counting_treasures_v1_r2

September 14, 2022

0.1 Counting_Treasures_v1_r2

Get the file treasure.py from Bb. It contains a spec for a simple program, including doctests. Run it using python -m doctest treasure.py. Add code to implement the spec. Do not change the doctests. Put your name(s) and ID(s) as shown at the top. Submit treasure.py. This is worth 5%.

[1]:
```
# I/we declare that this file represents our own work, and that we
# have not seen any work on this assignment done by others, and that
# we have not shown our work to any others.

# Student name(s): Jiarong Li
# Student ID(s): 20230033

# Do not change the formatting above. For multiple names/IDs, use
# commas to separate.


from collections import Counter

def dict_sort(d):
    result = {}
    for k in sorted(d.keys()):
        # in modern Python, dicts remember the order in which their keys
        # were added, and use that order when being printed
        result[k] = d[k]
    return result

def count_treasure(box):
    '''
    While wandering in the fortress of the goblin king, we've discovered a
    box of treasure!

    Count all the treasure and other items in the box and return the
    result as a `dict`, with keys sorted alphabetically.

    `box` is a `dict` specifying the number of each item, eg:
    >>> count_treasure({'coins': 10, 'diamonds': 10})
    {'coins': 10, 'diamonds': 10}
```

```
    The above is a box containing 10 coins and 10 diamonds, so the output
    is as shown.

    A container (the box itself, or a bag, pouch, etc.) can contain other
    containers. Instead of a number, the containers are specified as a
    list, tuple, or similar. We should include the containers in the count:
    >>> count_treasure({'coins': 10,
    ...                 'bags': [{'coins': 2}, {'coins': 5}]})
    {'bags': 2, 'coins': 17}

    Notice the above is a multi-line doctest, using ...

    Containers can be recursive:
    >>> count_treasure({'bags': [{'bags': [{'coins': 10}]}]})
    {'bags': 2, 'coins': 10}

    Here is a bigger example:
    >>> count_treasure({
    ...   'coins': 10,
    ...   'rubies': 10,
    ...   'enchanted pouches': [{
    ...      'coins': 10,
    ...      'rubies': 10,
    ...      'treasure chests': (
    ...         {'coins': 1000},
    ...         {'coins': 1000},
    ...         {'coins': 1000}
    ...      ) # this was a tuple of 3 treasure chests
    ...    }] # this was a list of 1 enchanted pouches
    ... })
    {'coins': 3020, 'enchanted pouches': 1, 'rubies': 20, 'treasure chests': 3}

    If the input is mis-specified, we expect to see an error:
    >>> count_treasure({'bags': (10, 20, 30)})
    Traceback (most recent call last):
    ...
    TypeError: 'int' object is not iterable
    '''

    # HINT: use a `Counter` to store your results while working
    result = Counter()


    ## YOUR CODE HERE

    ## I create an assistant function called identify_type()
```

```
    ## to deal with nested dictionaries/lists/tuples and
    ## return the counter with elements stored.
    result =  identify_type(box, result)


    # HINT: use `dict_sort(result)` at the end to sort and
    # convert to an ordinary `dict`
    return dict_sort(result)
```

[2]:
```
def identify_type(d, result):

    """This method is using to identify the types of values of a
    dictionary and returns a counter.
    We suppose *d* is the value we want to identify.
    We suppose *result* is the counter we passed in.

    """

    if isinstance(d, dict): # If d is an instance of dictionary.
        for key, value in d.items(): # For each item in d.
            if isinstance(d[key], int): # If the value of current item is an␣
↪integer.
                                        # We assume the number of treasures is␣
↪an integer.
                result.update({key: d[key]}) # We update the couter.
            else:                       # If the value of current item is not␣
↪an integer.
                result.update({key: len(d[key])}) # We first store the key and
                                                  # the length of the␣
↪corresposed value.
                identify_type(d[key], result)     # Then identify the type of␣
↪current value.
    elif isinstance(d, list): # If d is an instance of list
        for item in d:
            if not isinstance(item, dict): # If the innermost item is not a␣
↪dictionary type,
                raise TypeError("\'int\' object is not iterable") # it will␣
↪raise an error.
            identify_type(item, result)    # We check the type of current value␣
↪in the list.
    elif isinstance(d, tuple): # If d is an instance of tuple
        for item in d:
            if not isinstance(item, dict):
                raise TypeError("\'int\' object is not iterable")
            identify_type(item, result) # # We check the type of current value␣
↪in the tuple.
```

```
    return result
```

[3]:
```
import doctest
doctest.testmod()
```

[3]:
```
TestResults(failed=0, attempted=5)
```

[ ]:
```

```