Task 1

This task is about to use python `generators`, `modules`, `datetime`. Let's create 2 .py files:

The `generator file` must contain a `fake_user_generator` function that uses the power of a generator that creates a new user when the function is called.

`main.py(task1.py)`

- `User` class
- Import and use `generator` file which it will assign to the `User` class.

`User class`

- The `__init__` method must initialize `username`, `age`, `gender`, `created_at`
- The `age_group` method should return a group of ages based on their age: ** 0 to 12 years old - Child ** 13 to 17 years old - Adolescent ** 18 to 65 years old - Adult ** 65 and more years old - Old adult

`generators.py` **`fake_user_generator`**

- Infinite loop (add a limit as a protection mechanism to 100 users) which means you can use generator as many times as you want.
- The `username` format must contain `random_words`, from 0 to 1000 random numbers and `surnames`. (You can upgrade as you want)
- `Age` must be randomly selected from 0 to 100
- `Gender` must be `female` or `male`.
- `created_at` must be a date in the following format: `mm/dd/yy`

```
# generators.py
random_words = ['liam', 'snake', 'pool', 'ggg', 'hurray', 'yo', 'rock',
'football', 'basket', 'ice_cream', 'bing', 'chilling']
surnames = ['smith', 'black', 'white', 'johnson', 'williams', 'jones', 'millers',
'wilson', 'anderson', 'holmes', 'moore']

def fake_user_generator():
    # Main code ...
    pass
```

```
# main.py (task1.py)
class User:
    def __init__():
        pass

    def age_group():
        pass


# Printing area
```

```
user_generator = fake_user_generator()
for _ in range(10):
    new_user = next(user_generator)

    print('User:', new_user)
    user = User(new_user['username'], new_user['age'], new_user['gender'],
new_user['created_at'])
    print(f'{user.username} age group is', user.age_group(), "\n")
```

Sample Input: Generated user object(dictionary)

Sample Output: User: {'username': 'yo460johnson', 'age': 84, 'gender': 'male', 'created_at': '02/15/23'}
yo460johnson age group is Old adult

Task 2

Task 3

Imagine that you are data engineer. You have a task to separate group of users into two group for AB testing. The structure of the user object as the following:

```
{
  "name": "Name Lastname",
  "birthdate": "1997-01-13",
  "height": 180
}
```

The main requirements for separation are

- users in each group must be 40 y.o. or younger
- users should be equally distributed by their height

To evaluate equality of distribution, print size and average height of each group

```
from sample_inputs import users

def ab_groups(users, max_age):
    ...

def average_height(users):
    ...

max_age = 40
a_group, b_group = ab_groups(users, max_age)
print(len(a_group), len(b_group))
print(average_height(a_group), average_height(b_group))
```

Task 4

Task 5

Create an iterator class `CycleIterator` which can be used to cycle over an iterable object:

Attributes:
- `data` - data that we will iterate through (string value)
- `max_times` - maximum values that we have to extract from `data` (int value)
- `iterable_obj` - iterable object (int value)

Methods:
- `__init__` - initialization method
- `__next__` - must return the next item in the `data`. `iterable_obj` will be incremented by 1. Raise `StopIteration` if `iterable_obj >= max_times`.

Use `CycleIterator` as an iterator in `Cycle` class:

```
class Cycle():

    def __init__(self, data, max_times):
        self.data = data
        self.max_times = max_times

    def __iter__(self):
        return CycleIterator(self.data, self.max_times)
```

Sample Input: Cycle('abc', 5)

Sample Output: ['b', 'c', 'a', 'b']

Sample Input: Cycle('string', 9)

Sample Output: ['s', 't', 'r', 'i', 'n', 'g', 's', 't', 'r']

Task 6

Task 7

Write a Python class called `FibonacciIterator` that implements an iterator that generates an infinite sequence of Fibonacci numbers. Then, write a function called `get_fibonacci_numbers` that takes an integer n as input and returns the first n Fibonacci numbers as a list using the `FibonacciIterator`.

Methods:
- `__init__` - initializes the starting values of the sequence to 0 and 1.
- `__iter__` - method that returns `itself` as an `iterator`.
- `__next__` - method that generates the next Fibonacci number in the sequence and updates the internal state of the iterator.

The `get_fibonacci_numbers` function takes an integer `n` as input and returns the first n Fibonacci numbers as a list. It starts by creating an iterator object using `FibonacciIterator()`.

```
class FibonacciIterator:
    def __init__(self):
        self.a = 0
        self.b = 1

    def __iter__():


    def __next__():


def get_fibonacci_numbers(n):
```

`example` - get_fibonacci_numbers(10) return - [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Task 8

Task 9

Write a Python function that takes a JSON string representing a list of points in two-dimensional space and performs the following tasks:

1. Calculate the distance between the two closest points in the list.
2. Calculate the sum of the distances from each point in the list to the origin (0, 0).
3. Calculate the average angle between each pair of adjacent points in the list.

The JSON string will have the following format:

```
[
    {"x": x1, "y": y1},
    {"x": x2, "y": y2},
    ...,
    {"x": xn, "y": yn}
]
```

where `xi` and `yi` are integers representing the coordinates of the i-th point. The list may contain any number of points greater than or equal to 2.

You can assume that the JSON string is valid and that all points have integer coordinates.

To implement these tasks, you can use the following functions from the math module:

- `sqrt`: computes the square root of a number
- `atan2`: computes the arctangent of the quotient of its arguments, returning the angle in radians
- `degrees`: converts an angle in radians to degrees

Your function should return a JSON object with the following keys and values:

```
{
    "closest_pair_distance": closest_pair_distance,
    "sum_of_distances_to_origin": sum_of_distances_to_origin,
    "average_angle_between_points": average_angle_between_points
}
```

where `closest_pair_distance` is a float representing the distance between the two closest points in the list, `sum_of_distances_to_origin` is a float representing the sum of the distances from each point in the list to the origin, and `average_angle_between_points` is a float representing the average angle between each pair of adjacent points in the list.

For example, if the input JSON string is `'[{"x":0, "y":0}, {"x":1, "y":1}, {"x":2, "y":2}, {"x":3, "y":3}]'`, the output JSON object should be:

```
{
    "closest_pair_distance": 1.4142135623730951,
    "sum_of_distances_to_origin": 5.656854249492381,
    "average_angle_between_points": 45.0
}
```

Note: You will need to import the `json` module at the beginning of your program using the statement `import json` to work with the JSON input.

Task 10

Task 11

In this task, you should create an iteratable class `Example` in `module.py`.
This class should iterate triangle numbers `0, 1, 3, 6, ...`. In this class there are two another classes `Inner_iter` and `Reverse_iter`.
`Inner_iter` - should receive an integer number n for initialization and iterate triangle numbers.
`Reverse_iter` - should receive another iterator and iterate it in reverse order.

Also in `Example` there are another methods:

`integers()` - return infinite sequence of integers
`take(n, iterator)` - return list of n first values from iterator
`get_generator()` - return infinite sequence of tuples of legs of a right triangle
`to_day(n, month)` - accept n - integer number, month - month
return day of month,

- if month is Feb - same day
- if 30 days in month - day before
- if 31 days in month - day after

`to_month(n)` - accept n - integer number return month as integer (1..12)
`to_year(n)` - accept n - integer number return year as 2000 + n

`conver_to_dates(arr)` - accept arr - list of tuple with 3 numbers (1st param - day, 2nd - month, 3rd - year) return list of datetime
`date_in_format(date, format)` - accept date - datetime, format - integer number (default=1) return date as string in format:

- 0 - dd/MM/yyyy
- 1 - yyyy-MM-dd
- 2 - dd Month
- otherwise - print `Unsupported format` and return None

```
class Example:
    def __init__(self, n):
        pass

    def __iter__(self):
        pass

    class Inner_iter:
        def __init__(self, n):
            pass

        def __iter__(self):
            pass

        def __next__(self):
            pass


    class Reverse_iter:
        def __init__(self, iter):
            pass

        def __iter__(self):
            pass

        def __next__(self):
            pass


    def integers():
        pass

    def take(n, iterator):
        pass

    def get_generator():
        pass

    def to_day(n, month):
        pass

    def to_mon(n):
```

```
        pass

    def to_year(n):
        pass

    def conver_to_dates(arr):
        pass

    def date_in_format(date, format):
        pass
```

Example

```
params = Example.get_generator()
list_of_params = Example.take(5, params)
converted = Example.conver_to_dates(list_of_params)

for i in range(len(converted)):
    print(Example.date_in_format(converted[i], i % 4))

iter = Example(5)
it = Example.Reverse_iter(iter)
print(list(iter))
print(list(it))
```

Output

unsupported format
None
07/08/2010
2013-12-06
10 December, 2015
unsupported format
None
[0, 1, 3, 6, 10]
[10, 6, 3, 1, 0]

In another python file `task11.py` use `Example` from `module.py`

Task 12

Task 13

Find the roots of a quadratic equation

Write a Python function find_roots(input_dict) that takes a dictionary input_dict as argument and returns a tuple containing the roots of the corresponding quadratic equation in the form of (x1, x2). If the equation has no real roots, the function should return None.

The dictionary input_dict will contain the following keys and values:

- "a" (float) - the coefficient of x^2 in the quadratic equation
- "b" (float) - the coefficient of x in the quadratic equation
- "c" (float) - the constant term in the quadratic equation

The quadratic equation is of the form ax^2 + bx + c = 0, where a, b, and c are coefficients.

To solve the equation, you can use the quadratic formula:

x1 = (-b + sqrt(b^2 - 4ac)) / 2a x2 = (-b - sqrt(b^2 - 4ac)) / 2a

You can use the `math.sqrt()` function to calculate the square root.

Your function should handle the following cases:

- If the discriminant `b^2 - 4ac` is negative, the equation has no real roots and the function should return `None`.
- If the discriminant is zero, the equation has a single root, which is `x = -b / 2a`.
- If the discriminant is positive, the equation has two roots, which can be calculated using the quadratic formula.

For example, calling `find_roots({"a": 1, "b": -5, "c": 6})` should return `(3.0, 2.0)`, and calling `find_roots({"a": 1, "b": 2, "c": 1})` should return `(-1.0, -1.0)`.

Task 14

Write a Python function `count_words(input_file_path, output_file_path)` that takes as input the path to a JSON file input_file_path and the path to an output file output_file_path. The input JSON file contains a list of strings, where each string represents a sentence. The function should count the occurrences of each word in the sentences and save the results as a dictionary to the output file. The keys of the dictionary should be the words and the values should be the number of times the word occurs in the sentences.

The output file should be a `JSON file` containing the dictionary.

For example, suppose the input `JSON file` contains the following sentences:

json ["The quick brown fox jumps over the lazy dog.", "The quick brown fox jumps over the lazy dog again."]

The function should create a dictionary with the following key-value pairs:

```
{ "The": 2, "quick": 2, "brown": 2, "fox": 2, "jumps": 2, "over": 2, "the": 2, "lazy": 2, "dog.": 1, "again.": 1 }
```

And the output JSON file should contain this dictionary.

Note that the function should be case-insensitive, so that "The" and "the" are counted as the same word.

You can assume that the input file is well-formed JSON and that each string in the list is a valid sentence.

To implement this function, you can use the json module to read the input file and the collections module to create a Counter object, which can be used to count the occurrences of the words.

Example

count_words("sentences.json", "word_counts.json")

The function call above would read the sentences from a file called sentences.json, count the occurrences of each word, and save the results to a file called word_counts.json.