python_data_science_pt1_printVersion

April 5, 2019

```
In [10]: import pandas as pd
         import os

         tweets = pd.read_csv('tweets.csv')
```

1 Writing your own Functions

Docstrings - Used to describe what your function does or its' return value. They are placed in the immediate line after function header

```
In [2]: # 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)
```

```
congratulations!!!
```

Now we're going to learn to pass multiple arguements and return multiple values.
Multiple values cna be returned using `tuples`
Tuples: * Can contain multiple values. * But they values in a tuple can't be changed. * Tuples are constructed using parentheses `()`
Accessing tuple elements are done the same way you acces list elements

```
In [3]: # Tuples!!!

        # Create a tuple
```

```
        nums = (1, 3, 8)
        print(nums)

        # Index the tuple
        print(nums[1])

        # Unpack the tuples
        num1, num2, num3 = nums
        print(num1)
        print(num2)
        print(num3)
```

```
(1, 3, 8)
3
1
3
8
```

```
In [4]: # 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)
```

```
congratulations!!!!
you!!!!
```

```
In [5]: # 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)
```

```
{'en': 97, 'et': 1, 'und': 2}
```

```
In [6]: # Define count_entries()

        tweets_df = pd.read_csv('tweets.csv')

        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)
```

```
{'en': 97, 'et': 1, 'und': 2}
```

2 More function stuff! Default arguments, variable-length arguments and scope

SCOPE - Refers to the part of the program where the object may be accessible
There are 3 types of scopes:

1. Global - It's defined in the main body of the script
2. Local - It's defined within a function. Once execution of function is done, it ceaeses to exist
3. Built-in - Names in the pre-define built-ins python module

You can make a variable within a function part of the global scope with the keyword `global`

```
In [7]: num = 5

        def func1():
            num = 3
            print(num)

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


        func1()
        func2()
        print(num)
```

```
3
10
6
```

Below, by assigning a variable to the global scope within a function, it can be altered within the function even if it wasn't explicitly passed

```
In [8]: # 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)
```

```
teen titans
justice league
```

2.0.1 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

```
In [2]: import builtins

        print( dir(builtins), end = ',' )
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError', 'Bu
```

2.1 Nested Functions

A function that's defined within another function

Why you nest functions:

- Repeat a proccess multiple times within a larger function

```
In [10]: # 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'))
```

```
('a!!!!', 'b!!!!', 'c!!!!')
```

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.

```
In [11]: # 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'))
```

```
hellohello hellohellohello
```

Keyword `nonlocal` within a nested function to alter the value of a variable defined in the enclosing scope.

```
In [12]: # 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')
```

```
hellohello
hellohello!!!
```

2.1.1 Default and Flexible Arguments

You can give functions default arguments without having to specify every argument

Flexible Arguments are used when you aren't sure how many arguments a user will want to pass. You simply place `*args` in place of function arguments.

You can also call keyword arguments using `**kwargs`

```
In [13]: # Single default argument

         # 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)
```

```
Hey!!!
HeyHeyHeyHeyHey!!!
```

In [14]:
```
# Multiple Default Arguments

# 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

    # Capitalize echo_word if intense is True
    if intense is True:
        # Capitalize 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("Hey", echo = 5, intense = True)

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

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

```
HEYHEYHEYHEYHEY!!!
HEY!!!
```

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.

Parameters passed to flexible arguments can be called using `args` keyword

```
In [15]: # 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)
```

```
luke
lukeleiahanobidarth
```

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. `**kwargs` is a dictionary.

```
In [16]: # 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, val in kwargs.items():
            # Print out the keys and values, separated by a colon ':'
            print(key + ": " + val)

        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")
```

```
BEGIN: REPORT

name: luke
affiliation: jedi
status: missing

END REPORT

BEGIN: REPORT

name: anakin
affiliation: sith lord
status: deceased

END REPORT
```

Some Advanced/Fancy Stuff

```
In [17]: tweets_df = pd.read_csv('tweets.csv')

    # 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(tweets_df, col_name = 'lang')

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

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

```
{'en': 97, 'et': 1, 'und': 2}
{'<a href="http://twitter.com" rel="nofollow">Twitter Web Client</a>': 24, '<a href="http://ww
```

```
In [18]: # 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)
```

```
{'en': 97, 'et': 1, 'und': 2}
{'en': 97, 'et': 1, 'und': 2, '<a href="http://twitter.com" rel="nofollow">Twitter Web Client<,
```

3 Lambda/Anonymous Functions

There's a quicker way to write functions rather than using `def` by using the keyword `lambda`

It's a quick but potentially dirty way to write functions. Not advised to use all the time but can be very useful at times

ex:

`add_bangs = (lambda a: a + '!!!')`

The below code is taking the `echo_word` function and converting it to lambda function

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

```
In [19]: # 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)
```

```
heyheyheyheyhey
```

`lambda` functions are also useful for `map()` function which takes a function as an argument and applies that functions to all elements in a sequence

ex:

```
nums = [2, 4, 6, 8, 10]

result = map(lambda a: a ** 2, nums)
```

```
In [20]: # 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)

         # Convert shout_spells into a list and print it
         print(shout_spells_list)
```

```
['protego!!!', 'accio!!!', 'expecto patronum!!!', 'legilimens!!!']
```

Similar to above, you can use `lambda` functions as an argument in `filter()` functions

```
In [21]: # Create a list of strings: fellowship
         fellowship = ['frodo', 'samwise', 'merry', 'pippin', 'aragorn', 'boromir', 'legolas',

         # 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)

         # Convert result into a list and print it
         print(result_list)
```

```
['samwise', 'aragorn', 'boromir', 'legolas', 'gandalf']
```

A third function that can use a `lambda` function as an argument is `reduce` which returns input as a single result

To use it, you must first import `functools` module

```
In [22]: # 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)
```

```
robbsansaaryabrandonrickon
```

4 Error Handling

`Exception` - An error occurring during execution
A good way to handle these are using `try` and `except` statements

```
In [15]: # 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")
```

```
word1 must be a string and echo must be an integer.
```

```
Out[15]: ''
```

Another way to raise an error is by using `raise`
ex:

```
if echo < 0:
    raise ValueError("echo must be greater than 0")
```

```
In [14]: # 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 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=-1)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)

<ipython-input-14-c8832a857c08> in <module>()
     18
     19 # Call shout_echo
---> 20 shout_echo("particle", echo=-1)


<ipython-input-14-c8832a857c08> in shout_echo(word1, echo)
      6     # Raise an error with raise
      7     if echo < 0:
----> 8         raise ValueError("echo must be greater than 0")
      9
     10     # Concatenate echo copies of word1 using *: echo_word


ValueError: echo must be greater than 0
```

5 Brining it all together

Now we're going to add error messages previous functions

```
In [12]: tweets_df = pd.read_csv('tweets.csv')

         # 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)
```

```
{'en': 97, 'et': 1, 'und': 2}
```

```
In [13]: tweets_df = pd.read_csv('tweets.csv')

         # 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)

        # Print result1
        print(result1)
```

```
{'en': 97, 'et': 1, 'und': 2}
```