COMP10001 Foundations of Computing
Semester 1, 2022

Tutorial Solutions: Week 6

— VERSION: 1477, DATE: APRIL 4, 2022 —

Before the discussion questions, try Exercises 1–2 to revise previous material

Discussion

1. In what situations would we use a "dictionary"? How are they structured? How do we add and delete items?

   A: *A dictionary holds relations between "keys" and "values". It's useful for counting frequencies or storing information related to different objects in your code. Dictionaries are accessed in a similar way to other sequences, by using index notation. Values stored are retrieved by indexing with the associated key (`d[key]`). Values are added by indexing with assignment (`d[key] = value`) and deleted with the `.pop(key)` method, which takes as an argument the key we wish to delete. Working with data stored in dictionaries is easy using the `.keys()` and `.values()` methods, which return a collection of the keys and values respectively; and `.items()` which returns a collection of tuples representing each entry in the dictionary, in `(key, value)` format.*
   An empty dictionary is declared with a pair of braces { }.

2. What is the difference between using the `.pop()` method on a dictionary and using it on a list?

   A: *On a list: `.pop()` called without an index argument removes the last item in the list. Called with an index `.pop(index)` deletes the item at that index in the list. Both times it will return the object it has deleted from the list.*
   On a dictionary: `.pop(key)` deletes the (key: value) pair associated with that key in the dictionary, returning the value it has removed. Without an argument, `.pop()` will not work because unlike lists, dictionaries do not have a reliable ordering of entries. Therefore, `.pop()` needs a key to know which value to delete.

3. In what situations would we use a "set"? How does it differ from other "containers" such as lists and dictionaries?

   A: *A set stores a collection of unique objects. Perhaps most naturally, we may use sets to store a mathematical set of numbers, but we may also store a mixture of any other unique objects. Sets are useful when we want to remove duplicates from some other sequence, or combine sets with set operations.*
   Sets are somewhat like a dictionary without a value for each key: in both cases each entry is unique and there is no concept of an ordering. A list has an order and may have duplicates: both of these attributes are lost when converting to a set.

4. What special operations can we perform on sets? How do we add and remove items from them?

   A: *The three main operations are union: `s1 | s2` or `s1.union(s2)`; intersection: `s1 & s2` or `s1.intersection(s2)`; and difference: `s1 - s2` or `s1.difference(s2)`. Adding an item is possible with the `.add(item)` method and removing an item is done with the `.remove(item)` method.*
   Note that since a pair of empty braces { } denotes an empty dictionary, we use `set()` in order to create an empty set.

   Now try Exercises 3–4

5. What is `None`? How is it used?

   A: *`None` is a special value in Python, notable for being what's passed as the return value of a function when no return value is specified. It can therefore be used to represent the absence of a result, perhaps as a somewhat third option to a True/False boolean result. `None` is the value you will find if you assign the "output" from many mutating methods such as `.append()` which do not return anything. `None` is its own type, so no value of any other type has equality with it.*

6. What is the difference between `sorted()` and `.sort()` when applied to a list? What does it mean to edit an object "in-place"?

   A: *Say we're talking about a list `my_list`. Both `sorted(my_list)` and `my_list.sort()` will sort `my_list`. `sorted(my_list)` will return a new list which contains the items of `my_list` in sorted order. `my_list` is left unchanged by this function. `my_list.sort()` on the other hand, will mutate `my_list`, changing the order of its items to sort it. Nothing is returned from this method (`None` if you try to assign its output) because it does its work directly on the list. The original order of items in `my_list` is overwritten.*
   Editing an object in-place means mutating it: editing it directly without creating a copy or returning a new obejct. It can be dangerous if you're not sure you want your data to be changed so be careful!

Exercises

1. Do the following code snippets do the same thing? What are some advantages and disadvantages of each snippet? What if we needed a hundred different types of tool?

```python
print("We␣need␣some␣saws")
print("We␣need␣some␣hammers")
print("We␣need␣some␣cogs")
print("We␣need␣some␣nails")
```

```python
def get_str(part):
    return f"We␣need␣some␣{part}"

print(get_str("saws"))
print(get_str("hammers"))
print(get_str("cogs"))
print(get_str("nails"))
```

```python
def get_str(part):
    return f"We␣need␣some␣{part}"

parts = ("saws", "hammers", "cogs", "nails")

for part in parts:
    print(get_str(part))
```

A: *These snippets will all result in the same output. The first simply prints the lines which is relatively short but results in a lot of repetition of `"We␣need␣some"`. The second abstracts the creation of the string to be printed in a function which is more elegant, less repetitive and easier to edit. The third removes all repetition by using a for loop to iterate over the parts to be used in the `get_str()` function. This results in code which makes use of all programming structures available to be very clear and efficient. Crucially, while the first two examples will get far longer with each extra tool added, the third one will not, as it will be only the `parts` tuple which gets bigger, meaning this code scales much more easily to cope with hundreds of tools.*

2. Consider the following `while` loop and two conversions to `for` loops. Are the two `for` loops equivalent? Why might you choose one over the other?

```python
count = 0
items = ('eggs', 'spam', 'more_eggs')
while count < len(items):
    print(f"we␣need␣to␣buy␣more␣{items[count]}")
    count += 1
```

```python
items = ('eggs', 'spam', 'more_eggs')
for count in range(len(items)):
    print(f"we␣need␣to␣buy␣more
{items[count]}")
```

```python
items = ('eggs', 'spam', 'more_eggs')
for item in items:
    print(f"we␣need␣to␣buy␣more
{item}")
```

A: *Both are functionally equivalent and will do the same thing. The first uses range() to get indices which index `items`, making it closer to original loop. The second is cleaner since it iterates through list directly.*

3. Evaluate the following given the assignment `d = {"R": 0, "G": 255, "B": 0, "other": {"opacity": 0.6}}`. Specify whether the value of `d` changes as a result. Assume `d` is reset to its original value each time.

   (a) `"R"in d`

   A: *`True` (test for membership among keys)*

   (b) `d["R"]`

   A: *`0`*

   (c) `d["R"] = 255`

   A: *New value of `d`:*
   `{'R': 255, 'G': 255, 'B': 0, 'other': {'opacity': 0.6}}`

   (d) `d["A"]`

   A: *`KeyError: 'A'` (requesting a nonexistent key gives an error)*

   (e) `d["A"] = 50`

   A: *New value of `d`:*
   `{'R': 0, 'G': 255, 'B': 0, 'other': {'opacity': 0.6}, 'A': 50}`
   (assigning to a nonexistent key adds that (key: value) pair to the dictionary)

(f) `d.pop("G")`

A: `255` *(removes key from dictionary, returning its value)*
New value of `d`*:*
`{'R': 0, 'B': 0, 'other': {'opacity': 0.6}}`

(g) `d["other"]["blur"] = 0.1`

A: *New value of* `d`*:*
`{'R': 0, 'G': 255, 'B': 0, 'other': {'opacity': 0.6, 'blur': 0.1}}`

(h) `d.items()`

A: `dict_items([('R', 0), ('G', 255), ('B', 0), ('other', {'opacity': 0.6})])`

4. Evaluate the following given the assignment `s1 = {1, 2, 4}` and `s2 = {3, 4, 5}`. If `s1` or `s2` change as a result, give their new value. Assume `s1` and `s2` are reset to their original values each time.

(a) `s1.add(7)`

A: *New value of* `s1`*:*
`{1, 2, 4, 7}`

(b) `s1.add(2)`

A: `s1` *does not change (2 is already in the set)*

(c) `s2.remove(5)`

A: *New value of* `s2`*:*
`{3, 4}`

(d) `s1 & s2`

A: `{4}`

(e) `s1.union(s2)`

A: `{1, 2, 3, 4, 5}`

(f) `s1 - s2`

A: `{1, 2}`

Problems

1. Write a function which takes a string as input and prints the frequency of each character in the string using a dictionary. `freq_counts('booboo')` should print:

```
b 2
o 4
```

A:

```python
def freq_count(words):
    freqs = {}
    for letter in words:
        if letter in freqs:
            freqs[letter] += 1
        else:
            freqs[letter] = 1
    for key in freqs:
        print(key, freqs[key])
```

2. Write a function which takes two lists as input and returns a list containing the numbers which they both have in common. `in_common([1, 2, 4], [3, 4, 5])` should return `[4]`.

A:

```python
def in_common(list_1, list_2):
    set_1 = set(list_1)
    set_2 = set(list_2)
    common = set_1 & set_2
    return list(common)
```

3. Write a function which takes a dictionary and returns a sorted list containing the unique values in that dictionary. `unique_values({'a': 1, 'b': 0, 'c': 0})` should return `[0, 1]`.

A: *Note that this solution only works if the values of the input dictionaries are immutable, since the values in a set must be immutable.*

```python
def unique_values(d):
    values = d.values()
    set_values = set(values)
    sorted_values = sorted(set_values)
    return sorted_values
```

4. Write a function which takes a string, a character and an integer threshold and returns `True` if the character appears in the string with a frequency above the threshold, `False` if it appears at or below the threshold, and `None` if it doesn't appear at all. `above_thresh('I␣like␣the␣letter␣e', 'e', 3)` should return `True`.

   A:

```
def above_thresh(text, char, threshold):
    if char not in text:
        return None
    freq_dict = {}
    for letter in text:
        if letter in freq_dict:
            freq_dict[letter] += 1
        else:
            freq_dict[letter] = 1
    return freq_dict[char] > threshold
```