# Exercise: Iterators and Generators

Problems for exercise and homework for the Python OOP Course @SoftUni.
Submit your solutions in the SoftUni judge system at https://judge.softuni.org/Contests/1945.

## 1. 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)`<br>`for number in numbers:`<br>`  print(number)` | `0`<br>`2`<br>`4`<br>`6`<br>`8`<br>`10` |
| `numbers = take_skip(10, 5)`<br>`for number in numbers:`<br>`  print(number)` | `0`<br>`10`<br>`20`<br>`30`<br>`40` |

## 2. 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"})`<br>`for x in result:`<br>`  print(x)` | `(1, '1')`<br>`(2, '2')` |
| `result = dictionary_iter({"name": "Peter", "age": 24})`<br>`for x in result:`<br>`  print(x)` | `("name", "Peter")`<br>`("age", 24)` |

## 3. 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)`<br>`for item in iterator:`<br>`    print(item, end=" ")` | `10 9 8 7 6 5 4 3 2 1 0` |
| `iterator = countdown_iterator(0)`<br>`for item in iterator:`<br>`    print(item, end=" ")` | `0` |

## 4. 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)`<br>`for item in result:`<br>`    print(item, end ='')` | `abcab` |
| `result = sequence_repeat('I Love Python', 3)`<br>`for item in result:`<br>`    print(item, end ='')` | `I L` |

## 5. 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]`<br>`halves = solution()[1]`<br>`print(take(5, halves()))` | `[0.5, 1.0, 1.5, 2.0, 2.5]` |
| `take = solution()[0]`<br>`halves = solution()[1]`<br>`print(take(0, halves()))` | `[]` |

# 6. 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()`<br>`for i in range(5):`<br>`  print(next(generator))` | `0`<br>`1`<br>`1`<br>`2`<br>`3` |
| `generator = fibonacci()`<br>`for i in range(1):`<br>`  print(next(generator))` | `0` |

# 7. 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}):`<br>`  print(item, end='')` | `string2dict` |

| | |
|---|---|
| `for i in read_next("Need", (2, 3), ["words", "."]):`<br>`    print(i)` | `N`<br>`e`<br>`e`<br>`d`<br>`2`<br>`3`<br>`words`<br>`.` |

## 8. 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 here:

***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])))` | `[]` |

## 9. 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]`<br>`[1, 3, 2]`<br>`[2, 1, 3]`<br>`[2, 3, 1]`<br>`[3, 1, 2]`<br>`[3, 2, 1]` |
| `[print(n) for n in possible_permutations([1])]` | `[1]` |