COMP10001 Foundations of Computing
Semester 1, 2019

Tutorial Questions: Week 7

— VERSION: 1533, DATE: APRIL 16, 2019 —

Discussion

1. What do we mean by "mutability"? Which data types are mutable out of what we've seen?

   A: *If an object is mutable, its contents can be changed after it's created. Mutable objects include lists, dictionaries and sets. Immutable types can't be changed short of actually creating a new object (`a += 1` does this with an integer for example). Immutable objects are ints, floats, strings and tuples. The difference between lists and tuples is that tuples are immutable, while lists are mutable.*

2. What is the difference between `sorted()` and `.sort()` when applied to a lsit? 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 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!

3. What is a "namespace"? What do we mean by "local" and "global" namespace?

   A: *A namespace is the collection of variables which can be used in a certain part of your program. In effect, variables which are used inside a function are only accessible from within that function. When we are executing code inside that function, we're only allowed to use variables in the namespace of the function, which means we can't use variables defined in other functions, for instance. The namespace of a function is known as a local namespace as it is specific to that function, whereas anything defined outside of a function is in the global namespace, and is accessible to any function defined inside it.*
   Note that a function defined in another (while a strange thing to do) will be able to access the variables in the outer function, given they're not "overshadowed" by variables with the same name in the inner function.

   Now try Exercises 1 & 2

4. Why is it important to write comments for the code we write? Wouldn't it save time, storage space and processing time to write code without comments?

   A: *Comments are important so that others looking at our code in the future are able to understand what it intends to do and why we've made certain choices about what code to use. It also helps the programmer who wrote the code remember what they were thinking when they wrote the code in the first place, especially if they haven't been working on it in a while. If a bug is found in the code and it needs to be fixed, comments help guide whoever is maintaining it in understanding where the problem lies, and communicating the logic of the program if it's not clear.*
   It would save time to not write comments, but in the long term much time might be wasted debugging if it's impossible to tell why certain choices had been made by the writer of the code. The time taken to properly document code is more than made up for in how much less error-prone and easy to read it is. Also, all comments are discarded before a program is run, so there's no performance cost for them to be there.

5. How should we choose variable names? How do good variable names add to the readability of code?

   A: *Variable names should accurately name the object they're storing. They should allow code to be readable, so you can understand which information is being passed around, processed and stored based on which variables are in use. If bad variable names, such as the arbitrary `a`, `b` ect. are used, it is much harder to understand what data is being stored. It's good practice to write names reflecting what the variable represents, not what it stores: for example `names` instead of `string`.*

6. What are "magic numbers"? How do we write code without them and how should we store global constants?

   A: *Magic numbers are constants which are written into code as literals, making it very difficult to understand what their purpose is. An example could be a threshold such as a pass mark: if written as `if mark > 0.5:` the meaning of 0.5 is obscured, just like using a bad variable name. Instead, we should use capitals to define a global constant at the start of our program `PASS_MARK = 0.5` and then refer to that variable (which is actually constant because we will never change it) where necessary in the code: `if mark > PASS_MARK:` This makes our code much more readable because we can see where we are using a constant value by the capitals and also understand what it represents. We can also edit the value of a constant easily at the top of our code.*

7. What is a "docstring"? What is its purpose?

A: *A docstring is a big comment which we write for a function to describe its operation. It helps give an overview of what the function does as well as communicating its inputs and outputs. The main purpose of doctrings is to help programmers who aren't familiar with a function to understand what it does, giving them an explanation of how to use it. Along with general comments, good docstrings are key to well-documented and readable code.*

Now try Exercises 3 & 4

Exercises

1. What is the output of this code? Why?

```
def mutate(x, y):
    x = x + "The␣End"
    y.append("The␣End")
    print(x)
    print(y)

my_str = "It␣was␣a␣dark␣and␣stormy␣night."
my_list = my_str.split()
my_list_2 = my_list
mutate(my_str, my_list_2)
print(my_str)
print(my_list_2)
```

A:

```
It was a dark and stormy night.The End
['It', 'was', 'a', 'dark', 'and', 'stormy', 'night.', 'The␣End']
It was a dark and stormy night.
['It', 'was', 'a', 'dark', 'and', 'stormy', 'night.', 'The␣End']
```

The difference between what's happening with the string and the list is that the list is mutated while the string is not (it's an immutable type). Passed into a function, there's still a link between a list inside the function and one outside (the parameter `y` *and the argument* `my_list_2`*) so a mutation done inside the function has effect outside as well. However, when reassigning a variable as we do with* `x` *in the function, we're only altering the internal variable and not making any change to* `my_str` *outside the function. Basically, only mutation has effect outside a function, anything else is local and does not cross over to the calling code.*

2. What is the output of the following code? Classify the variables by which namespace they belong in.

```
def foo(x, y):
    a = 42
    x, y = y, x
    print(a, b, x, y)

a, b, x, y = 1, 2, 3, 4
foo(17, 4)
print(a, b, x, y)
```

A:

```
42 2 4 17
1 2 3 4
```

`a`*,* `b`*,* `x` *and* `y` *are all global variables. In the function* `foo()`*,* `a` *is overshadowed by local variable* `a` *and* `x` *&* `y` *are overshadowed by the parameters* `x` *and* `y`*.* `b` *references the global variable as there is no variable* `b` *declared inside the function.*
No change is made to global variables `a`*,* `b`*,* `x` *or* `y` *since the only changes* `foo()` *can make are to its internal local variables.*

3. Consider the following program. How would you improve the variable names and use of magic numbers to make it easier to read?

(a)
```
a = float(input("Enter␣days:␣"))
b = a * 24
c = b * 60
d = c * 60
print("There␣are", b, "hours,", c, "minutes,", d, "seconds␣in", a, "days")
```

A:

```
HOUR_DAY = 24
MINUTE_HOUR = 60
SECOND_MINUTE = 60

days = float(input("Enter␣days:␣"))
hours = days * HOUR_DAY
minutes = hours * MINUTE_HOUR
seconds = minutes * SECOND_MINUTE
print("There␣are", hours, "hours,", minutes,
      "minutes", seconds, "seconds␣in", days, "days")
```

Using constants for the conversion multipliers and appropriate variable names will make this code much more easy to read.

(b)
```
word = input("Enter␣text:␣")
words = 0
vowels = 0
word_2 = word.split()
for word_3 in word_2:
    words += 1
    for word_4 in word_3:
        word_5 = word_4.lower()
        if word_5 in "aeiou":
            vowels += 1
if vowels/words > 0.4:
    print("Above␣threshold")
```

A:

```
THRESHOLD = 0.4

text = input("Enter␣text:␣")
n_words = 0
n_vowels = 0
words = text.split()
for word in words:
    n_words += 1
    for letter in word:
        letter = letter.lower()
        if letter in "aeiou":
            n_vowels += 1
if n_vowels/n_words > THRESHOLD:
    print("Above␣threshold")
```

Rather than a series of numbered variable names with `word` in them, we've named the variables accurate names from `text` to `word` and `letter`. Matching the plurality of nouns is a good idea, such as naming a list of words `words` while referring to a single word (in the for loop) as `word`. Reassigning to `letter` when converting case is better in this situation that creating a new variable. The prefix of `n_` to variables which count the number of something is useful as it indicates the difference between, for example a collection of words and a number representing an amount of words. A constant `THRESHOLD` has been used in place of a magic number at the end of the code.

4. Fill in the blanks with comments and a docstring for the following function, which finds the average frequency of letters over frequency `n` in `text`. There's no definite right or wrong answer here, try and develop your style.

```
def average_freq_over(text, n):
    """ ... """
    freqs = {}

    # ...
    for letter in text:
        if letter in freqs:
            freqs[letter] += 1
        else:
            freqs[letter] = 1
    # ...
    over = []
    total = 0
    for key in freqs:
        if freqs[key] > n:
            over.append(key)
            total += freqs[key]

    # ...
    average = total / len(over)
    return average
```

A: *Below are suggestions, you may write in a different style while retaining the important information.*
Docstring: Takes a string 'text', and an integer 'n' as input. Calculates and returns the average frequency of letters which occur more than 'n' times in 'text'.
Comment 1: Counts frequencies for each letter in text.
Comment 2: Fills list 'over' with letters which appear over the threshold amount and adds their frequency to a running total
Comment 3: Calculates and returns average

Problems

1. Write a function which takes a string containing an FM radio frequency and returns whether it is a valid frequency. A valid frequency is within the range 88.0-108.0 inclusive with 0.1 increments, meaning it must have only one decimal place.

   A:

```
def valid_fm(freq):
    """ Takes an FM frequency as a string and returns a boolean
    value indicating whether it's a valid frequency or not. """

    # Checks length of string
    if len(freq) != 4 and len(freq) != 5:
        return False

    # Checks characters conform to a number
    if not freq[:-2].isdigit() or freq[-2] != "." or not freq[-1].isdigit():
        return False

    # Returns based on final range check
    return 88.0 <= float(freq) <= 108.0
```

2. Write a function which takes a string and returns a 2-tuple containing the most common letter in the string and its frequency. In the case of a tie, it should return the letter which occurs first in the text.

A:

```python
def most_common_letter(text):
    """ Finds the most common letter in `string` and returns
    that letter in a tuple with its frequency. In case of a
    tie, selects the letter which occured first. """

    letter_freqs = {}
    first_index = {}

    # Builds dictionary of letter frequencies and records first
    # index where each letter was encountered in `first_index`
    for i in range(len(text)):
        letter = text[i]
        if letter not in letter_freqs:
            letter_freqs[letter] = 1
            first_index[letter] = i
        else:
            letter_freqs[letter] += 1

    highest_letter = ''
    highest_freq = 0

    # Iterates through each letter to find most frequent
    for letter in letter_freqs:
        if letter_freqs[letter] > highest_freq:
            # Replaces current highest if more frequent letter found
            highest_letter = letter
            highest_freq = letter_freqs[letter]
        elif letter_freqs[letter] == highest_freq:
            # Tests which letter came first in event of a tie
            if first_index[letter] < first_index[highest_letter]:
                highest_letter = letter

    return (highest_letter, highest_freq)
```