COMP10001 Foundations of Computing
Semester 1, 2022

Tutorial Solutions: Week 7

— VERSION: 1477, DATE: APRIL 11, 2022 —

Discussion

1. 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 how it is written. In the case that some code must be edited after it is first written, comments help guide whoever is maintaining the code to follow what the code does (or if they wrote it, remember what they were thinking when they wrote it in the first place).*
   It is said that code is far more often read than it is written, and while Python's friendly syntax helps with readability, comments communicate program structure to a reader far more effectively.
   It would save time to not write comments, but in the long term much time would be wasted debugging if it's impossible to tell what the code does and why it was written the way it is. The time taken to properly document code is more than made up for in how less error-prone and how much easier 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.

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

   A: *A docstring is like a big comment which we write for a function to describe its operation. It helps anyone using your function to understand what it does. It's important to include in the docstring any inputs and outputs of the function, as well as any important information about what it does. Along with general comments, good docstrings are key to well-documented and readable code.*

   Now try Exercise 1

3. 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 the reader can understand which information is being passed around, processed and stored based on which variables are in use. If bad variable names, such as arbitrary* `a`, `b`, `c` *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* `name` *instead of* `string`.

4. What are "magic numbers"? How do we write code without them by using 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 store these values as global variables named with capital letters at the top of our program:* `PASS_MARK = 0.5` *and then refer to that variable where necessary in the code:* `if mark > PASS_MARK:` *The capital letters indicate that the variable is actually a constant, and our code will never change its value (using global variables in other contexts is bad style). This makes our code much more readable because we can see where we are using a constant value by the capital letters and understand what it represents. We can also edit the value of a constant easily at the top of our program and that change applied to every place that constant is used in our program.*

   Now try Exercise 2

5. 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 (for example,* `a += 1` *creates a new integer assigned to the same variable name). Immutable objects are integers, floats, strings and tuples. The difference between lists and tuples is that tuples are immutable, while lists are mutable.*

6. What is a "namespace"?

   A: *A namespace is a mapping from names (of variables or functions) to objects. It defines the collection of variables which can be used in a certain part of your program.*

7. What do we mean by "local" and "global" namespace? What is "scope"?

   A: *The global namespace is the collection of variables and functions available outside of any functions in a program. When a function is called, it will have a local namespace which is unique to that function's execution and forgotten once it returns.*
   When a variable is referred to, Python looks in the most local namespace first, and if it can't be found there, proceeds to check the global namespace. This means a function can use its local variables and global variables but not variables defined in another function. There is also the subtlety that variables not in the local namespace may not be edited without declaring that you intend to do so. We discourage the editing of global variables from inside a function because it is safer to return values from your function.
   Scope is the area of a program where a particular namespace is used. Variables in a function's local namespace are said to be in the function's scope.
   Note that a function defined in another (while a strange thing to do) will be able to read the variables in the outer function, given they're not "overshadowed" by variables with the same name in the inner function. Think of it like a venn diagram - if a function is inside another, it can access anything directly outside it. Any function can therefore access the global namespace.

Now try Exercises 3 & 4

8. When is it useful to "return early"? How can it make our code safer and more efficient?

   A: *Returning early is where during a long-running computation the answer is returned as soon as it is known rather than waiting for the process to complete. It's often used in the case where a sequence of values is iterated through and a test applied to each one. If we're looking to find whether one value fulfils the test, we can return a* `True` *result as soon as we find a single such value: There's no need to continue testing the rest of the values as nothing will change the output from being* `True`*. This increases the efficiency of the program.*
   This principle is also used in a concept called "short circuiting" in boolean tests where a result is returned as soon as it is known based on the rules of `and` *and* `or`*. For example,* `if num != 0 and 4/num == 2:` *would never cause an error as a false first statement (where num = 0) would cause a* `False` *response to be sent immediately without needing to execute the division by zero which would cause a* `ZeroDivisionError`*. This can make code safer.*

9. What are helper functions? How can they make our code more readable and reusable?

   A: *A helper function is a function that performs some part of the computation of another function. Helper functions can make programs more readable by giving descriptive names to computations. By taking computations out of a function and placing them in helper functions, we can then reuse those helper functions if we ever need that computation again. This is much easier than selectively cutting out parts of a larger function.*

Now try Exercise 5

Exercises

1. Fill in the blanks with comments and a docstring for the following function, which finds the most popular animals by counting ballots. An example for `ballots` is `['dog', 'pig', 'cat', 'pig', 'dog']`, in which case the function returns `['dog', 'pig']`. There's no definite right or wrong answer here, try and develop your style.

```
def favourite_animal(ballots):
    """ ... """
    tally = {}

    # ...
    for animal in ballots:
        if animal in tally:
            tally[animal] += 1
        else:
            tally[animal] = 1

    # ...
    most_votes = max(tally.values())
    favourites = []
    for animal, votes in tally.items():
        if votes == most_votes:
            favourites.append(animal)

    return favourites
```

   A: *Below are suggestions, you may write in a different style while retaining the important information.*
   Docstring: Takes a list 'ballots' as input. Counts the frequency of each animal in 'ballots', and returns a list of the most frequentently voted animals.
   Comment 1: Counts frequencies of each animal in the ballots.
   Comment 2: Find and store the animals that received the highest number of votes.

2. Consider the following programs. What are the problematic aspects of their variable names and use of magic numbers? What improvements would you make to improve readability?

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

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

(b)

```
word = input("Enter text: ")
x = 0
vowels = 0
word_2 = word.split()
for word_3 in word_2:
    x += 1
    for word_4 in word_3:
        if word_4.lower() in "aeiou":
            vowels += 1
if vowels/x > 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:
        if letter.lower() 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 more accurately according to what they store, 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 than creating a new variable, because we never use the original casing after that. 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.

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

```
def mystery(x):
    x.append(5)
    x[0] += 1
    print("mid-mystery:", x)

my_list = [1,2]
print(my_list)
mystery(my_list)
print(my_list)
mystery(my_list.copy())
print(my_list)
```

A:

```
[1, 2]
mid-mystery: [2, 2, 5]
[2, 2, 5]
mid-mystery: [3, 2, 5, 5]
[2, 2, 5]
```

The `mystery()` function is a mutating function, as it changes the elements of the list argument it is given. In the first example, the function is called with the original list and therefore it is mutated: the value printed mid-mystery is reflected outside the function. The second time, a copy of the original list is passed as an argument, meaning any change made inside the function are not applied to the list, as we can see with how the last two lines printed are different.
If the `mystery()` function were changed to alter a string, we would see that no matter how a string is passed into a function, it can never be mutated because it is an immutable type. Returning a new string is the only way to pass changes outside a function.

4. 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 (while they're all immutable types).

5. Compare the two functions below. Are they equivalent? Why would we prefer one over the other?

```
def noletter_1(words, letter='z'):
    for word in words:
        if letter in word:
            return False
    return True

def noletter_2(words, letter='z'):
    no_z = True
    for word in words:
        if letter in word:
            no_z = False
    return no_z

wordlist = ['zizzer'] + ['aardvark'] * 1_000_000
print(noletter_1(wordlist))
print(noletter_2(wordlist))
```

A: *The two functions are functionally equivalent, but the first one uses lazy evaluation while the second one doesn't. In the example of `'zizzer'` followed by a million instances of `'aardvark'`, the first function will perform much faster as it's able to return `False` as soon as it tests `'zizzer'`. The second will continue to iterate through every instance of `'aardvark'` before returning `False`, taking much more time unnecessarily.*

Problems

1. Write a function which takes a list of words and checks whether any of those words are palindromes (spelled the same way backwards as forwards, like "hannah"). It should print True if there are any palindromes and False if there are none. Use lazy evaluation to save some time!
`any_palindrome(['hannah', 'sven'])` should return `True`.

A:

```
def any_palindrome(words):
    for word in words:
        if word == word[::-1]:
            return True
    return False
```

2. Write a function which takes a list of lists of integers and returns a sorted list containing the unique numbers. For example `unique_2d_list([[1, 5, 3], [2], [5, 1, 2]])` should return `[1, 2, 3, 5]`.

A:

```
def unique_2d_list(list_2d):
    unique_nums = set()
    for lst in list_2d:
        unique_nums = unique_nums | set(lst)

    return sorted(unique_nums)
```

3. Write a function which takes a string and returns a 2-tuple containing the most common character in the string and its frequency. In the case of a tie, it should return the character which occurs first in the text. `most_common_char('hi there')` should return `('h', 2)`.

A:

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

    freqs = {}
    first_index = {}

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

    highest_char = ''
    highest_freq = 0

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

    return (highest_char, highest_freq)
```