Python Exercises - Part III

Python and R for Data Science

Data Science and Management

Exercise 1: shortest words

Write a function `shortest_words` that:

- takes a list of words
- return a list containing the shortest words in the list received as argument. The list will containt more than one word when there multiple words with the same length.

Examples:

- `shortest_words([])` returns `[]`
- `shortest_words(['sheldon', 'cooper'])` returns `['cooper']`
- `shortest_words(['sheldon', 'cooper', 'howard'])` returns `['cooper', 'howard']`

NOTE: do not use any buit-in function from Python to solve the exercise

In [2105]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2107]:
```
try: assert shortest_words([]) == [] and not print("Test #1 passed")
except: print('Test #1 failed')
try: assert sorted(shortest_words(['sheldon', 'cooper'])) == ['cooper'] and not p
except: print('Test #2 failed')
try: assert sorted(shortest_words(['sheldon', 'cooper', 'howard'])) == ['cooper',
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 2: multiply tuples

Write a function called `mul_tuple` that:

- Takes two tuples of equal length containing integers as arguments
- Returns a new tuple containing the products of the corresponding elements (at the same position) of the two tuples

If the two tuples have different lengths, the function should return `None`.

In [2108]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2110]:
```
try: assert mul_tuple((1, 2, 3), (4, 5, 6, 7)) == None and not print("Test #1 pas
except: print('Test #1 failed')

try: assert mul_tuple((1, 2), (4, 5)) == (4, 10) and not print("Test #2 passed")
except: print('Test #2 failed')
```

```
Test #1 passed
Test #2 passed
```

Exercise 3: max distance

Write a function `max_dist_point` that:

- Takes as arguments:
    - A point in the Cartesian plane represented as a tuple with its coordinates (x, y), where x and y are integers.
    - A list of points in the Cartesian plane.
- Returns:
    - If the list received as an argument is empty: `None`.
    - Otherwise: a tuple of two values, consisting of:
        1. The maximum distance (integer) between the given point and all the points in the list. To calculate the distance between a pair of points ((x1, y1)) and ((x2, y2)), use the Euclidean distance formula:

$$distance = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

        2. The point from the list that produced the maximum distance. Round the distance down using `int(distance)`.

NOTE: The square root can be calculated using the function `math.sqrt()` from the math library.

In [2111]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2113]:
```
try: assert max_dist_point((0, 0), []) == None and not print("Test #1 passed")
except: print('Test #1 failed')

try: assert max_dist_point((0, 0), [(1, 1), (2, 2), (3, 3)]) == (4, (3, 3)) and n
except: print('Test #2 failed')

try: assert max_dist_point((10, 12), [(1, 3), (4, 23), (-100, 0), (1, 1)]) and no
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 4: Character Position Tracker

Write a function `track_char_positions` that:

- Takes a string `text` as input.
- Returns a dictionary where:
    - The keys are the unique characters in the string.
    - The values are lists of positions (indices) where each character appears in the string.

Example:

```
text = "hello"
result = track_char_positions(text)
```

The `result` should be:

```
{
  'h': [0],
  'e': [1],
  'l': [2, 3],
  'o': [4]
}
```

NOTE: The function should track both uppercase and lowercase characters as distinct. NOTE: Spaces and punctuation should also be tracked as characters.

In [2114]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2116]:
```
text = "hello"
expected_result = {'h': [0], 'e': [1], 'l': [2, 3], 'o': [4]}
try: assert track_char_positions(text) == expected_result and not print("Test #1
except: print('Test #1 failed')

text = "banana"
expected_result = {'b': [0], 'a': [1, 3, 5], 'n': [2, 4]}
try: assert track_char_positions(text) == expected_result and not print("Test #2
except: print('Test #2 failed')

# Test Case 3: Sentence with spaces and punctuation
text = "Hi, there !"
expected_result = {
    'H': [0], 'i': [1], ',': [2], ' ': [3, 9], 't': [4], 'h': [5], 'e': [6, 8], '
}
try: assert track_char_positions(text) == expected_result and not print("Test #3
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 5: Anagram Grouping

Write a function `group_anagrams` that:

- Takes a list of strings `words` as input.
- Returns a dictionary where:
    - The keys are sorted strings (alphabetically).
    - The values are lists of words from the input list that are anagrams of each other.

An anagram is a word formed by rearranging the letters of another word, using all the original letters exactly once.

Example:

```
words = ["listen", "silent", "enlist", "hello", "world", "drown", "word"]
result = group_anagrams(words)
```

The `result` should be:

```
{
  'eilnst': ['listen', 'silent', 'enlist'],
  'ehllo': ['hello'],
  'dlorw': ['world'],
```

```
  'dnorw': ['drown', 'word']
}
```

NOTE: The words should be grouped based on their sorted letter order. NOTE: If no anagram pairs are found, each word should still appear in its own list.

In [2117]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2119]:
```
words = ["listen", "silent", "enlist", "hello"]
expected_result = {
    'eilnst': ['listen', 'silent', 'enlist'],
    'ehllo': ['hello']
}
try: assert group_anagrams(words) == expected_result and not print("Test #1 passe
except: print('Test #1 failed')

words = ["apple", "banana", "orange"]
expected_result = {
    'aelpp': ['apple'],
    'aaabnn': ['banana'],
    'aegnor': ['orange']
}
try: assert group_anagrams(words) == expected_result and not print("Test #2 passe
except: print('Test #2 failed')

words = ["Listen", "Silent", "enlist"]
expected_result ={'Leinst': ['Listen'], 'Seilnt': ['Silent'], 'eilnst': ['enlist'
try: assert group_anagrams(words) == expected_result and not print("Test #3 passe
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
```

```
Test #3 passed
```

Exercise 6: ISBN Validator

Write a function `validate_isbn` that:

- Takes a string `isbn` as input, representing a 10-digit ISBN number.
- Returns a dictionary containing:
    - `valid`: A boolean indicating whether the ISBN is valid.
    - `digits`: A list of the individual digits in the ISBN.

An ISBN is considered valid if it meets the following criteria:

1. It consists of exactly 10 characters, where the first 9 are digits (0-9), and the last character can be a digit or an 'X' (which represents the number 10).
2. The ISBN is valid if the weighted sum of the digits (where the weight decreases from 10 to 1) is divisible by 11. For example:
    - For ISBN `0-306-40615-2`, the calculation would be: [(0 \times 10) + (3 \times 9) + (0 \times 8) + (6 \times 7) + (4 \times 6) + (0 \times 5) + (6 \times 4) + (1 \times 3) + (5 \times 2) + (2 \times 1) = 0 + 27 + 0 + 42 + 24 + 0 + 24 + 3 + 10 + 2 = 132] Since (132 \mod 11 = 0), it is valid.

Example:

```
isbn = "0306406152"
result = validate_isbn(isbn)
```

The `result` should be:

```
{
    'valid': True,
    'digits': ['0', '3', '0', '6', '4', '0', '6', '1', '5', '2']
}
```

NOTE: If the input is not a valid ISBN (e.g., it contains non-digit characters or is of the wrong length), return `{'valid': False, 'digits': []}`. NOTE: Ensure to treat 'X' as a digit representing 10.

In [2120]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2122]:
```
isbn = "0306406152"
expected_result = {'valid': True, 'digits': ['0', '3', '0', '6', '4', '0', '6', '
try: assert validate_isbn(isbn) == expected_result and not print("Test #1 passed"
except: print('Test #1 failed')

isbn = "123456789X"
expected_result = {'valid': True, 'digits': ['1', '2', '3', '4', '5', '6', '7', '
try: assert validate_isbn(isbn) == expected_result and not print("Test #2 passed"
except: print('Test #2 failed')

isbn = "12345678"
expected_result = {'valid': False, 'digits': []}
try: assert validate_isbn(isbn) == expected_result and not print("Test #3 passed"
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 7: Acronym Generator

Write a function `generate_acronym` that:

- Takes a string `phrase` as input, representing a multi-word phrase (e.g., "As Soon As Possible").
- Returns a dictionary where:
    - The key is the acronym formed from the first letter of each word in the phrase (case insensitive).
    - The value is the original phrase with each word capitalized.

Example:

```
phrase = "as soon as possible"
result = generate_acronym(phrase)
```

The `result` should be:

```
{
    'ASAP': 'As Soon As Possible'
}
```

NOTE: Ignore any non-alphabetic characters when forming the acronym. NOTE: The acronym should be in uppercase. NOTE: If the input string is empty, return `{'acronym': '',`

`'phrase': ''}`.

In [2123]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2125]:
```
phrase = "as soon as possible"
expected_result = {'acronym': 'ASAP', 'phrase': 'As Soon As Possible'}
try: assert generate_acronym(phrase) == expected_result and not print("Test #1 pa
except: print('Test #1 failed')

phrase = "   keep it simple stupid  "
expected_result = {'acronym': 'KISS', 'phrase': 'Keep It Simple Stupid'}
try: assert generate_acronym(phrase) == expected_result and not print("Test #2 pa
except: print('Test #2 failed')

phrase = "for your information."
expected_result = {'acronym': 'FYI', 'phrase': 'For Your Information.'}
try: assert generate_acronym(phrase) == expected_result and not print("Test #3 pa
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 8: Movie Rating Organizer

Write a function `organize_movie_ratings` that:

- Takes a list of tuples `ratings` as input, where each tuple contains two elements:
    - A string `movie` representing the name of a movie.
    - An integer `rating` representing the rating of that movie (from 1 to 10).
- Returns a dictionary where:
    - The keys are the unique movie titles.
    - The values are lists of ratings for each movie.

Example:

```
ratings = [
    ("Inception", 9),
    ("The Matrix", 8),
    ("Inception", 10),
    ("The Godfather", 9),
    ("The Matrix", 9)
]
result = organize_movie_ratings(ratings)
```

The `result` should be:

```
{
    'Inception': [9, 10],
    'The Matrix': [8, 9],
    'The Godfather': [9]
}
```

NOTE: If a movie appears multiple times in the input list, all ratings should be included in the list for that movie. NOTE: The order of the ratings in the lists should reflect the order they appear in the input list.

In [2126]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2128]:

```
ratings = [
    ("Inception", 9),
    ("The Matrix", 8),
    ("Inception", 10),
    ("The Godfather", 9),
    ("The Matrix", 9)
]
expected_result = {
    'Inception': [9, 10],
    'The Matrix': [8, 9],
    'The Godfather': [9]
}
try: organize_movie_ratings(ratings) == expected_result and not print("Test #2 pa
except: print('Test #2 failed')

ratings = [
    ("Titanic", 7),
    ("Titanic", 7),
    ("Titanic", 7)
]
expected_result = {
    'Titanic': [7, 7, 7]
}
try: organize_movie_ratings(ratings) == expected_result and not print("Test #1 pa
```

```
except: print('Test #1 failed')

ratings = [
    ("Avatar", 8),
    ("Avatar", 9),
    ("Avatar", 10)
]
expected_result = {
    'Avatar': [8, 9, 10]
}
try: organize_movie_ratings(ratings) == expected_result and not print("Test #3 pa
except: print('Test #3 failed')
```

```
Test #2 passed
Test #1 passed
Test #3 passed
```

Exercise 9: Contact Book

Write a function `create_contact_book` that:

- Takes a list of tuples `contacts` as input, where each tuple contains two elements:
    - A string `name` representing the name of a contact.
    - A string `phone_number` representing the contact's phone number.
- Returns a dictionary where:
    - The keys are the unique names of the contacts (case insensitive).
    - The values are the corresponding phone numbers.

Example:

```
contacts = [
    ("Alice", "123-456-7890"),
    ("Bob", "987-654-3210"),
    ("alice", "555-555-5555"),
    ("Charlie", "111-222-3333")
]
result = create_contact_book(contacts)
```

The `result` should be:

```
{
    'alice': '555-555-5555',
    'bob': '987-654-3210',
    'charlie': '111-222-3333'
}
```

NOTE: If a contact appears multiple times in the input list, the last occurrence should be kept in the dictionary.
NOTE: The names in the dictionary should be in lowercase to maintain case insensitivity.

In [2129]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2131]:

```
contacts = [
    ("Alice", "123-456-7890"),
    ("Bob", "987-654-3210"),
    ("alice", "555-555-5555"),
    ("Charlie", "111-222-3333")
]
expected_result = {
    'alice': '555-555-5555',
    'bob': '987-654-3210',
    'charlie': '111-222-3333'
}
try: assert create_contact_book(contacts) == expected_result and not print("Test
except: print('Test #1 failed')

contacts = [
    ("John", "555-123-4567"),
    ("john", "555-765-4321"),
    ("Doe", "555-987-6543")
]
expected_result = {
    'john': '555-765-4321',
    'doe': '555-987-6543'
}
try: assert create_contact_book(contacts) == expected_result and not print("Test
```

```
except: print('Test #2 failed')

# Test Case 3: Only one contact
contacts = [
    ("Alice", "123-456-7890")
]
expected_result = {
    'alice': '123-456-7890'
}
try: assert create_contact_book(contacts) == expected_result and not print("Test
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 10: Library Management System

Write a function `manage_library` that:

- Takes a list of tuples `library_updates` as input, where each tuple contains:
    - A string `book_title` representing the title of the book.
    - An integer `quantity` representing the number of copies to be added to or removed from the library. Note: If `quantity` is negative, it means that books are being removed from the library.
- Returns a dictionary representing the current inventory of the library where:
    - The keys are unique book titles (case insensitive).
    - The values are dictionaries containing:
        - `total_copies`: the total number of copies of the book available in the library (should not go below zero).
        - `available_copies`: the number of copies currently available for borrowing (initially equal to `total_copies`).

Example:

```
library_updates = [
    ("The Great Gatsby", 5),
    ("1984", 10),
    ("the great gatsby", 2),
    ("1984", -3),
    ("To Kill a Mockingbird", 7),
    ("1984", -8),
    ("Moby Dick", -2)
]
result = manage_library(library_updates)
```

The `result` should be:

```
{
    'the great gatsby': {
        'total_copies': 7,
        'available_copies': 7
    },
    '1984': {
        'total_copies': 2,
        'available_copies': 2
    },
    'to kill a mockingbird': {
        'total_copies': 7,
        'available_copies': 7
    },
    'moby dick': {
        'total_copies': 0,
        'available_copies': 0
```

```
    }
}
```

NOTE:

- If the quantity for a book goes below zero, it should not be removed from the inventory; instead, it should be set to zero for both `total_copies` and `available_copies`.
- The function should maintain case insensitivity for book titles (e.g., "The Great Gatsby" and "the great gatsby" should be treated as the same book).

In [2132]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2134]:

```
library_updates = [
    ("The Great Gatsby", 5),
    ("1984", 10),
    ("the great gatsby", 2),
    ("1984", -3),
    ("To Kill a Mockingbird", 7),
    ("1984", -8),
    ("Moby Dick", -2)
]
expected_result = {
    'the great gatsby': {
        'total_copies': 7,
        'available_copies': 7
    },
    '1984': {
        'total_copies': 0,
        'available_copies': 0
    },
    'to kill a mockingbird': {
        'total_copies': 7,
        'available_copies': 7
    },
    'moby dick': {
        'total_copies': 0,
```

```
        'available_copies': 0
    }
}
try: assert manage_library(library_updates) == expected_result and not print("Tes
except: print('Test #1 failed')

library_updates = [
    ("The Catcher in the Rye", 5),
    ("The Catcher in the Rye", -5),
    ("Brave New World", 10),
    ("Brave New World", -10)
]
expected_result = {
    'the catcher in the rye': {
        'total_copies': 0,
        'available_copies': 0
    },
    'brave new world': {
        'total_copies': 0,
        'available_copies': 0
    }
}
try: assert manage_library(library_updates) == expected_result and not print("Tes
except: print('Test #2 failed')

library_updates = []
expected_result = {}
try: assert manage_library(library_updates) == expected_result and not print("Tes
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 11: Social Media Connections

Write a function `manage_connections` that:

- Takes a list of tuples `connections` as input, where each tuple contains:
    - A string `user` representing the username.
    - A set of strings `friends` representing the usernames of friends that the user is connected to.
- The function should return a dictionary representing each user and their unique connections (friends) where:
    - The keys are unique usernames (case insensitive).
    - The values are sets of unique friends for that user.

Example:

```
connections = [
    ("Alice", {"Bob", "Charlie"}),
    ("Bob", {"Alice", "David"}),
    ("alice", {"Eve"}),
    ("Charlie", {"Bob"}),
    ("david", {"Alice", "Eve"}),
```

```
    ("Eve", set())
]
result = manage_connections(connections)
```

The `result` should be:

```
{
    'alice': {"bob", "charlie", "eve"},
    'bob': {"alice", "david"},
    'charlie': {"bob"},
    'david': {"alice", "eve"},
    'eve': set()
}
```

NOTE:

- If a user has multiple connections with the same friend, those should only be counted once.
- The function should maintain case insensitivity for usernames (e.g., "Alice" and "alice" should be treated as the same user).
- If a user has no friends, their value in the dictionary should be an empty set.

In [2135]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2137]:

```
connections = [
    ("Alice", {"Bob", "Charlie"}),
    ("Bob", {"Alice", "David"}),
    ("alice", {"Eve"}),
    ("Charlie", {"Bob"}),
    ("david", {"Alice", "Eve"}),
    ("Eve", set())
]
expected_result = {
    'alice': {"bob", "charlie", "eve"},
    'bob': {"alice", "david"},
    'charlie': {"bob"},
    'david': {"alice", "eve"},
    'eve': set()
}
try: assert manage_connections(connections) == expected_result and not print("Tes
except: print('Test #1 failed')

connections = [
    ("John", set()),
    ("Doe", set())
]
expected_result = {
    'john': set(),
```

```
    'doe': set()
}
try: assert manage_connections(connections) == expected_result and not print("Tes
except: print('Test #2 failed')

connections = [
    ("Alice", {"Bob", "Charlie"}),
    ("Alice", {"Bob", "Eve"}),
    ("bob", {"Alice"}),
    ("charlie", {"Alice"}),
]
expected_result = {
    'alice': {"bob", "charlie", "eve"},
    'bob': {"alice"},
    'charlie': {"alice"},
}
try: assert manage_connections(connections) == expected_result and not print("Tes
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
Test #3 passed
```

Exercise 12: Company Employee Records

Write a function `manage_employees` that:

- Takes a list of tuples `employee_updates` as input, where each tuple contains:
    - A string `department` representing the name of the department (e.g., "HR", "Engineering").
    - A string `employee_name` representing the name of the employee.
    - An integer `salary` representing the employee's salary (can be negative to indicate salary reductions).
- The function should return a dictionary representing each department's employees where:
    - The keys are unique department names (case insensitive).
    - The values are dictionaries containing:
        - `employees`: a dictionary of employee names (case insensitive) and their current salaries.
        - `average_salary`: the average salary of employees in that department, rounded to two decimal places.

Example:

```
employee_updates = [
    ("HR", "Alice", 50000),
    ("Engineering", "Bob", 70000),
    ("HR", "Alice", -5000),
    ("Engineering", "Charlie", 60000),
    ("HR", "Dave", 55000),
    ("engineering", "Alice", -10000),  # Salary reduction for Alice
    ("Engineering", "Charlie", -10000), # Salary reduction for Charlie
    ("HR", "Eve", 45000)
]
result = manage_employees(employee_updates)
```

The `result` should be:

```
{
    'hr': {
        'employees': {
            'alice': 45000,
            'dave': 55000,
            'eve': 45000
        },
        'average_salary': 45000.0
    },
    'engineering': {
        'employees': {
            'bob': 70000,
            'charlie': 50000
```

```
        },
        'average_salary': 60000.0
    }
}
```

NOTE:

- If an employee appears multiple times in the updates for the same department, update its salary condering the value as an increment or a reduction.
- If a salary goes below zero after an update, set it to zero.
- The function should maintain case insensitivity for department names and employee names.
- If the input list is empty, return an empty dictionary.

In [2138]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2140]:

```
employee_updates = [
    ("HR", "Alice", 50000),
    ("Engineering", "Bob", 70000),
    ("HR", "Alice", -5000),
    ("Engineering", "Charlie", 60000),
    ("HR", "Dave", 55000),
    ("Engineering", "Charlie", -10000), # Salary reduction for Charlie
    ("HR", "Eve", 45000)
]
expected_result = {
    'hr': {
        'employees': {
            'alice': 45000,
            'dave': 55000,
            'eve': 45000
        },
        'average_salary': 48333.33
    },
    'engineering': {
        'employees': {
            'bob': 70000,
            'charlie': 50000
        },
        'average_salary': 60000.0
```

```
    }
}
try: assert manage_employees(employee_updates) == expected_result and not print("
except: print('Test #1 failed')

employee_updates = [
    ("HR", "Alice", 30000),
    ("HR", "Alice", -15000),  # Reduction
    ("HR", "Bob", 20000),
    ("Engineering", "Charlie", 100000),
    ("Engineering", "Charlie", -20000),
    ("HR", "Alice", -20000),  # Reduction to zero
]
expected_result = {
    'hr': {
        'employees': {
            'alice': 0,
            'bob': 20000
        },
        'average_salary': 10000.0
    },
    'engineering': {
        'employees': {
            'charlie': 80000
        },
        'average_salary': 80000.0
    }
}
try: assert manage_employees(employee_updates) == expected_result and not print("
except: print('Test #2 failed')
```

```
employee_updates = [
    ("HR", "Alice", 30000),
    ("HR", "Alice", 25000),
    ("Engineering", "Bob", 50000),
    ("Engineering", "Bob", 10000),
    ("Engineering", "Bob", -20000),
]
expected_result = {
    'hr': {
        'employees': {
            'alice': 55000
        },
        'average_salary': 55000
    },
    'engineering': {
        'employees': {
            'bob': 40000
        },
        'average_salary': 40000.0
    }
}
try: assert manage_employees(employee_updates) == expected_result and not print("
except: print('Test #3 failed')
```

```
Test #1 passed
Test #2 passed
{'hr': {'employees': {'alice': 55000}, 'average_salary': 55000.0}, 'engin
eering': {'employees': {'bob': 40000}, 'average_salary': 40000.0}}
{'hr': {'employees': {'alice': 55000}, 'average_salary': 55000}, 'enginee
```

```
ring': {'employees': {'bob': 40000}, 'average_salary': 40000.0}}
Test #3 passed
```

Exercise 13: sum of the first n numbers

1. Define the integer `n` equal to `100`
2. Using a loop compute the sum of the first `n` numbers (starting from `1`), storing the result into `s`
3. print `s`

In [2141]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2143]:
```
try: assert s == 5050 and not print("Test passed")
except: print('Test failed')
```

```
Test passed
```

Exercise 14: sum of the prime numbers

1. Define the integer `n` equal to `100`
2. Using a loop compute the sum of prime numbers up to `n`, storing the result into `s`
3. print `s`

In [2144]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2146]:
```
try: assert s == 1060 and not print("Test passed")
except: print('Test failed')
```

```
Test passed
```

Exercise 15: prefixes of a string

1. Define the string `s` equal to `The Big Bang Theory`
2. Create the empty list `p`
3. Using a loop, add all prefixes of `s` to `p` (note: `The Big Bang Theory` is a prefix of `The Big Bang Theory`)
4. print `p`

In [2147]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2149]:
```
try: assert sorted(p) == ['T', 'Th', 'The', 'The ', 'The B', 'The Bi', 'The Big',
except: print('Test failed')
```

```
Test passed
```

Exercise 16: check postfixes of a string

1. Define the string `s` equal to `The Big Bang Theory`
2. Define the list `p` equal to `["y", "ry", "ery", ""]`
3. Remove from `p` any string that is not a postfix of `s` (note: `""` is a postfix of `s`)
4. print `p`

In [2150]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2152]:
```
try: assert p == ['y', 'ry', ''] and not print("Test passed")
except: print('Test failed')
```

```
Test passed
```

Exercise 17: max of a list

Define a function `max_from_list` that:

- takes as arguments a list of integers
- returns:
    - if the list is not empty: the maximum value in the list
    - otherwise: `None`

Do not use the built-in function `max` in this exercise.

In [2153]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2155]:
```
try: max_from_list([]) == None and max_from_list([1, 2, 3]) == 3 and not print("T
except: print('Test failed')
```

```
Test passed
```

Exercise 19: prime numbers

Define a function `is_prime` that:

- takes as arguments a list `L` of positive integers
- returns:
    - if the list is not empty: a new list where the i-th element is a boolean asserting whether the i-th element from `L` is a prime number
    - otherwise: `[]`

In [2156]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2158]:
```
try: assert is_prime([]) == [] and is_prime([3, 4, 9, 11]) == [True, False, False
except: print('Test failed')
```

```
Test passed
```

Exercise 20: word frequency

Define a function `count_freq` that:

- takes as arguments:
  - a string `s`
  - a list `L` of words
- returns:
  - if the list is not empty: a new list where the i-th element is the number of occurences in `s` of the i-th word from `L`
  - otherwise: `[]`

In [2159]:
```
# Solution goes here
```

Test your code

Run this code to test your solution:

In [2161]:
```
try: assert count_freq("test", []) == [] and count_freq("Aejeje", ["e", "je", "aj
except: print('Test failed')
```

```
Test passed
```