
- Complete the bodies of the if-else statements in the for loop: **if** the key is in the dictionary langs_count, add 1 to the value corresponding to this key in the dictionary, **else** add the key to langs_count and set the corresponding value to 1. Use the loop variable entry in your code.

```
# Import pandas
import pandas as pd

# Import Twitter data as DataFrame: df
df = pd.read_csv("tweets.csv")

# Initialize an empty dictionary: langs_count
langs_count = {}

# Extract column from DataFrame: col
col = df['lang']

# Iterate over lang column in DataFrame
for entry in col:

    # If the language is in langs_count, add 1
    if entry in langs_count.keys():
        langs_count[entry] += 1
    # Else add the language to langs_count, set the value to 1
    else:
```

```
        langs_count[entry] = 1

# Print the populated dictionary
print(langs_count)
```

1.10 Bringing it all together (2)

Great job! You've now defined the functionality for iterating over entries in a column and building a dictionary with keys the names of languages and values the number of tweets in the given language.

In this exercise, you will define a function with the functionality you developed in the previous exercise, return the resulting dictionary from within the function, and call the function with the appropriate arguments.

For your convenience, the pandas package has been imported as `pd` and the `'tweets.csv'` file has been imported into the `tweets_df` variable.

Instructions:

- Define the function `count_entries()`, which has two parameters. The first parameter is `df` for the DataFrame and the second is `col_name` for the column name.
- Complete the bodies of the `if-else` statements in the `for` loop: **if** the key is in the dictionary `langs_count`, add `1` to its current value, **else** add the key to `langs_count` and set its value to `1`. Use the loop variable `entry` in your code.
- Return the `langs_count` dictionary from inside the `count_entries()` function.
- Call the `count_entries()` function by passing to it `tweets_df` and the name of the column, `'lang'`. Assign the result of the call to the variable `result`.

```
# Define count_entries()
def count_entries(df, col_name):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: langs_count
    langs_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over lang column in DataFrame
    for entry in col:

        # If the language is in langs_count, add 1
        if entry in langs_count.keys():
            langs_count[entry] += 1
        # Else add the language to langs_count, set the value to 1
        else:
            langs_count[entry] = 1

    # Return the langs_count dictionary
    return(langs_count)

# Call count_entries(): result
result = count_entries(tweets_df, 'lang')

# Print the result
print(result)
```

2. Default arguments, variable-length arguments and scope

2.1 Pop quiz on understanding scope

Crash course on scope in functions

- Not all objects are accessible everywhere in a script
- Scope - part of the program where an object or name may be accessible
  - Global scope - defined in the main body of a script
  - Local scope - defined inside a function
  - Built-in scope - names in the pre-defined built-ins module

-2:25 1x auto

Global vs. local scope (1)

```
def square(value):
    """Returns the square of a number."""
    new_val = value ** 2
    return new_val
square(3)
```

```
9
```

```
new_val
```

```
<hr />---------------------------------------------------------------
NameError                 Traceback (most recent call last)
<ipython-input-3-3cc6c6de5c5c> in <module>()
<hr />--> 1 new_value
NameError: name 'new_val' is not defined
```

-1:55 1x auto

Global vs. local scope (2)

```
new_val = 10

def square(value):
    """Returns the square of a number."""
    new_val = value ** 2
    return new_val
square(3)
```

```
9
```

```
new_val
```

```
10
```

-1:17 1x auto

In this exercise, you will practice what you've learned about scope in functions. The variable num has been predefined as 5, alongside the following function definitions:

```
def func1():
  num = 3
  print(num)
def func2():
  global num
  double_num = num * 2
  num = 6
  print(double_num)
```

Try calling func1() and func2() in the shell, then answer the following questions:

- What are the values printed out when you call func1() and func2()?
- What is the value of num in the global scope after calling func1() and func2()?

Possible Answers

- [ ] func1() prints out 3, func2() prints out 6, and the value of num in the global scope is 3.
- [ ] func1() prints out 3, func2() prints out 3, and the value of num in the global scope is 3.
- [ ] func1() prints out 3, func2() prints out 10, and the value of num in the global scope is 10.
- [x] func1() prints out 3, func2() prints out 10, and the value of num in the global scope is 6.

2.2 The keyword global

Let's work more on your mastery of scope. In this exercise, you will use the keyword global within a function to alter the value of a variable defined in the global scope.

Instructions:

- Use the keyword global to alter the object team in the global scope.

- Change the value of team in the global scope to the string "justice league". Assign the result to team.
- Hit the Submit button to see how executing your newly defined function change_team() changes the value of the name team!

```
# Create a string: team
team = "teen titans"

# Define change_team()
def change_team():
  """Change the value of the global variable team."""

  # Use team in global scope
  global team

  # Change the value of team in global: team
  team = "justice league"
# Print team
print(team)

# Call change_team()
change_team()

# Print team
print(team)
```

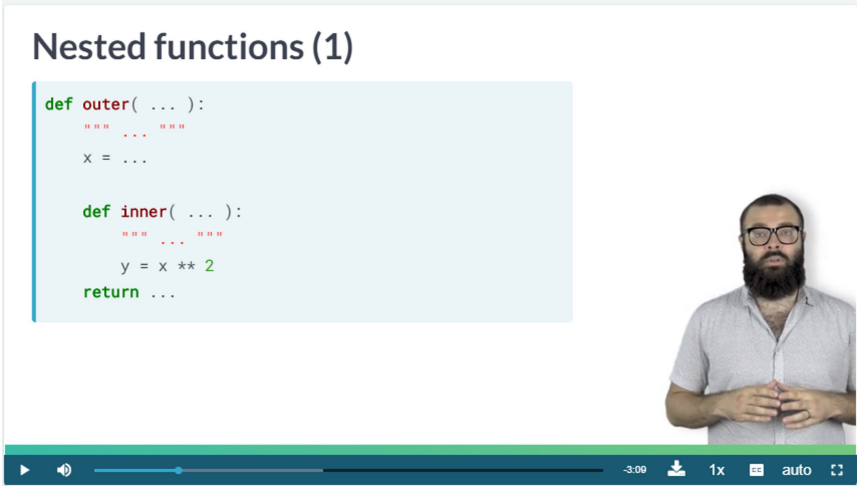
2.3 Python's built-in scope

Here you're going to check out Python's built-in scope, which is really just a built-in module called builtins. However, to query builtins, you'll need to import builtins 'because the name builtins is not itself built in...No, I'm serious!' (***Learning Python*, 5th edition**, Mark Lutz). After executing import builtins in the IPython Shell, execute dir(builtins) to print a list of all the names in the module builtins. Have a look and you'll see a bunch of names that you'll recognize! Which of the following names is NOT in the module builtins?

Possible Answers

- [ ] 'sum'
- [ ] 'range'
- [x] 'array'
- [ ] 'tuple'

2.4 Nested Functions I



Nested functions (3)

```
def mod2plus5(x1, x2, x3):
    """Returns the remainder plus 5 of three values."""

    def inner(x):
        """Returns the remainder plus 5 of a value."""
        return x % 2 + 5

    return (inner(x1), inner(x2), inner(x3))
```

```
print(mod2plus5(1, 2, 3))
```

```
(6, 5, 6)
```

-2:31 1x auto

Returning functions

```
def raise_val(n):
    """Return the inner function."""

    def inner(x):
    """Raise x to the power of n."""
        raised = x ** n
        return raised

    return inner
```

```
square = raise_val(2)
cube = raise_val(3)
print(square(2), cube(4))
```

```
4 64
```

-1:21 1x auto

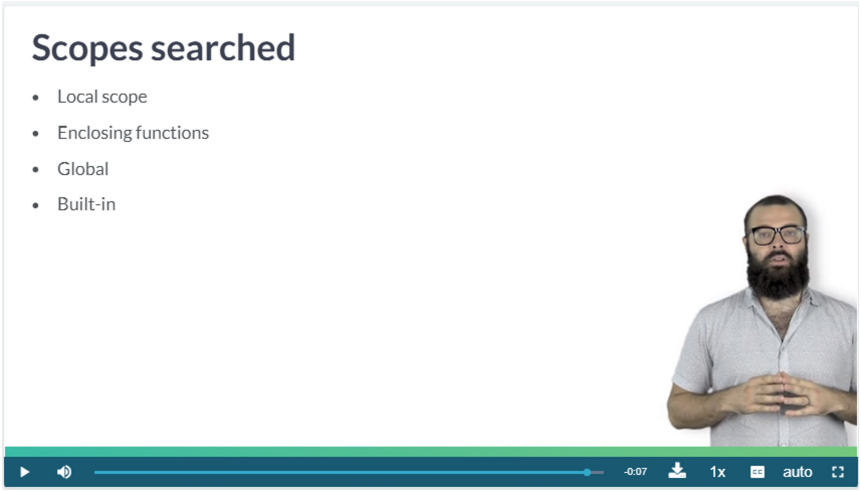
Using nonlocal

```
def outer():
    """Prints the value of n."""
    n = 1

    def inner():
        nonlocal n
        n = 2
        print(n)

    inner()
    print(n)
```

```
outer()
```

```
2
2
```

-0:47 1x auto



You've learned in the last video about nesting functions within functions. One reason why you'd like to do this is to avoid writing out the same computations within functions repeatedly. There's nothing new about defining nested functions: you simply define it as you would a regular function with def and embed it inside another function!

In this exercise, inside a function three_shouts(), you will define a nested function inner() that concatenates a string object with !!!. three_shouts() then returns a tuple of three elements, each a string concatenated with !!! using inner(). Go for it!

Instructions:

- Complete the function header of the nested function with the function name inner() and a single parameter word.
- Complete the return value: each element of the tuple should be a call to inner(), passing in the parameters from three_shouts() as arguments to each call.

```
# Define three_shouts
def three_shouts(word1, word2, word3):
    """Returns a tuple of strings
    concatenated with '!!!'."""

    # Define inner
    def inner(word):
        """Returns a string concatenated with '!!!'."""
        return word + '!!!'

    # Return a tuple of strings
    return (inner(word1), inner(word2), inner(word3))

# Call three_shouts() and print
print(three_shouts('a', 'b', 'c'))
```

2.5 Nested Functions II

Great job, you've just nested a function within another function. One other pretty cool reason for nesting functions is the idea of a **closure**. This means that the nested or inner function remembers the state of its enclosing scope when called. Thus, anything defined locally in the enclosing scope is available to the inner function even when the outer function has finished execution.

Let's move forward then! In this exercise, you will complete the definition of the inner function inner_echo() and then call echo() a couple of times, each with a different argument. Complete the exercise and see what the output will be!

Instructions:

- Complete the function header of the inner function with the function name inner_echo() and a single parameter word1.
- Complete the function echo() so that it returns inner_echo.
- We have called echo(), passing 2 as an argument, and assigned the resulting function to twice. Your job is to call echo(), passing 3 as an argument. Assign the resulting function to thrice.
- Hit Submit to call twice() and thrice() and print the results.

```
# Define echo
```

```
def echo(n):
    """Return the inner_echo function."""

    # Define inner_echo
    def inner_echo(word1):
        """Concatenate n copies of word1."""
        echo_word = word1 * n
        return echo_word

    # Return inner_echo
    return(inner_echo)

# Call echo: twice
twice = echo(2)

# Call echo: thrice
thrice = echo(3)

# Call twice() and thrice() then print
print(twice('hello'), thrice('hello'))
```

```
In [3]: echo(2)("hello")
Out[3]: 'hellohello'

In [4]: echo(4)("hello")
Out[4]: 'hellohellohellohello'
```

2.6 The keyword nonlocal and nested functions

Let's once again work further on your mastery of scope! In this exercise, you will use the keyword `nonlocal` within a nested function to alter the value of a variable defined in the enclosing scope.

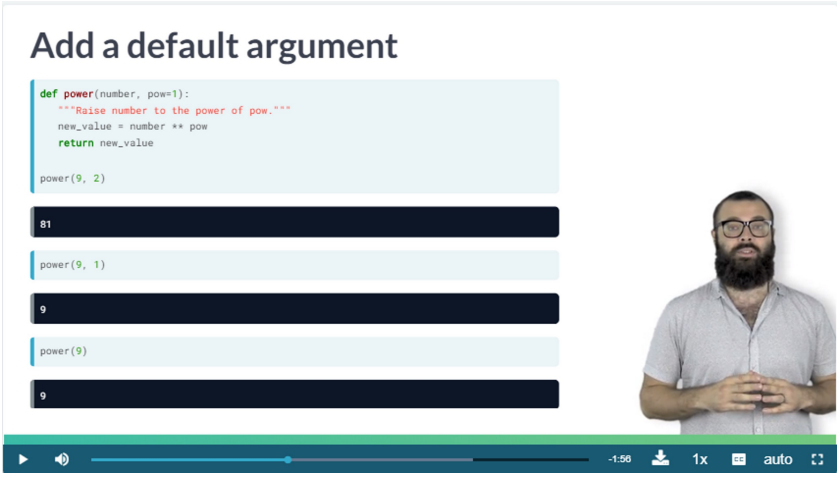
Instructions:

- Assign to `echo_word` the string `word`, concatenated with itself.
- Use the keyword `nonlocal` to alter the value of `echo_word` in the enclosing scope.
- Alter `echo_word` to `echo_word` concatenated with '!!!'.
- Call the function `echo_shout()`, passing it a single argument 'hello'.

```
# Define echo_shout()
def echo_shout(word):
    """Change the value of a nonlocal variable"""

    # Concatenate word with itself: echo_word
    echo_word = word + word

    # Print echo_word
    print(echo_word)

    # Define inner function shout()
    def shout():
        """Alter a variable in the enclosing scope"""
        # Use echo_word in nonlocal scope
        nonlocal echo_word

        # Change echo_word to echo_word concatenated with '!!!'
        echo_word = echo_word + "!!!"

    # Call function shout()
    shout()

    # Print echo_word
    print(echo_word)
```

```
# Call function echo_shout() with argument 'hello'
echo_shout("hello")
```

2.7 Functions with one default argument



In the previous chapter, you've learned to define functions with more than one parameter and then calling those functions by passing the required number of arguments. In the last video, Hugo built on this idea by showing you how to define functions with default arguments. You will practice that skill in this exercise by writing a function that uses a default argument and then calling the function a couple of times.

Instructions:

- Complete the function header with the function name `shout_echo`. It accepts an argument `word1` and a default argument `echo` with default value `1`, in that order.
- Use the `*` operator to concatenate `echo` copies of `word1`. Assign the result to `echo_word`.
- Call `shout_echo()` with just the string, `"Hey"`. Assign the result to `no_echo`.
- Call `shout_echo()` with the string `"Hey"` and the value `5` for the default argument, `echo`. Assign the result to `with_echo`.

```
# Define shout_echo
def shout_echo(word1, echo = 1):
    """Concatenate echo copies of word1 and three
     exclamation marks at the end of the string."""

    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Concatenate '!!!' to echo_word: shout_word
    shout_word = echo_word + '!!!'

    # Return shout_word
    return shout_word

# Call shout_echo() with "Hey": no_echo
no_echo = shout_echo("Hey")

# Call shout_echo() with "Hey" and echo=5: with_echo
with_echo = shout_echo("Hey", echo = 5)

# Print no_echo and with_echo
print(no_echo)
print(with_echo)
```

2.8 Functions with multiple default arguments

You've now defined a function that uses a default argument - don't stop there just yet! You will now try your hand at defining a function with more than one default argument and then calling this function in various ways.

After defining the function, you will call it by supplying values to *all* the default arguments of the function. Additionally, you will call the function by not passing a value to one of the default arguments - see how that changes the output of your function!

Instructions:

- Complete the function header with the function name `shout_echo`. It accepts an argument `word1`, a default argument `echo` with

default value 1 and a default argument intense with default value False, in that order.
- In the body of the if statement, make the string object echo_word upper case by applying the method .upper() on it.
- Call shout_echo() with the string, "Hey", the value 5 for echo and the value True for intense. Assign the result to with_big_echo.
- Call shout_echo() with the string "Hey" and the value True for intense. Assign the result to big_no_echo.

```
# Define shout_echo
def shout_echo(word1, echo = 1, intense = False):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""

    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Make echo_word uppercase if intense is True
    if intense is True:
        # Make uppercase and concatenate '!!!': echo_word_new
        echo_word_new = echo_word.upper() + '!!!'
    else:
        # Concatenate '!!!' to echo_word: echo_word_new
        echo_word_new = echo_word + '!!!'

    # Return echo_word_new
    return echo_word_new

# Call shout_echo() with "Hey", echo=5 and intense=True: with_big_echo
with_big_echo = shout_echo(word1 = "Hey", echo = 5, intense = True)

# Call shout_echo() with "Hey" and intense=True: big_no_echo
big_no_echo = shout_echo(word1 = "Hey", intense = True)

# Print values
print(with_big_echo)
print(big_no_echo)
```

*2.9 Functions with variable-length arguments (*args)*

Flexible arguments enable you to pass a variable number of arguments to a function. In this exercise, you will practice defining a function that accepts a variable number of string arguments.

The function you will define is gibberish() which can accept a variable number of string values. Its return value is a single string composed of all the string arguments concatenated together in the order they were passed to the function call. You will call the function with a single string argument and see how the output changes with another call using more than one string argument. Recall from the previous video that, within the function definition, args is a tuple.

Instructions:

- Complete the function header with the function name gibberish. It accepts a single flexible argument *args.
- Initialize a variable hodgepodge to an empty string.
- Return the variable hodgepodge at the end of the function body.
- Call gibberish() with the single string, "luke". Assign the result to one_word.
- Hit the Submit button to call gibberish() with multiple arguments and to print the value to the Shell.

```
# Define gibberish
def gibberish(*args):
    """Concatenate strings in *args together."""

    # Initialize an empty string: hodgepodge
    hodgepodge = ""

    # Concatenate the strings in args
    for word in args:
        hodgepodge += word

    # Return hodgepodge
```

```
    return(hodgepodge)

# Call gibberish() with one string: one_word
one_word = gibberish("luke")

# Call gibberish() with five strings: many_words
many_words = gibberish("luke", "leia", "han", "obi", "darth")

# Print one_word and many_words
print(one_word)
print(many_words)
```

*2.10 Functions with variable-length keyword arguments (**kwargs)*

Let's push further on what you've learned about flexible arguments - you've used *args, you're now going to use **kwargs! What makes **kwargs different is that it allows you to pass a variable number of *keyword arguments* to functions. Recall from the previous video that, within the function definition, kwargs is a dictionary.

To understand this idea better, you're going to use **kwargs in this exercise to define a function that accepts a variable number of keyword arguments. The function simulates a simple status report system that prints out the status of a character in a movie.

Instructions:

- Complete the function header with the function name report_status. It accepts a single flexible argument **kwargs.
- Iterate over the key-value pairs of kwargs to print out the keys and values, separated by a colon ':'.
- In the first call to report_status(), pass the following keyword-value pairs: name="luke", affiliation="jedi" and status="missing".
- In the second call to report_status(), pass the following keyword-value pairs: name="anakin", affiliation="sith lord" and status="deceased".

```
# Define report_status
def report_status(**kwargs):
    """Print out the status of a movie character."""

    print("\nBEGIN: REPORT\n")

    # Iterate over the key-value pairs of kwargs
    for key, value in kwargs.items():
        # Print out the keys and values, separated by a colon ':'
        print(key + ": " + value)

    print("\nEND REPORT")

# First call to report_status()
report_status(name = "luke", affiliation = "jedi", status = "missing")

# Second call to report_status()
report_status(name = "anakin", affiliation = "sith lord", status = "deceased")
```

2.11 Bringing it all together (1)

Recall the *Bringing it all together* exercise in the previous chapter where you did a simple Twitter analysis by developing a function that counts how many tweets are in certain languages. The output of your function was a dictionary that had the language as the *keys* and the counts of tweets in that language as the *value*.

In this exercise, we will generalize the Twitter language analysis that you did in the previous chapter. You will do that by including a **default argument** that takes a column name.

For your convenience, pandas has been imported as pd and the 'tweets.csv' file has been imported into the DataFrame tweets_df. Parts of the code from your previous work are also provided.

Instructions:

- Complete the function header by supplying the parameter for a DataFrame df and the parameter col_name with a default value

of 'lang' for the DataFrame column name.
- Call count_entries() by passing the tweets_df DataFrame and the column name 'lang'. Assign the result to result1. Note that since 'lang' is the default value of the col_name parameter, you don't have to specify it here.
- Call count_entries() by passing the tweets_df DataFrame and the column name 'source'. Assign the result to result2.

```
# Define count_entries()
def count_entries(df, col_name = "lang"):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over the column in DataFrame
    for entry in col:

        # If entry is in cols_count, add 1
        if entry in cols_count.keys():
            cols_count[entry] += 1

        # Else add the entry to cols_count, set the value to 1
        else:
            cols_count[entry] = 1

    # Return the cols_count dictionary
    return cols_count

# Call count_entries(): result1
result1 = count_entries(df = tweets_df)

# Call count_entries(): result2
result2 = count_entries(df = tweets_df, col_name = "source")

# Print result1 and result2
print(result1)
print(result2)
```

2.12 Bringing it all together (2)

Wow, you've just generalized your Twitter language analysis that you did in the previous chapter to include a default argument for the column name. You're now going to generalize this function one step further by allowing the user to pass it a flexible argument, that is, in this case, as many column names as the user would like!

Once again, for your convenience, pandas has been imported as pd and the 'tweets.csv' file has been imported into the DataFrame tweets_df. Parts of the code from your previous work are also provided.

Instructions:

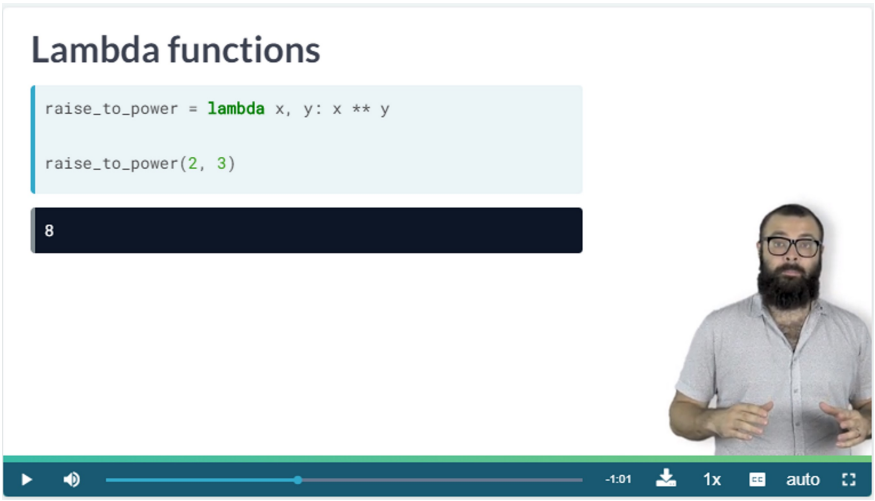
- Complete the function header by supplying the parameter for the dataframe df and the flexible argument *args.
- Complete the for loop within the function definition so that the loop occurs over the tuple args.
- Call count_entries() by passing the tweets_df DataFrame and the column name 'lang'. Assign the result to result1.
- Call count_entries() by passing the tweets_df DataFrame and the column names 'lang' and 'source'. Assign the result to result2.

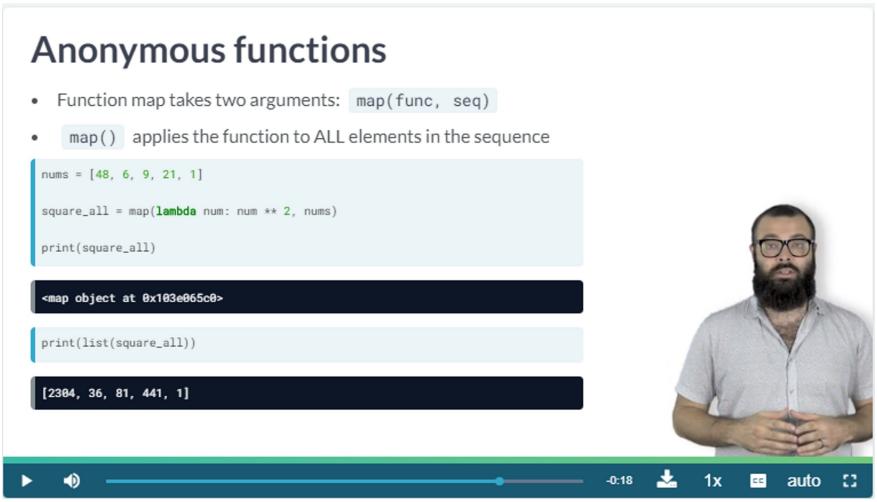
```
# Define count_entries()
def count_entries(df, *args):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    #Initialize an empty dictionary: cols_count
    cols_count = {}
```

```
    # Iterate over column names in args
    for col_name in args:

        # Extract column from DataFrame: col
        col = df[col_name]

        # Iterate over the column in DataFrame
        for entry in col:

            # If entry is in cols_count, add 1
            if entry in cols_count.keys():
                cols_count[entry] += 1

            # Else add the entry to cols_count, set the value to 1
            else:
                cols_count[entry] = 1

    # Return the cols_count dictionary
    return cols_count

# Call count_entries(): result1
result1 = count_entries(tweets_df, "lang")

# Call count_entries(): result2
result2 = count_entries(tweets_df, "lang", "source")

# Print result1 and result2
print(result1)
print(result2)
```

3. Lambda functions and error-handling

3.1 Pop quiz on lambda functions





In this exercise, you will practice writing a simple lambda function and calling this function. Recall what you know about lambda functions and answer the following questions:

- How would you write a lambda function add_bangs that adds three exclamation points '!!!' to the end of a string a?
- How would you call add_bangs with the argument 'hello'?

You may use the IPython shell to test your code.

Possible Answers

- [ ] The lambda function definition is: add_bangs = (a + '!!!'), and the function call is: add_bangs('hello').
- [x] The lambda function definition is: add_bangs = (lambda a: a + '!!!'), and the function call is: add_bangs('hello').
- [ ] The lambda function definition is: (lambda a: a + '!!!') = add_bangs, and the function call is: add_bangs('hello').

3.2 Writing a lambda function you already know

Some function definitions are simple enough that they can be converted to a lambda function. By doing this, you write less lines of code, which is pretty awesome and will come in handy, especially when you're writing and maintaining big programs. In this exercise, you will use what you know about lambda functions to convert a function that does a simple task into a lambda function. Take a look at this function definition:

```
def echo_word(word1, echo):
    """Concatenate echo copies of word1."""
    words = word1 * echo
    return words
```

The function echo_word takes 2 parameters: a string value, word1 and an integer value, echo. It returns a string that is a concatenation of echo copies of word1. Your task is to convert this simple function into a lambda function.

Instructions:

- Define the lambda function echo_word using the variables word1 and echo. Replicate what the original function definition for echo_word() does above.
- Call echo_word() with the string argument 'hey' and the value 5, in that order. Assign the call to result.

```
# Define echo_word as a lambda function: echo_word
echo_word = lambda word1, echo : word1 * echo

# Call echo_word: result
result = echo_word("hey", 5)

# Print result
print(result)
```

3.3 Map() and lambda functions

So far, you've used lambda functions to write short, simple functions as well as to redefine functions with simple functionality. The best use case for lambda functions, however, are for when you want these simple functionalities to be anonymously embedded within larger expressions. What that means is that the functionality is not stored in the environment, unlike a function defined with def. To understand this idea better, you will use a lambda function in the context of the map() function.

Recall from the video that map() applies a function over an object, such as a list. Here, you can use lambda functions to define the function that map() will use to process the object. For example:

```
nums = [2, 4, 6, 8, 10]
result = map(lambda a: a ** 2, nums)
```

You can see here that a lambda function, which raises a value a to the power of 2, is passed to map() alongside a list of numbers, nums. The *map object* that results from the call to map() is stored in result. You will now practice the use of lambda functions with map(). For this exercise, you will map the functionality of the add_bangs() function you defined in previous exercises over a list of strings.

Instructions:

- In the map() call, pass a lambda function that concatenates the string '!!!' to a string item; also pass the list of strings, spells. Assign the resulting map object to shout_spells.
- Convert shout_spells to a list and print out the list.

```
# Create a list of strings: spells
spells = ["protego", "accio", "expecto patronum", "legilimens"]

# Use map() to apply a lambda function over spells: shout_spells
shout_spells = map(lambda item : item + "!!!", spells)

# Convert shout_spells to a list: shout_spells_list
shout_spells_list = list(shout_spells)

# Print the result
print(shout_spells_list)
```

3.4 Filter() and lambda functions

In the previous exercise, you used lambda functions to anonymously embed an operation within map(). You will practice this again in this exercise by using a lambda function with filter(), which may be new to you! The function filter() offers a way to filter out elements from a list that don't satisfy certain criteria.

Your goal in this exercise is to use filter() to create, from an input list of strings, a new list that contains only strings that have more than 6 characters.

Instructions:

- In the filter() call, pass a lambda function and the list of strings, fellowship. The lambda function should check if the number of characters in a string member is greater than 6; use the len() function to do this. Assign the resulting filter object to result.
- Convert result to a list and print out the list.

```
# Create a list of strings: fellowship
fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'legolas', 'gimli', 'gandalf']

# Use filter() to apply a lambda function over fellowship: result
result = filter(lambda member : len(member) > 6, fellowship)

# Convert result to a list: result_list
result_list = list(result)

# Print result_list
print(result_list)
```

3.5 Reduce() and lambda functions

You're getting very good at using lambda functions! Here's one more function to add to your repertoire of skills. The reduce() function

is useful for performing some computation on a list and, unlike map() and filter(), returns a single value as a result. To use reduce(), you must import it from the functools module.

Remember gibberish() from a few exercises back?

```
# Define gibberish
def gibberish(*args):
    """Concatenate strings in *args together."""
    hodgepodge = ''
    for word in args:
        hodgepodge += word
    return hodgepodge
```

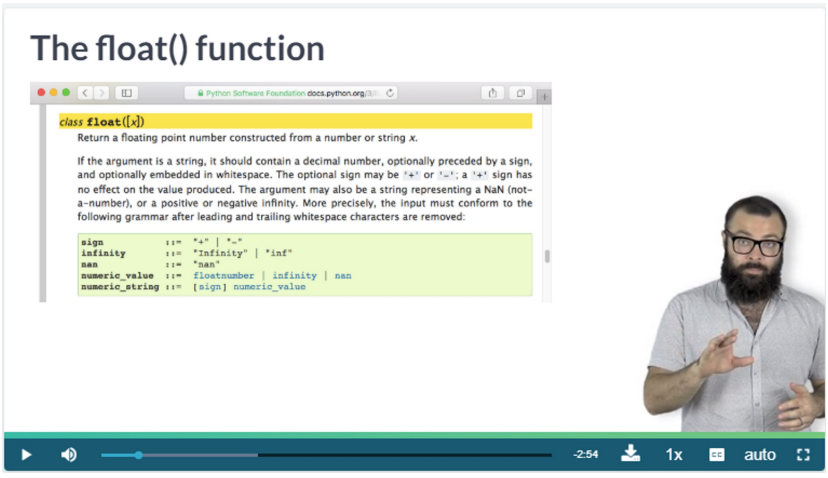
gibberish() simply takes a list of strings as an argument and returns, as a single-value result, the concatenation of all of these strings. In this exercise, you will replicate this functionality by using reduce() and a lambda function that concatenates strings together.

Instructions:

- Import the reduce function from the functools module.
- In the reduce() call, pass a lambda function that takes two string arguments item1 and item2 and concatenates them; also pass the list of strings, stark. Assign the result to result. The first argument to reduce() should be the lambda function and the second argument is the list stark.

```
# Import reduce from functools
from functools import reduce

# Create a list of strings: stark
stark = ['robb', 'sansa', 'arya', 'brandon', 'rickon']

# Use reduce() to apply a lambda function over stark: result
result = reduce(lambda item1, item2: item1 + item2, stark)

# Print the result
print(result)
```

3.6 Pop quiz about errors



Passing an incorrect argument

```
float(2)
```

```
2.0
```

```
float('2.3')
```

```
2.3
```

```
float('hello')
```

```
<hr />---------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-d0ce8bccc8b2> in <module>()
<hr />-> 1 float('hi')
ValueError: could not convert string to float: 'hello'
```

-2:33 1x auto

Passing valid arguments

```
def sqrt(x):
    """Returns the square root of a number."""
    return x ** (0.5)
sqrt(4)
```

```
2.0
```

```
sqrt(10)
```

```
3.1622776601683795
```

-2:18 1x auto

Passing invalid arguments

```
sqrt('hello')
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-cfb99c64761f> in <module>()
----> 1 sqrt('hello')
<ipython-input-1-939b1a60b413> in sqrt(x)
      1 def sqrt(x):
----> 2     return x**(0.5)
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'float'
```

-1:55 1x auto

Errors and exceptions

```
def sqrt(x):
    """Returns the square root of a number."""
    try:
        return x ** 0.5
    except:
        print('x must be an int or float')

sqrt(4)
```

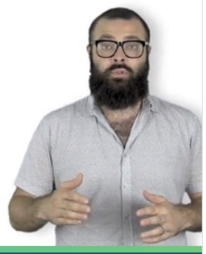
```
2.0
```

```
sqrt(10.0)
```

```
3.1622776601683795
```

```
sqrt('hi')
```

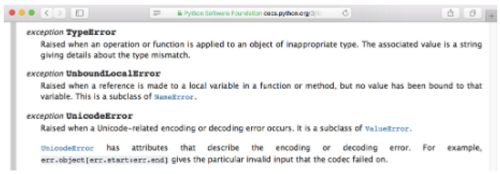
```
x must be an int or float
```



-1:08 1x auto

Errors and exceptions

```
def sqrt(x):
    """Returns the square root of a number."""
    try:
        return x ** 0.5
    except TypeError:
        print('x must be an int or float')
```



exception **TypeError**
Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.

exception **UnboundLocalError**
Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable. This is a subclass of `NameError`.

exception **UnicodeError**
Raised when a Unicode-related encoding or decoding error occurs. It is a subclass of `ValueError`.

`UnicodeError` has attributes that describe the encoding or decoding error. For example, `err.object[err.start:err.end]` gives the particular invalid input that the codec failed on.

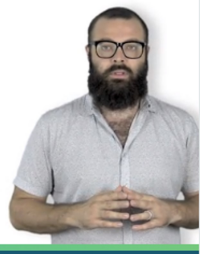
-0:51 1x auto

Errors and exceptions

```
sqrt(-9)
```

```
(1.8369701987210297e-16+3j)
```

```
def sqrt(x):
    """Returns the square root of a number."""
    if x < 0:
        raise ValueError('x must be non-negative')
    try:
        return x ** 0.5
    except TypeError:
        print('x must be an int or float')
```



-0:26 1x auto

In the video, Hugo talked about how errors happen when functions are supplied arguments that they are unable to work with. In this exercise, you will identify which function call raises an error and what type of error is raised.

Take a look at the following function calls to len():

len('There is a beast in every man and it stirs when you put a sword in his hand.')
len(['robb', 'sansa', 'arya', 'eddard', 'jon'])
len(525600)
len(('jaime', 'cersei', 'tywin', 'tyrion', 'joffrey'))
Which of the function calls raises an error and what type of error is raised?

Possible Answers

- [ ] The call len('There is a beast in every man and it stirs when you put a sword in his hand.') raises a TypeError.
- [ ] The call len(['robb', 'sansa', 'arya', 'eddard', 'jon']) raises an IndexError.
- [x] The call len(525600) raises a TypeError.
- [ ] The call len(('jaime', 'cersei', 'tywin', 'tyrion', 'joffrey')) raises a NameError.

```
In [1]: len('There is a beast in every man and it stirs when you put a sword in his hand.')
Out[1]: 76

In [2]: len(525600)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    len(525600)
TypeError: object of type 'int' has no len()

In [3]: len(['robb', 'sansa', 'arya', 'eddard', 'jon'])
Out[3]: 5

In [4]: len(('jaime', 'cersei', 'tywin', 'tyrion', 'joffrey'))
Out[4]: 5
```

3.7 Error handling with try-except

A good practice in writing your own functions is also anticipating the ways in which other people (or yourself, if you accidentally misuse your own function) might use the function you defined.

As in the previous exercise, you saw that the len() function is able to handle input arguments such as strings, lists, and tuples, but not int type ones and raises an appropriate error and error message when it encounters invalid input arguments. One way of doing this is through exception handling with the try-except block.

In this exercise, you will define a function as well as use a try-except block for handling cases when incorrect input arguments are passed to the function.

Recall the shout_echo() function you defined in previous exercises; parts of the function definition are provided in the sample code.

Your goal is to complete the exception handling code in the function definition and provide an appropriate error message when raising an error.

Instructions:

- Initialize the variables `echo_word` and `shout_words` to empty strings.
- Add the keywords `try` and `except` in the appropriate locations for the exception handling block.
- Use the `*` operator to concatenate `echo` copies of `word1`. Assign the result to `echo_word`.
- Concatenate the string `'!!!'` to `echo_word`. Assign the result to `shout_words`.

```
# Define shout_echo
def shout_echo(word1, echo=1):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""

    # Initialize empty strings: echo_word, shout_words
    echo_word = ''
    shout_words = ''
    # Add exception handling with try-except
    try:
        # Concatenate echo copies of word1 using *: echo_word
        echo_word = word1 * echo

        # Concatenate '!!!' to echo_word: shout_words
        shout_words = echo_word + "!!!"
    except:
        # Print error message
        print("word1 must be a string and echo must be an integer.")

    # Return shout_words
    return shout_words

# Call shout_echo
shout_echo("particle", echo="accelerator")
```

3.8 Error handling by raising an error

Another way to raise an error is by using `raise`. In this exercise, you will add a `raise` statement to the `shout_echo()` function you defined before to raise an error message when the value supplied by the user to the `echo` argument is less than 0.

The call to `shout_echo()` uses valid argument values. To test and see how the `raise` statement works, simply change the value for the `echo` argument to a *negative* value. Don't forget to change it back to valid values to move on to the next exercise!

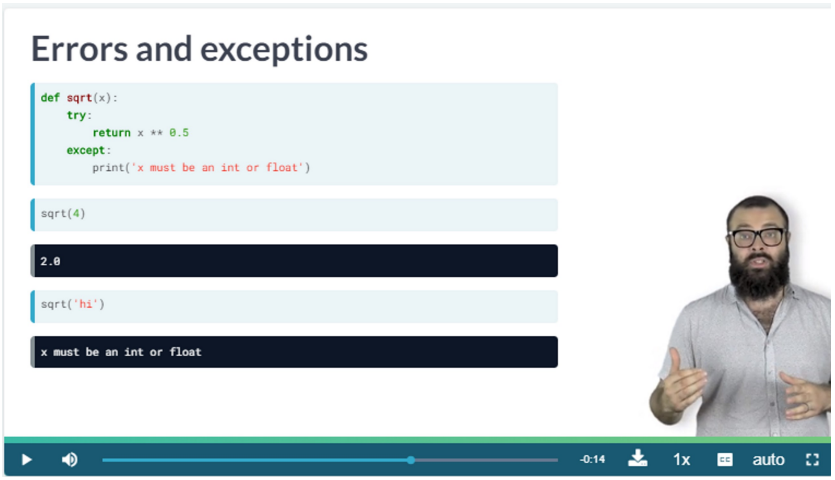
Instructions:

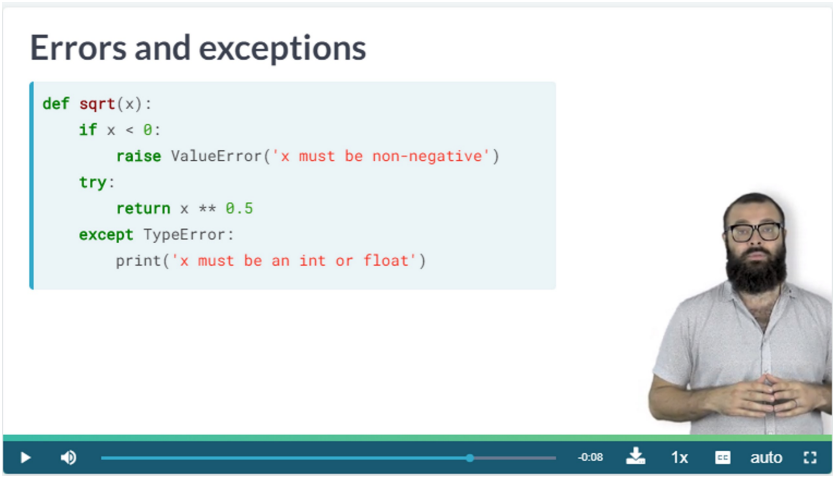
- Complete the `if` statement by checking if the value of `echo` is *less than* 0.
- In the body of the `if` statement, add a `raise` statement that raises a `ValueError` with message `'echo must be greater than or equal to 0'` when the value supplied by the user to `echo` is less than 0.

```
# Define shout_echo
def shout_echo(word1, echo=1):
    """Concatenate echo copies of word1 and three
    exclamation marks at the end of the string."""

    # Raise an error with raise
    if echo < 0:
        raise ValueError("echo must be greater than or equal to 0")

    # Concatenate echo copies of word1 using *: echo_word
    echo_word = word1 * echo

    # Concatenate '!!!' to echo_word: shout_word
    shout_word = echo_word + '!!!'

    # Return shout_word
```

```
    return shout_word

# Call shout_echo
shout_echo("particle", echo=5)
```

3.9 Bringing it all together (1)





This is awesome! You have now learned how to write anonymous functions using lambda, how to pass lambda functions as arguments to other functions such as map(), filter(), and reduce(), as well as how to write errors and output custom error messages within your functions. You will now put together these learnings to good use by working with a Twitter dataset. Before practicing your new error handling skills,in this exercise, you will write a lambda function and use filter() to select retweets, that is, tweets that begin with the string 'RT'.

To help you accomplish this, the Twitter data has been imported into the DataFrame, tweets_df. Go for it!

Instructions:

- In the filter() call, pass a lambda function and the sequence of tweets as strings, tweets_df['text']. The lambda function should check if the first 2 characters in a tweet x are 'RT'. Assign the resulting filter object to result. To get the first 2 characters in a tweet x, use x[0:2]. To check equality, use a Boolean filter with ==.
- Convert result to a list and print out the list.

```
# Select retweets from the Twitter DataFrame: result
result = filter(lambda x : x[0:2] == 'RT', tweets_df['text'])

# Create list from filter object result: res_list
res_list = list(result)

# Print all retweets in res_list
for tweet in res_list:
```

```
    print(tweet)
```

3.10 Bringing it all together (2)

Sometimes, we make mistakes when calling functions - even ones *you* made yourself. But don't fret! In this exercise, you will improve on your previous work with the count_entries() function in the last chapter by adding a try-except block to it. This will allow your function to provide a helpful message when the user calls your count_entries() function but provides a column name that isn't in the DataFrame.

Once again, for your convenience, pandas has been imported as pd and the 'tweets.csv' file has been imported into the DataFrame tweets_df. Parts of the code from your previous work are also provided.

Instructions:

- Add a try block so that when the function is called with the correct arguments, it processes the DataFrame and returns a dictionary of results.
- Add an except block so that when the function is called incorrectly, it displays the following error message: 'The DataFrame does not have a ' + col_name + ' column.'.

```
# Define count_entries()
def count_entries(df, col_name='lang'):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Add try block
    try:
        # Extract column from DataFrame: col
        col = df[col_name]

        # Iterate over the column in dataframe
        for entry in col:

            # If entry is in cols_count, add 1
            if entry in cols_count.keys():
                cols_count[entry] += 1
            # Else add the entry to cols_count, set the value to 1
            else:
                cols_count[entry] = 1

        # Return the cols_count dictionary
        return cols_count

    # Add except block
    except:
        print('The DataFrame does not have a ' + col_name + ' column.')

# Call count_entries(): result1
result1 = count_entries(tweets_df, 'lang')

# Print result1
print(result1)
```

3.11 Bringing it all together (3)

In the previous exercise, you built on your function count_entries() to add a try-except block. This was so that users would get helpful messages when calling your count_entries() function and providing a column name that isn't in the DataFrame. In this exercise, you'll instead raise a ValueError in the case that the user provides a column name that isn't in the DataFrame.

Once again, for your convenience, pandas has been imported as pd and the 'tweets.csv' file has been imported into the DataFrame tweets_df. Parts of the code from your previous work are also provided.

Instructions:

- If col_name is *not* a column in the DataFrame df, raise a ValueError 'The DataFrame does not have a ' + col_name + ' column.'.
- Call your new function count_entries() to analyze the 'lang' column of tweets_df. Store the result in result1.
- Print result1. This has been done for you, so hit 'Submit Answer' to check out the result. In the next exercise, you'll see that it raises the necessary ValueErrors.

```
# Define count_entries()
def count_entries(df, col_name='lang'):
    """Return a dictionary with counts of
    occurrences as value for each key."""

    # Raise a ValueError if col_name is NOT in DataFrame
    if col_name not in df.columns:
        raise ValueError('The DataFrame does not have a ' + col_name + ' column.')

    # Initialize an empty dictionary: cols_count
    cols_count = {}

    # Extract column from DataFrame: col
    col = df[col_name]

    # Iterate over the column in DataFrame
    for entry in col:

        # If entry is in cols_count, add 1
        if entry in cols_count.keys():
            cols_count[entry] += 1
            # Else add the entry to cols_count, set the value to 1
        else:
            cols_count[entry] = 1

        # Return the cols_count dictionary
    return cols_count

# Call count_entries(): result1
result1 = count_entries(tweets_df, 'lang')

# Print result1
print(result1)
```

3.12 Bringing it all together: testing your error handling skills

You have just written error handling into your count_entries() function so that, when the user passes the function a column (as 2nd argument) NOT contained in the DataFrame (1st argument), a ValueError is thrown. You're now going to play with this function: it is loaded into pre-exercise code, as is the DataFrame tweets_df. Try calling count_entries(tweets_df, 'lang') to confirm that the function behaves as it should. Then call count_entries(tweets_df, 'lang1'): what is the last line of the output?

Possible Answers

- [ ] 'ValueError: The DataFrame does not have the requested column.'
- [x] 'ValueError: The DataFrame does not have a lang1 column.'
- [ ] 'TypeError: The DataFrame does not have the requested column.'