# Exercise: Iterators and Generators

Problems for exercise and homework for the [Python OOP Course @SoftUni](https://softuni.bg/trainings/3964/python-oop-february-2023).

Submit your solutions in the SoftUni judge system at <https://judge.softuni.org/Contests/1945>.

## Take Skip

Create a **class** called take\_skip. Upon initialization, it should receive a **step** (int) and a **count** (int). Implement the \_\_iter\_\_ and \_\_next\_\_ functions. The iterator should return the **count** numbers (**starting** **from 0**) with the **given step**. For more clarification, see the examples:

***Note: Submit only the class in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| numbers = take\_skip(2, 6)  for number in numbers:  print(number) | 0  2  4  6  8  10 |
| numbers = take\_skip(10, 5)  for number in numbers:  print(number) | 0  10  20  30  40 |

## Dictionary Iterator

Create a class called dictionary\_iter. Upon initialization, it should receive a **dictionary** object. Implement the iterator to return **each key-value pair** of the dictionary as a **tuple of two elements** (the key and the value).

***Note: Submit only the class in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| result = dictionary\_iter({1: "1", 2: "2"})  for x in result:  print(x) | (1, '1')  (2, '2') |
| result = dictionary\_iter({"name": "Peter", "age": 24})  for x in result:  print(x) | ("name", "Peter")  ("age", 24) |

## Countdown Iterator

Create a class called countdown\_iterator. Upon initialization, it should receive a **count**. Implement the **iterator** to return **each countdown number** (from **count** to **0** inclusive), separated by a single space.

***Note: Submit only the class in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| iterator = countdown\_iterator(10)  for item in iterator:  print(item, end=" ") | 10 9 8 7 6 5 4 3 2 1 0 |
| iterator = countdown\_iterator(0)  for item in iterator:  print(item, end=" ") | 0 |

## Sequence Repeat

Create a class called **sequence\_repeat** which should receive a **sequence** and a **number** upon initialization. Implement an **iterator** to return the given elements, so they form a string with a length - the given **number**. If the **number is greater** than the number of elements, then the **sequence repeats** as necessary. For more clarification, see the examples:

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| result = sequence\_repeat('abc', 5)  for item in result:  print(item, end ='') | abcab |
| result = sequence\_repeat('I Love Python', 3)  for item in result:  print(item, end ='') | I L |

## Take Halves

You are given a skeleton with the following code:

**def** solution():

**def** integers():

*#* ***TODO: Implement***

**def** halves():

**for** i **in** integers():

*#* ***TODO: Implement***

**def** take(n, seq):

*#* ***TODO: Implement***

**return** (take, halves, integers)

Implement the **three** generator functions:

* integers() - generates an **infinite** amount of **integers** (starting from **1**)
* halves() - generates the halves of those integers (each integer **/ 2**)
* take(n, seq) - takes the **first** **n** halves of those integers

***Note: Complete the functionality in the skeleton and submit it to the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| take = solution()[0]  halves = solution()[1]  print(take(5, halves())) | [0.5, 1.0, 1.5, 2.0, 2.5] |
| take = solution()[0]  halves = solution()[1]  print(take(0, halves())) | [] |

## Fibonacci Generator

Create a generator function called fibonacci() that generates the **Fibonacci numbers** infinitely. **The first two numbers** in the sequence are **always 0 and 1.** Each following Fibonacci number is created by the **sum** of the **current** number **with the previous one.**

***Note: Submit only the function in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| generator = fibonacci()  for i in range(5):  print(next(generator)) | 0  1  1  2  3 |
| generator = fibonacci()  for i in range(1):  print(next(generator)) | 0 |

## Reader

Create a generator function called read\_next() which should receive a **different number** of arguments (all iterable). On each iteration, the function should return each element from each sequence.

***Note: Submit only the function in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| for item in read\_next("string", (2,), {"d": 1, "i": 2, "c": 3, "t": 4}):  print(item, end='') | string2dict |
| for i in read\_next("Need", (2, 3), ["words", "."]):  print(i) | N  e  e  d  2  3  words  . |

## Prime Numbers

Create a generator function called get\_primes() which should receive a **list of integer numbers** and return a list containing only the **prime numbers** from the initial list. You can learn more about prime numbers from [here](https://www.mathopenref.com/prime-number.html):

***Note: Submit only the function in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| print(list(get\_primes([2, 4, 3, 5, 6, 9, 1, 0]))) | [2, 3, 5] |
| print(list(get\_primes([-2, 0, 0, 1, 1, 0]))) | [] |

## Possible permutations

Create a generator function called possible\_permutations() which should receive **a list** and return lists with all possible permutations between its elements.

***Note: Submit only the function in the judge system***

### Examples

|  |  |
| --- | --- |
| **Test Code** | **Output** |
| [print(n) for n in possible\_permutations([1, 2, 3])] | [1, 2, 3]  [1, 3, 2]  [2, 1, 3]  [2, 3, 1]  [3, 1, 2]  [3, 2, 1] |
| [print(n) for n in possible\_permutations([1])] | [1] |