Typing and Type Hints Practice

1. Complete this table.
Answers to these questions are in the Python typing and collections.abc documentation pages.

In the "Example use" column, assume that **x** refers to an object that provides the Type in the left column.
As an example, for `Sized` type:

```
    # string is a Sized type
    x = "strings have length"
```

Type	Provides methods	Example use (*)

	`__call__()`	`x = MyCallable()`
		`x()`

Sized		`len(x)`

	`__next__()`	`while True:`
		`    print(next(x))`

	`__iter__()`	`# 2 typical uses that do not`
		`# explicitly call iter()`

		`"apple" in x`

	combines 3 types:	`"apple" in x   # True or False`
		`len(x)`
		`[print(item) for item in x]`

	`__getitem__()`
	`__len__()`
		`x[2]`
		`x["foo"]`
		Name the most basic type that specifies this behavior

2. We have a Scorecard class that creates an *iterator*. How can we specify that the iterator always produces float values?

```
    class Scorecard(________________________):
         def __init__(self, name):
        self.name = name
        self.scores = []
```

3. Fill in the blanks with correct types. Use the most specific type that applies

```
# in actual use, type hints aren't need in assignments like this
today:__________ = datetime.today()
weekend:________ = today.isoweekday()==0 or today.isoweekday()==6
# average expects the values to be float or int
Number = __________[      ,      ]
# The average of some items.
# "items" can be anything that we can sum and has a length.
# This includes: a list, set, tuple, and more
def average(items: _____________________) -> __________:
    return sum(items)/max(1, len(items))


# Get a mapping of sizes to price
def prices( ) -> _________________________:
     price_by_size = { "small": 25.0, "medium": 35.0, "large": 45.0 }
     return price_by_size
```

4. Add type hints to the code below.

```
class Product:
    """A kind of item that the store sells, e.g Nescafe Ice Coffee."""

    def __init__(self, product_id: ______,
                       description: _____,
                       price: _________):
        self.id: str = product_id
        self.description = description
        self.unit_price = price

class LineItem:
    """LineItem represents the purchase of a product, with a quantity"""

    def __init__(self, product: _______, quantity: ______ = 1):
        self.product = product
        self.quantity = quantity

    def get_total(self) ____________:
        return self.product.unit_price * self.quantity

    def __str__(self)  ____________:
        return self.product.description


class Sale(_______________________):
    """A sale of a collection of items"""
    def __init__(self):
        self.items: __________________ = []

    def add_item(self, item: ____________):
        """Add a LineItem to this sale"""
        self.items.append(item)

    def total(self) ____________:
        total_price = sum( item.get_total() for item in self.items )
        tax = TaxCalc.get_tax(total_price)
        return total_price + tax
    def __iter__(self):
        return iter(self.items)

    def __len__(self):
        return len(self.items)


class TaxCalc:
    # tax rate is a static (class) value
    TAX_RATE = 0.07

    @classmethod
    def get_tax(cls, amount: __________) ______________:
        """compute the tax on given amount"""
        return cls.TAX_RATE * amount
```

5. Refactoring:

In some countries, the tax rate depends on the kind of item. Food is often not taxed and luxury items are taxed at a higher rate.

a) How would you modify TaxCalc to make this sort of tax calculation possible?

b) What is the name of the refactoring?