COMP 204: Sets, Commenting & Exceptions

Mathieu Blanchette, based on material from Carlos G. Oliver, Christopher J.F. Cameron

October 12, 2019

Assignment 2 - Reminders

- The only code that will be graded will be that within the 5 functions.
- All your code that is not part of a function will not be graded. It should be
  - At the end of your program
  - Indented under: if __name__ == "__main__":
  - This makes it easier for the TA to call your functions without executing the entire program.
- This code will not be marked. This is for yourself to call your functions and verify that they work correctly.

- Submit a single Python file named TFBS_scanning.py (DO NOT CHANGE THE FILE NAME).
- Your functions should NOT prompt the user of inputs
- Your functions should NOT print anything
- Your functions MUST *return* an object

Refresher - Looping over elements of a list

Many students are confused about the idea that if we want to loop over the elements of a list, we can either loop over the elements themselves:

```
1  my_list = ["BRCA1", "P53", "RB"]
2  for element in my_list:  # element is a string
3       print(element)
```

or we can loop over the indexes:

```
1  my_list = ["BRCA1", "P53", "RB"]
2  for index in range(len(my_list)):  # index is an int
3       element= my_list[index]
4       print(element)
```

Finding the smallest element of a list - Version 1

Goal: Write a function that returns the smallest element from a list of numbers passed as an argument to the function.

```
1  import math
2
3  def min_list1(L):
4      """ Input: A list L of numbers
5          Return: The smallest element in L
6      """
7      smallest = math.inf
8      for element in L:  # iterates over elements of L
9          if element < smallest:
10             smallest = element
11     return smallest
```

Finding the smallest element of a list - Version 2

```
1  def min_list2(L):
2      """ Input: A list L of numbers
3          Return: The smallest element in L
4      """
5      smallest = math.inf
6      for index in range(len(L)):   # iterates over integers
7                                    # between 0 and len(L)-1
8          if L[index] < smallest:
9              smallest = L[index]
10     return smallest
```

Finding the smallest element of a list - Version 3

```
1  def min_list3(L):
2      """ Input: A list L of numbers
3          Return: The smallest element in L
4      """
5      smallest = math.inf
6      index = 0
7      while index<len(L):   # iterates over integers
8                            # between 0 and len(L)-1
9          if L[index] < smallest:
10             smallest = L[index]
11         index += 1
12     return smallest
```

Finding the smallest element of a list - Version 4

New goal: Return a tuple made of the smallest value in the list AND the index where this value is found in the list.

```
1  def min_list4(L):
2      """ Input: A list L of numbers
3          Return: A tuple containing two elements:
4                  (1) the smallest element in L
5                  (2) the index of the smallest element
6      """
7      smallest = math.inf
8      index_of_smallest = -1  # indicates that no smallest
9                              # element was found
10     for index in range(len(L)):  # iterates over integers
11                                  # between 0 and len(L)-1
12         if L[index] < smallest:
13             smallest = L[index]
14             index_of_smallest = index
15     return smallest,index_of_smallest
```

Commenting: Docstrings

- A triple quoted string directly under a function header is stored as function documentation.
- Specifies what arguments the function takes as input, and what it returns.
- Allows code documentation to be generated automatically

```
1  def min_list4(L):
2      """ Input: A list L of numbers
3          Return: A tuple containing two elements:
4                  (1) the smallest element in L
5                  (2) the index of the smallest element
6      """
```

```
1  >>> help(min_list4)
2  Help on function min_list4 in module __main__:
3          min_list4(L)
4          Input: A list L of numbers
5          Return: A tuple containing two elements:
```

Example: analysis of PWMs (assignment 2)

Task: Write a function called max_PWM_score_and_sequence that takes as argument a PWM (stored as a two-dimensional list of lists; see assignment #2) and returns a tuple made of

1. the highest score achievable with that PWM
2. the sequence that achieves that highest possible score

Example:

```
my_PWM= [[-0.1,  0.5,  0.2,  0.5, -0.1],
         [ 0.2, -0.1,  0.2, -0.1,  0.2],
         [-0.1, -0.2, -0.3, -1.3,  0.3],
         [-0.1, -0.2, -0.9, -1.3, -0.6]]
 score, seq = max_PWM_score_and_sequence(my_PWM)
 # score should be 1.7 = 0.2 + 0.5 + 0.2 + 0.5 + 0.3
 # seq should be either "CAAAG" or "CACAG"
```

Function header and doctstring

```
1  def max_PWM_score_and_sequence(PWM):
2      """ Input: A position weight matrix PWM,
3                 stored as a list of lists of float.
4          Output: A tuple containing two elements:
5                  (1) the highest score achievable with that PWM
6                  (2) the sequence that achieves that highest score
7      """
```

Planning the algorithm

```
1  def max_PWM_score_and_sequence(PWM):
2      # Algorithm (high-level):
3      # - Look for the highest value in each column of the matrix
4      # - Add up those values to get maximum score
5      # - Remember which entry in the matrix yielded each maximum value,
6      #   and produce the DNA sequence that corresponds to it
```

```
1  my_PWM= [[-0.1,  0.5,  0.2,  0.5, -0.1],
2           [ 0.2, -0.1,  0.2, -0.1,  0.2],
3           [-0.1, -0.2, -0.3, -1.3,  0.3],
4           [-0.1, -0.2, -0.9, -1.3, -0.6]]
```

Planning the algorithm in more details

```
1  def max_PWM_score_and_sequence(PWM):
2      # Algorithm (medium-level)
3      # - Use an accumulator variable sum_best_scores (of type float) to
4      #   keep track of the sum of the max values found in each column
5      # - Use an accumulator variable best_sequence (of type String)
6      #   to keep track of the sequence that achieves the best score
7      # - For each column in the PWM:
8      #       - Look for row with the maximum score
9      #       - Add that maximum score to sum_best_scores
10     #       - Add the corresponding nucleotide to best_sequence
```

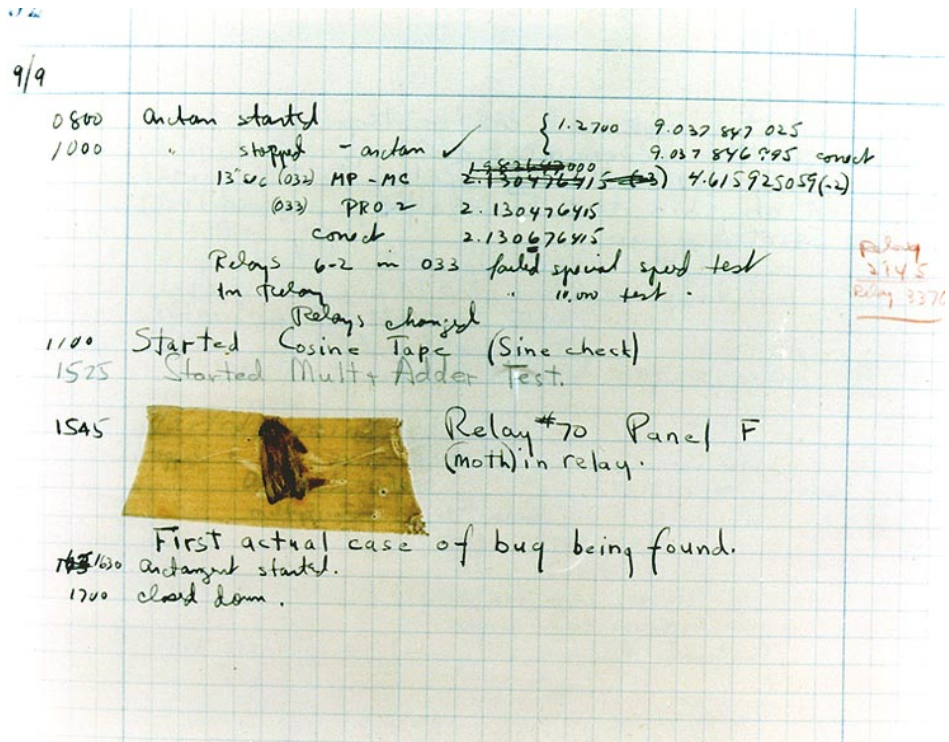
```
1  my_PWM= [[-0.1,  0.5,  0.2,  0.5, -0.1],
2           [ 0.2, -0.1,  0.2, -0.1,  0.2],
3           [-0.1, -0.2, -0.3, -1.3,  0.3],
4           [-0.1, -0.2, -0.9, -1.3, -0.6]]
```

Planning the algorithm in more details

```
1  def max_PWM_score_and_sequence(PWM):
2      sum_best_scores = 0
3      best_sequence = ""
4      number_of_columns = len(PWM[0])  # This is the number of
5                                       # elements in row 0 of the PWM,
6                                       # which is the number of columns
7      nucleotides = ["A","C","G","T"]  # Will be useful later
8
9      for column_index in range(number_of_columns):
10         # Look of row with highest value
11         highest_value_in_row = -math.inf
12         row_index_of_highest_value = -1
13         for row_index in range(len(PWM)): # for each of the 4 rows
14             if PWM[row_index][column_index]>highest_value_in_row:
15                 highest_value_in_row = PWM[row_index][column_index]
16                 row_index_of_highest_value = row_index
17
18         # Now highest_value_in_row contains the highest value in column
19         # and row_index_of_highest_value is the row where it is found
20         sum_best_scores += highest_value_in_row
21
22         best_sequence = best_sequence + \
23                             nucleotides[row_index_of_highest_value]
24
25     return sum_best_scores, best_sequence
```

Bugs: when things break

- You will probably have noticed by now that things don't always go as expected when you try to run your code.
- We call this kind of occurrence a "bug".
- One of the first uses of the term was in 1946 when Grace Hopper's software wasn't working due to an actual moth being stuck in her computer.



[1]

[1] Wikipedia

Types of bugs

There are three major ways your code can go wrong.

1. Syntax errors
2. Exceptions (runtime)
3. Logical (semantic) errors

Syntax Errors: "Furiously sleep ideas green colorless."[2]

- When you get a syntax error it means you violated a writing rule and the interpreter doesn't know how to run your code.
- Your program will crash without running any other commands and produce the message `SyntaxError` with the offending line and a ^ pointing to the part in the line with the error.
- Game: spot the syntax errors!

```
1  print("hello)
2  x = 0
3  while True
4      x = x + 1
5  mylist = ["bob" 2, False]
6  if x < 1:
7  print("x less than 1")
```

[2] Noam Chomsky (1955)

Exceptions: "Colorless green ideas sleep furiously"[3]

- If you follow all the syntax rules, the interpreter will try to execute your code.
- However, the interpreter may encounter into code it is unable to execute, so it raises an Exception
- The program has to deal with this Exception if it is not handled, execution aborts.
- Note: unlike with syntax errors, all the instructions before the interpreter reaches an exception **do** execute.
- Here is a list of all the built-in exceptions and some info on them.

[3] Noam Chomsky (1955)

Logical error

So you've fixed all the syntax errors, you run your program and don't get an Exception, but... the result produced is incorrect...

That's due to a logical error: the execution of your program is not what you intended.

Debugging strategies:

- Use the debugger to run your program step by step (Run current line button, or Step into function button), looking at the values of the variable (Variable Explorer) after each step. Issue: Sometimes requires a lot of clicking before you get to the point of the bug.
- Insert print statements at strategic places in your code, to show what the value of important variables are. Are they what they should be? What is the first point where they become incorrect? That's where your (first) error is.

Exceptions: ZeroDivisionError

- There are many types of exceptions, and eventually you will also be able to define your own exceptions.
- I'll show you some examples of common Exceptions.
- `ZeroDivisionError`

```
1  x = 6
2  y = x  / (x - 6) #syntax is OK, executing fails
3
4  File "test.py", line 2, in <module>
5  y = x / (x - 6)
6  ZeroDivisionError: integer division or modulo by
   ↪  zero
```

Exceptions: NameError

- Raised when the interpreter cannot find a name-binding you are requesting.
- Usually happens when you forget to bind a name, or you are trying to access a name outside your namespace.

```
1  def foo():
2      x = "hello"
3  foo()
4  print(x)
5  Traceback (most recent call last):
6    File "exceptions.py", line 4, in <module>
7      print(x)
8  NameError: name 'x' is not defined
```

Exceptions: IndexError

- Raised when the interpreter tries to access a list index that does not exist

```
1 mylist = ["bob", "alice", "nick"]
2 print(mylist[len(mylist)])
3
4 Traceback (most recent call last):
5   File "exceptions.py", line 2, in <module>
6     print(mylist[len(mylist)])
7 IndexError: list index out of range
```

Exceptions: TypeError

- Raised when the interpreter tries to do an operation on a non-compatible type.

```
>>> mylist = ["bob", "alice", "nick"]
>>> mylist + "mary"

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to
 ↪  list

# this is okay
>>> mylist * 2
["bob", "alice", "nick", "bob", "alice", "nick"]
```

Traceback

- When an exception is raised, you get a traceback message which tells you where the error was raised.

```
    def foo():
        return 5 / 0
    def fee():
        return foo()
    fee()

    Traceback (most recent call last):
  File "exception.py", line 5, in <module>
    fee()
  File "exception.py", line 4, in fee
    return foo()
  File "exception.py", line 2, in foo
    return 5 / 0
ZeroDivisionError: division by zero
```

The rest of this material is OPTIONAL.

Sets: the unordered container for unique things

- Syntax: `myset = {1, 2, 3}` or `myset = set([1, 2, 3])` (careful, `myset = {}` is an empty dictionary)
- Sets never contain duplicates. Python checks this using the `==` operator.

```
1  >>> myset = set([1, 1, 2, 3])
2  set([1,2 , 3]) #only keep unique values
3  >>> myset.add(4)
4  set([1, 2, 3, 4])
5  >>> myset.add(1)
6  set([1, 2, 3, 4])
7  #get unique characters of string
8  >>> charset = set("AAACCGGGA")
9  {A, C, G}
```

- Sets can only contain immutable objects (like dictionary keys)
- Elements in sets do not preserve their order.

Useful set methods and operations

- Membership testing

```
>>> 4 in myset
False
```

- Set intersection (elements common to A and B, if A and B are sets)

```
>>> A = {"a", "b", "c"}
>>> B = {"a", "b", "d"}
>>> A & B # equivalent to: A.intersection(B)
set(["a", "b"])
```

- Click here for a full list of set functionality.

Useful set methods and operations

- Set difference (elements in A that are **not** in B)

```
>>> A - B
set(["c"]) #same as: A.difference(B)
```

- Set union (Elements found in A or B)

```
>>> A | B # equivalent to: A.union(B)
set(["a", "b", "c", "d"])
```

- These can be applied to multiple sets

```
>>> C = {"a", "c", "d", "e"}
>>> A & B & C # A.intersection(B, C)
set(["a"]) #elements common to A and all others
```

Practice problems

1. Write a program that counts the number of unique letters in a given string. E.g. `"bob"` should give `2`.
2. Write a program that checks whether a list of strings contains any duplicates. `['att', 'gga', 'att']` should return `True`

```
# 1. long way
uniques = 0
for c in "bob":
    if c not in bob:
        uniques += 1
#1. short way
len(set("bob"))
#2. long way
uniques = []
mylist = ['att', 'gga', 'att']
for item in mylist:
    if item not in uniques:
        uniques.append('att')
if len(uniques) ! = len(mylist):
    print("found duplicates")
#3. short way
if len(set(mylist)) != len(mylist):
    print("found duplicates")
```

Practice problem: putting it all together

- You're going to create your own dating app. Each user's profile is a dictionary with the following keys:
  - `'movies'` set of strings.
  - `'foods'` set of strings.
  - `'genome'` set of DNA strings.
- The user database will also be a dictionary where each key is a person's name and the value is its profile dictionary.
- E.g. database['bob'] maps to

```
1 {
2  'movies':{'legally blonde', 'mission
   ↪  impossible'}, 'foods': {'mexican',
   ↪  'vegetarian'},
3  'genes': {'AAC', 'AAT', "GGT", "GGA"}
4 }
```

Your app will support 3 functions:

1. `add_user(name, profile, database)` creates a key for the user with its profile info and returns the updated database. (assume all names given are unique)
2. `compatibility_score(user_1, user_2, database)` Returns the compatibility score between two user profiles. Given as:
   - similarity(u1, u2) = # of movies in common + # of foods in common + genome diversity i.e. number of genes in u1 or u2 but not in both.
3. `most_compatible(user, database)` returns user with the highest compatibility score to `user`.

Commenting: rules of thumb

- Comments should be informative but not overly detailed.
- Comments should be indented with the block they address

Which is better?

```
1  #this line binds an empty list to the name
   ↪  'students'
2  students = []
3  for s in students:
4  #loop over list and print
5      print(s)
```

```
1  #keep track of students in a list
2  students = []
3  #display student list
4  for s in students:
5      print(s)
```

Tips on coding style

- Be critical of your code. → is this the best it can be?
- Avoid hard-coding
  - `for i in range(len(mylist))` is better than
  - `for i in range(5)`
- Give objects meaningful names. Avoid names like `string, list, number, result, x, y`
- When lines get too long you are either doing something wrong or you should break the line

```
1 for mylistitem in [innerlistitem in
2     originallist if innerlistitem / 2 + 4 > 9]:
3     print("hi")
```

- A complete description of Python's coding style guidelines is here