ASSIGNMENT(S):
PYTHON

W.Corbo Ugulino (Wallace)

Faculty EEMCS
Module 1 (202001060)
Intro. to Computer Science (202001061)
Year: 2023-2024
Version for Students

UNIVERSITY OF TWENTE.

SEPTEMBER 5, 2023

Assignment 1

Python Fundamentals

CodeGrade automatically grades this Assignment (except for some exercises in which the description says it must be checked manually by your mentor). A submission gets signed-off if the successfully checked exercises are worth at least 40 points (out of 64, the total points of the Assignment). The distribution of points is as follows: 10 points in Entry Level exercises, 18 points in Intermediate Level exercises, and 36 points in Target Level Exercises. This distribution means you can't get enough points to sign-off only by solving the Entry+Intermediate exercises; you will have to solve at least 3 Target exercises.

If you're already a programmer and want to work on more challenging activities, you can focus on the Target exercises (36 points). However, to get a minimum of 40 points, you'll have to solve some of the Intermediate/Entry exercises. It is strongly advised that you solve **all** exercises and even find some more online exercises because the Test requires a certain level of proficiency to be demonstrated in a couple of hours. The more proficient you are, the quicker you'll finish the questions in the Test, and the bigger the chances you will get a satisfactory grade. Finally, the Test aims at the Target level - which means (approximately) 60% of the Test points will be on the Target level. **Good luck, it's time to get hands-on!**!

1.1 Entry Level Exercises

The Entry Level Exercises of this week include a Quiz on Canvas. Follow this link https://canvas.utwente.nl/courses/12812/quizzes/17541 to answer the questions. Create a file named "`python_entry.py`" to implement the **def** functions specified in this section. Each exercise of this section is worth 1 point.

1. Implement the function `def hello_world_python():`
   This function must print to the console the following message:

   ```
   M1BIT:␣Hello,␣Python␣World!
   ```

   In order to be successfully checked by CodeGrade, you must write your code to print the same sequence of characters as shown in the box above. Any extra space affects the evaluation done by CodeGrade. Please, notice that the box above uses a special representation for spaces. These are the only spaces you should add.

2. Implement the function `def hello_world_one_line():`
   This function must have **only one line** of Python code. Still it must print a message that spans across two lines, exactly as shown in the box below:

   ```
   Hello,␣world,␣this␣is␣a␣Python␣def␣function.
   It's␣awesome␣to␣make␣the␣computer␣do␣stuff!
   ```

   Tip: use the escape sequence `\n` in your text (it indicates the end of a line and automatically starts another) to avoid writing two `print` commands.

3. Implement the function `def fancy_hello_world():`
   This function must use the **triple quotes** to print the ASCII ART below:

   ```
       _    ____   ____ ___ ___ 
      / \  / ___| / ___|_ _|_ _|
     / _ \ \___ \| |    | | | | 
    / ___ \ ___) | |___ | | | | 
   /_/   \_\____/ \____|___|___|
   ```

   Figure 1.1: ASCII Art.

   Attention: this exercise may require a manual sign-off by your mentor. If you want to get signed-off automatically by CodeGrade, download the text file from Canvas and don't change/add any character. If (and only if) you are sure that your implementation is correct **and** CodeGrade marks it as incorrect, then ask your mentor to check this exercise manually.

   Tip 1: download this ASCII Art file from Canvas, following the link below:
   https://canvas.utwente.nl/files/2533406/download.
   Tip 2: If you like ASCII Art, check the website below and have fun!
   https://fsymbols.com/text-art/
   Tip 3: There's an example of the use of triple quotes in the function
   `def test_sleep_at_home` of exercise 1.2.1.

4. Implement the function `def grade_is_valid(grade):`
   This function must return `True` if the argument `grade` has its value between `1` and `10`. The function must return `False` if the grade is lesser than `1` **or** greater than `10`.

5. Implement the function `def test_is_valid(test):`
   This function must return `True` if the argument `test` is an `int` (no floating point allowed here) **and** its value is between `1` and `3`. To check if a variable is of `int` type, use the function `isinstance`.
   Example: `isinstance(my_variable, int)`. The former example tests if the variable `my_variable` is of type `int`.

6. Implement the function `def is_the_same(message1, message2):`
   This function must return `True` if the arguments `message1` and `message2` have the same textual content, independently of the characters' case. The function must abort execution if any of the arguments are not of the type `str`. Consider the tests below
   `is_the_same("Flamengo", "fLaMeNgO")` –> Expected result: `True`
   `is_the_same("flamengo", "FLAMENGO")` –> Expected result: `True`
   `is_the_same("flamengo", "flamingo")` –> Expected result: `False`

7. Implement the function `def month_name(month_number):`
   This function receives an argument named `month_number` that must be an `int` number from `1` to `12`. In case this function receives an invalid argument, it must print the following message to the console `"Invalid argument. The month_number must be an int value between 1 and 12"`. Invalid contents for the argument `month_number` are any non-`int` content (like `str` or `float`) and also `int` values that are not between `1 and 12`. Use Switch Case statements for solving this exercise.

   In case the argument is valid, the function must return the name of the month in English, according to the list below:

```
1: "January"
2: "February"
3: "March"
4: "April"
5: "May"
6: "June"
7: "July"
8: "August"
9: "September"
10: "October"
11: "November"
12: "December"
```

8. Implement the function `def sleep_at_home(weekday, vacation):`
   This function helps the cat Tigra figure out where to sleep. It must return `True` when she should sleep at home and `False` when she should sleep away. Tigra sleeps at home on the weekdays and away on the weekends or if she is on vacation. **Tip**: if Tigra is on vacation, she always sleeps away. Test your code with the function `def test_sleep_at_home()` listed below:

```
def test_sleep_at_home():
   print("""
weekday: True, vacation: True.
       Sleep at home? = """, sleep_at_home(True, True))
   print("""
weekday: False, vacation: True.
       Sleep at home? = """, sleep_at_home(False, True))
   print("""
weekday: False, vacation: False.
       Sleep at home? = """, sleep_at_home(False, False))
   print("""
weekday: True, vacation: False.
       Sleep at home? = """, sleep_at_home(True, False))
```

9. Implement the function below

   ```
   def format_name(name, surname):
   ```

   This function must return the name and surname formatted according to the following rule: Initial letter of the name + dot + space + surname + " (" + name + ")". Observe the examples below and the expected result:

   ```
   format_name("Wallace", "Corbo Ugulino")
   ```

   Expected result: `W. Corbo Ugulino (Wallace)`

   ```
   format_name("Luis", "Ferreira Pires")
   ```

   Expected result: `L. Ferreira Pires (Luis)`

   Before formatting the name, this function must validate if the arguments are valid. Name and Surname must be of the type `str` and must have `len >=2`. In case of invalid arguments, print the appropriate message (see below) and abort the execution:

   ```
   inv_name ="Invalid argument. Name must be string (len >= 2)."
   inv_surname ="Invalid argument. Surname must be string (len >= 2)."
   ```

10. Implement the function below

```python
def calculate_ics_grade(grade_python, grade_oscn, grade_java,
                        test_to_add_bonus, bonus_is_full):
```

This function calculates the final grade of the course "Introduction to Computer Science." The final grade is a weighted average of the 3 tests of this course, namely Test 1 (Python), Test 2 (O.S. and Computer Networks), and Test 3 (Java). The weights are defined as follows: Tests 1 and 2 are worth 25% of the final grade each, while Test 3 alone is worth 50% of the Final Grade.

The argument `test_to_add_bonus` must be either `0, 1 or 3`, otherwise the argument is considered invalid. If the argument is invalid, then the following error message must be printed before aborting the execution:
`"Invalid argument test_to_add_bonus. Accepted values are 0, 1 or 3."`

When the argument `test_to_add_bonus` has value `0`, it indicates that the candidate is not eligible for bonus points. A value of `1` indicates that the candidate opted for the bonus to be applied to the Test 1 (Python) grade, while a value of `3` indicates that the candidate opted for the bonus to be applied to the Test 3 (Java) grade.

The bonus can only be applied to the grade of Test 1 or 3 and only if the chosen grade is `>=5 and <=9.5`. If the chosen test grade is not eligible for bonus points, the function must print the following message to the console before aborting the execution:
`Test grade is not eligible to bonus points. Eligible grades are >=5 and <=9.5.`

If the argument `bonus_is_full` is set to `False`, then the candidate is awarded a partial bonus (0.25 points). A partial bonus is only applicable to the Test 1 grade (and only if the grade is in the range mentioned above). In case the argument `bonus_is_full` is `False` and the argument `test_to_add_bonus ==3`, then the function must print the following message to the console before aborting execution:
`"Partial bonus points are not allowed in Test 3."`

If all the checks pass, then the function should return the final grade for the course based on the weights and rules mentioned above.

1.2 Intermediate Level Exercises

Create a file named `python_intermediate.py` to implement the **def** functions specified in this section. Each exercise of this section is worth 2 points.

1. Implement an alternative solution to the exercise 1.1.7 that uses a Python's Dictionary as a replacement for the (*so missed!*) Switch Case statement.

   Dictionary has been used as a replacement solution by some developers that miss the agility of the Switch Case statement. In short, a dictionary is a data structure that has an *ordered* list of $key : value$ pairs. It has some useful methods that we can use to inform the key and obtain its associated value. Make a web search to discover how the Python's Dictionary can be used as a replacement to the Switch Case statement and implement this alternative solution in a function called: `def month_name_alt(month_number):`

   Attention: this exercise must be checked manually by your mentor. If you submit a solution to this exercise, inform your mentor and ask to get the solution checked manually.

2. Implement the function `def linear_search(list_of_items, item):`

   This function receives two arguments: The argument `list_of_items` is the List in which the linear search will be performed. The argument `item` is the element that will be searched in the said List. If `item` is present in `list_of_items`, then this function returns its index in `list_of_items`. If the `item` is not found in the `list_of_items`, then the function must return -1. Before performing the linear search, this function must check if the argument `list_of_items` is not empty. If the list is empty, the function must return the value -2.

3. Implement the function `def reverse_list(list_of_int):`

   This function takes a list of `int` numbers as argument (`list_of_int`), reverses it and returns the reversed list to the caller. Before reversing the list, this function must check if the argument is valid. A valid list has only elements of the type `int` and has `len >=2`. If the argument is invalid, this function must print the message below to the console and abort operations.

   Invalid argument message:
   ```
   "Invalid argument. It must be a list of int (len >= 2)"
   ```

4. Implement the function `def find_max(list_of_int):`

   This function searches the *unordered* list of `int` values (argument) named `list_of_int` and returns the max value found in it (**attention**: it returns the item itself, not its index!). The function must check if the argument is valid before performing the search. The argument `list_of_int` is considered invalid if the list is empty or if it's not a list of `int` values exclusively. You can assume, without the need of validating, that the minimum value possible in this list is **-9999999**, while the max value possible in this list is **+9999999**.

   Invalid argument message:
   `"Invalid argument: list_of_int must be non-empty. Only int allowed."`

5. Implement the function `def remove_duplicates(list_of_names):`

   This function takes an *unordered* list of strings as argument. It must return a list with the same strings, but without duplicates (if any). Before removing the duplicates, this function must check if the argument is valid. Valid arguments are lists with at least 2 elements and all elements must be of type `str`.

   Invalid argument message:
   `"Invalid argument: list_of_names must contain only string (len >= 2)"`

6. Implement the function `def check_type(list_of_items, required_type):`

   This function takes two arguments: a list (named `list_of_items`) and the required type (named `required_type`, expected values are: `str`, `int`, `float`, etc.) for the elements of this list. The function must check each element and return `True` if all elements are instances of the type specified in `required_type`. The function must return `False` if any element is not of the specified type. Once a non-compliant element is found, a message must be printed to the console according to the template below:

   ```
   "Item ’ [item] ’ (of index [item_index])is of type [item_type] while the required
   type is [required_type]"
   ```

   Example:
   ```
   "Item ’ Tomatoes ’ (of index 4 )is of type <class ’str’> while the required type
   is <class ’float’>
   ```

7. Implement the function `def get_longest_word(list_of_words):`

   This function takes a list of words as argument (named `list_of_words`), finds the longest word and returns a tuple with two values: the longest word and its length.

   Validation: the list must have at least 2 elements and all elements must be of type `str`

   Invalid Argument Message:
   `"Invalid Argument. List length must be >= 2, only strings allowed."`

8. Implement the function `def get_least_vowels_word(list_of_words):`

   This function takes a list of words as an argument (named `list_of_words`), finds the word that has less vowels than the others and returns a tuple with two values: the word with the least amount of vowels and the number of vowels.

   Validation: the list must have at least 2 elements and all elements must be of type `str`

   Invalid Argument Message:
   `"Invalid Argument. List length must be >= 2, only strings allowed."`

9. Implement the function `def capitalize_words(sentence):`

   This function takes a string argument named `sentence` and formats the sentence so that all the words are capitalized, i.e., the first letter of each word must be in BIG CAPS, while the other letters of the same word must be lowercase. The function returns the Capitalized version of `sentence`.

   Validation: the argument must be a non-empty string.`str`

   Invalid Argument Message:
   `"Invalid Argument. It must be a non-empty string."`

1.3 Target Level Exercises

Create a file named `python_target.py` to implement the **def** functions specified in this section. Each exercise of this section is worth 4.5 points.

1. Implement the function `def calculate_average_price(products_dict):`
   This function receives a dictionary whose *keys* are the name of a grocery item (type `str`) and the *value* associated to each key is a list of prices (numbers of type `float`). The function must calculate the average price for each product and return a dictionary with each product name as a **key** and the average price of said product as its corresponding **value**, *rounded to 2 decimals*.

   Validation: The function must check if the list of prices of each product is a list of `float` numbers, as expected.
   Hint: remember that you can use the function implemented in exercise 1.2.6 to check the types of the list of prices of each product.
   Tip (Testing): Find below a dictionary with a list of grocery items and a list of prices associated with each product. You can use this dictionary (among others) to test your code.

```
pasta_al_salmone ={
   "Pomodorini/Tomatten/Tomatoes 250g pack": [1.99, 2.49, 1.19, 1.99],
   "Capperi/Kappertjes/Capers": [0.69, 0.79, 0.89, 0.69],
   "Olive oil extra virgin 750ml": [6.59, 7.29, 5.99, 6.39],
   "Prezzemolo/Peterselie/Parsley": [0.99, 0.89, 0.99, 0.99],
   "Coriandolo/Koriander/Coriander": [0.99, 0.89, 0.89, 0.89],
   "Pasta Rummo 500g (Conchiglioni Rigati)": [2.29, 2.99, 2.99, 2.09],
   "Salmone/Zalm/Salmon 250g": [4.99, 4.99, 4.99, 4.99],
   "Aglio/Knoflook/Garlic": [0.99, 0.89, 0.99, 1.09],
   "Olive/Olijven/Olives (Taggiasche)": [2.79, 3.49, 2.99, 3.09],
   "Vino Bianco/Wit Wijn/White Wine (Greco di Tufo)": [14.95, 13.99, 14.18, 16.69],
   "Caffe Lavazza (Black) 250g": [2.79, 3.09, 3.39, 2.79]
}
print(calculate_average_price(pasta_al_salmone))
```

   As a result, you should get the dictionary below:

```
{
   'Pomodorini/Tomatten/Tomatoes 250g pack': 1.92,
   'Capperi/Kappertjes/Capers': 0.77,
   'Olive oil extra virgin 750ml': 6.56,
   'Prezzemolo/Peterselie/Parsley': 0.97,
   'Coriandolo/Koriander/Coriander': 0.92,
   'Pasta Rummo 500g (Conchiglioni Rigati)': 2.59,
   'Salmone/Zalm/Salmon 250g': 4.99,
   'Aglio/Knoflook/Garlic': 0.99,
   'Olive/Olijven/Olives (Taggiasche)': 3.09,
   'Vino Bianco/Wit Wijn/White Wine (Greco di Tufo)': 14.95,
   'Caffe Lavazza (Black) 250g': 3.01
}
```

2. Implement the function `def count_word_occurrences(text, stop_words):`

   This function takes text as the first argument and a list of stop words as the second argument. It must count the occurrence of each word (ignoring character case differences, which means 'Hello' is counted together with 'HELLO') except stop words, and create a dictionary in which the entries are as follows: the word itself (`str`) is the key and the number of its occurrences in `text` is a `int` value associated with this key.

   Validation: `text` must be a non-empty `str` and `list_of_words` must be a non-empty list of `str`. If validation fails, the following message must be printed to the console and the function must abort its execution.

   ```
   inv_text_msg ="Argument text must be a non-empty string"
   inv_stop_word ="stop_words must be a non-empty list of strings"
   ```

   Tip1: Remember that you can use the function implemented in exercise 1.2.6 to check if all elements in a list have a certain required type.
   Tip2: You can find an example list of English stops words on Canvas, link:
   https://canvas.utwente.nl/files/2520462

3. Implement the function `def binary_search(ordered_list, item_to_search):`

   This function takes an *ordered* list of items (`ordered_list`) of any type and uses the binary search algorithm to search for the item specified in the argument `item_to_search`. If the searched item is found, then the function returns its index in the list, otherwise it returns -1.

   Validation: The `ordered_list` must be non-empty and all its elements must have the same type of `item_to_search`. If the validation fails, one of the following message must be printed to the console (according to the error):

   ```
   "Invalid Argument: ordered_list must be non-empty"
   "Type Mismatch: not all items of 'ordered_list' have the same type of 'item_to_search'"
   ```

   Hint: CodeGrade will test this function and give it 2.5 points if it's correct. The other 2 points depend on the manual check by your mentor. Your mentor will check whether you used the binary search algorithm or not, while CodeGrade will test the execution of your function against various test cases.

4. Implement the function `def amplitude_temperature(cities_temperature):`

   This function takes as argument the dictionary `cities_temperature` in which the *key* is a string representing the name of a city and the value associated with each key is a list of the temperatures (`float`) observed in the respective city over a certain period of time. This function calculates the amplitude of temperature variation (the difference between the maximum and minimum temperature) for each city and returns a *dictionary* with the name of the city as **key** and the amplitude observed in each respective time series as **value**.
   Hint: temperatures are measured in Celsius degree and this function admits temperatures from -100C to +100C. Use only 1-digit decimals.
   Validations:
   1.) the dictionary must have at least one entry (key:value,
   example entry: `"city":[12.8, 13.5, 25.0, 20.0, 22.0]`)
   2.) the key must be a `str`
   3.) the value associated with a key must be a non-empty list of `float`
   Error message: `err_msg ="Invalid argument. Check specifications."`

5. Implement the function `def count_evens(list_of_ints):`

   This function takes as argument a list of `int` numbers and returns the count of even numbers in the list.

   Validations:
   1.) The list must be non empty
   2.) All elements of the list must be of type `int`
   Error message: `err_msg ="Invalid argument. Check specifications."`
   Tip1: Remember that you can use the function implemented in exercise 1.2.6 to check if all elements in a list have a certain required type.
   Tip2: the % ("mod") operator computes the remainder, e.g. 5 % 2 is 1.

6. Implement the function `def centered_average(list_of_ints):`

   This function takes as argument a list of `int` numbers and returns the mean average of the values, ignoring the highest and lowest in the list. This function parts from the assumption, without validation, that the items in the list are in the range -9999999999 to +9999999999.

   Validation:
   The list must be non empty. Once it is guaranteed that the list is non-empty, remove the elements that have the highest or the lowest values of this list. The resulting list must have length `>= 2`.
   Error message: `err_msg ="Invalid argument. Check specifications."`

7. Implement the function `def has22(list_of_ints):`

   This function takes as argument a list of `int` numbers and returns true if the list contains at least one element with value 2 followed by another element with value 2.
   Examples:
   `has22([1, 35, 2, 2, 1])`, expected result: `True`
   `has22([1, 35, 2, 1, 2])`, expected result: `False`
   `has22([1, 5, 9, 1, 2, 1, 7, 6, 2, 2, 98])`, expected result: `True`

   No validations required. If the list is empty, the function must return `False`.

8. Implement the function `def sum_of_diff(list_of_numbers):`

   This function takes a list of **positive** numbers as argument and returns the sum of the difference between all pairs of consecutive elements. For instance, if the list is `[1, 5, 7, 3]`, then the expected result is (calculated as `5-1 + 3-7`). If the list has an odd number of elements, ignore the last element.

   No validations required. If the list is empty, the function must return 0.

(final)