Zróbmy sobie proste ćwiczenie myślowe…

Klasy i metody abstrakcyjne - ABC

- Czasami klasy bazowe, po których dziedziczymy, informują jedynie jakie metody muszą być zaimplementowane w klasach pochodnych ale nie dostarczają konkretnej implementacji.
- Takie klasy nazywamy abstrakcyjnymi. Dokładnie każda klasa, która posiada co najmniej jedną abstrakcyjną metodę (taką, która posiada jedynie sygnaturę ale nie dostarcza jej implementacji) jest nazywana abstrakcyjną.
- Nie można stworzyć bezpośrednio obiektu klasy, która jest abstrakcyjna, najpierw trzeba dostarczyć konkretną klasę, dziedziczącą po abstrakcyjnej, która implementuje wszystkie abstrakcyjne metody.
- Np. możemy stworzyć abstrakcyjną klasę reprezentującą instrument muzyczny. Wiemy, że każdy instrument potrafi jakoś grać i to jest nasza abstrakcyjna metoda.
- Każdy konkretny instrument będzie dostarczał implementacji abstrakcyjnej metody **play**.

```python
import abc


class MusicalInstrument(abc.ABC):

    @abc.abstractmethod
    def play(self):
        pass
```

Klasy i metody abstrakcyjne - ABC

- Do stworzenia abstrakcyjnej klasy jest nam potrzebny standardowy moduł **abc** (**A**bstract **B**ase **C**lasses).
- Klasa abstrakcyjna powinna dziedziczyć po **abc.ABC** aby zaznaczyć fakt, że jest abstrakcyjna.
- Ponadto powinna posiadać co najmniej jedną metodę udekorowaną dekoratorem **@abc.abstractclass**. W naszym przypadku jest to metoda **play**. Jak widać, nie dostarczamy żadnej implementacji tej metody, od razu piszemy **pass**.
- Chcemy stworzyć konkretną klasę **Guitar** dlatego w deklaracji, klasy zaznaczamy, że dziedziczymy z **MusicalInstrument**. Jednak nasza definicja nie jest poprawna, co sygnalizuje nam PyCharm podkreślając nazwę naszej klasy.
- Problem polega na tym, że nie dostarczyliśmy w klasie **Guitar** konkretnej implementacji metody **play**.

```python
import abc


class MusicalInstrument(abc.ABC):

    @abc.abstractmethod
    def play(self):
        pass
```

```python
class Guitar(MusicalInstrument):
    pass
```

Klasy i metody abstrakcyjne – definicja metody abstrakcyjnej

```python
from abc import ABC, abstractmethod

class MusicalInstrument(ABC):
    @abstractmethod
    def play(self):
        pass
```

lub

```python
from abc import ABC, abstractmethod

class MusicalInstrument(ABC):
    @abstractmethod
    def play(self):
        ...
```

Klasy i metody abstrakcyjne - ABC

- Teraz nasza konkretna klasa jest zdefiniowana poprawnie - gitara mówi nam konkretnie jakie dźwięki z siebie wydaje.
- Ponadto PyCharm na marginesie prezentuje nam przyciski, dzięki którym możemy się wygodnie przełączać pomiędzy abstrakcyjną deklaracją metody a jej konkretnymi implementacjami w klasach pochodnych - nawet jeśli klasy są w oddzielnych plikach.

```
1   import abc
2
3
4   class MusicalInstrument(abc.ABC):
5
6       @abc.abstractmethod
7       def play(self):
8           pass
9
10
11  class Guitar(MusicalInstrument):
12      def play(self):
13          return "Brzdęk, brzdęk"
```

```
In [1]: paste
import abc


class MusicalInstrument(abc.ABC):

    @abc.abstractmethod
    def play(self):
        pass


class Guitar(MusicalInstrument):
    def play(self):
        return "Brzdęk, brzdęk"
## -- End pasted text --

In [2]: some_instrument = MusicalInstrument()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-a5b6411af71c> in <module>
----> 1 some_instrument = MusicalInstrument()

TypeError: Can't instantiate abstract class MusicalInstrument with abstract methods play

In [3]: my_gibson = Guitar()

In [4]: my_gibson.play()
Out[4]: 'Brzdęk, brzdęk'
```

- Na przykładzie jasno widać, że nie da się stworzyć obiektu klasy abstrakcyjnej.

Klasy i metody abstrakcyjne - ABC

- Możemy stworzyć więcej klas instrumentów i zestawić je w orkiestrę!
- Możemy ponadto stworzyć funkcję, która dyryguje instrumentami.
- Najlepsze jest to, że funkcja wcale nie musi wiedzieć, jakimi konkretnie instrumentami dyryguje. Wszystko co jest wystarczy to wiedza, że każdy instrument, aby mógł nazywać się instrumentem, musi dostarczać metodę **play**, więc można ją wykonać i wtedy instrument zagra w sposób specyficzny dla swojego rodzaju.
- Ta możliwość jednolitego traktowania obiektów różnych klas o wspólnej bazie jest nazywana **polimorfizmem**, który oprócz dziedziczenia i abstrakcji jest kolejną ważną koncepcją programowania obiektowego.

```
12  class Guitar(MusicalInstrument):
13      def play(self):
14          return "Brzdęk, brzdęk"
15
16
17  class Flute(MusicalInstrument):
18      def play(self):
19          return "Fiu, fiu!"
20
21
22  class Violin(MusicalInstrument):
23      def play(self):
24          return "Skrzyp, skrzyp!!"
25
26
27  def conductor(instruments:
28                typing.Sequence[MusicalInstrument]
29                ) -> None:
30      for instrument in instruments:
31          print(instrument.play())
```

Klasy i metody abstrakcyjne - ABC

- Możemy stworzyć więcej klas instrumentów i zestawić je w orkiestrę!
- Możemy ponadto stworzyć funkcję, która dyryguje instrumentami.
- Najlepsze jest to, że funkcja wcale nie musi wiedzieć, jakimi konkretnie instrumentami dyryguje. Wszystko co jest wystarczy to wiedza, że każdy instrument, aby mógł nazywać się instrumentem, musi dostarczać metodę **play**, więc można ją wykonać i wtedy instrument zagra w sposób specyficzny dla swojego rodzaju.
- Ta możliwość jednolitego traktowania obiektów różnych klas o wspólnej bazie jest nazywana polimorfizmem, który oprócz dziedziczenia i abstrakcji jest kolejną ważną koncepcją programowania obiektowego.

```
In [2]: orchestra = [Guitar(), Violin(), Flute()]

In [3]: conductor(orchestra)
Brzdęk, brzdęk
Skrzyp, skrzyp!!
Fiu, fiu!
```

Python ćwiczenia

Napisz program symulujący zoo.
Kod powinien się składać z klasy Zoo (na podstawie orkiestry z przykładu), posiadającej listę zwierząt typu Animal (napisz do trzech klas reprezentujących trzy dowolnie wybrane zwierzęta powinny one dziedziczyć po klasie Animal). Zaimplementuj w klasach metody odpowiedzialne za poruszanie się, dawanie głosu i odżywianie zwierząt.

Kod reprezentujący zachowywanie się zwierząt (przykładowy):
>> burek = Dog()
>> burek.make_noise()
”Hau hau!”
>> nemo = Fish()
>> nemo.make_noise()
”Bul bul bul”

Wykorzystaj inwencję twórczą ☺️

ITERATORY I GENERATORY

Iteratory

- Iteratory są obiektami, które można używać w pętli.
- Iterator powinien dostarczać dwóch specjalnych metod: __iter__ oraz __next__.
- Metoda next ma zwracać wartość kolejnego obiegu pętli lub wyrzucać specjalny wyjątek StopIteration, aby zakończyć działanie pętli.
- Metoda __iter__ w iteratorze zawsze powinna zwracać jego samego, czyli self.
- Przykładem może być klasa implementująca ciąg liczb Fibonacciego o długości N lub klasa implementująca ciąg potęg dwójki.

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max = 0):
        self.max = max
        self.n = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

Iteratory

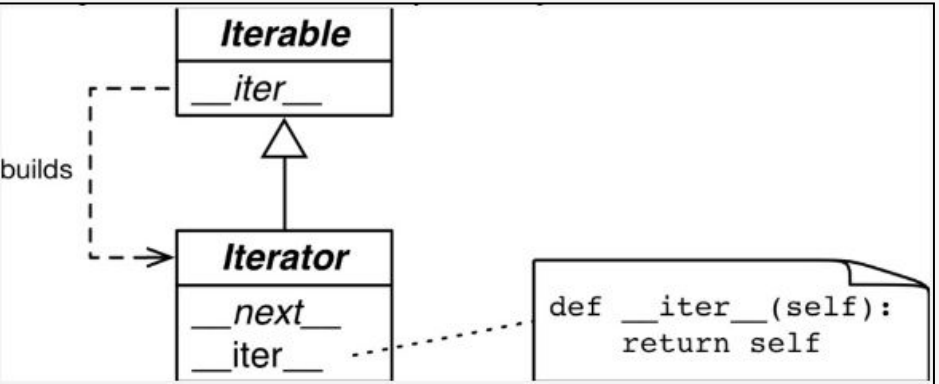
- Iteratory są obiektami, które można używać w pętli.
- Iterator powinien dostarczać dwóch specjalnych metod: __iter__ oraz __next__.
- Metoda next ma zwracać wartość kolejnego obiegu pętli lub wyrzucać specjalny wyjątek StopIteration, aby zakończyć działanie pętli.
- Metoda __iter__ w iteratorze zawsze powinna zwracać jego samego, czyli self.
- Przykładem może być klasa implementująca ciąg liczb Fibonacciego o długości N.

```
In [43]: for i in PowTwo(10):
    ...:     print(i)
    ...:
1
2
4
8
16
32
64
128
256
512
1024
```

Iteratory

● Pojawia się jednak pewien problem - po iteratorze można przejść tylko raz - zużyty iterator wyrzuca ciągle StopIteration i nie da się go ponownie wykorzystać.

● Rozwiązaniem jest podzielenie funkcjonalności pomiędzy Iterable-obiekt, po którym można iterować. Obiekt ten dostarcza metodę __iter__, która za każdym razem zwraca jednorazowy iterator ale dzięki temu można po niej iterować




```
In [50]: power = PowTwo(10)

In [51]: for i in power:
    ...:     print(i)
    ...:
1
2
4
8
16
32
64
128
256
512
1024

In [52]: for i in power:
    ...:     print(i)
    ...:
```

Iteratory

```python
class PowTwo:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        return PowTwoIterator(self.max)


class PowTwoIterator:
    def __init__(self, max):
        self.max = max
        self.n = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

```
In [59]: power = PowTwo(5)

In [60]: for i in power:
    ...:     print(i)
    ...:
1
2
4
8
16
32

In [61]: for i in power:
    ...:     print(i)
    ...:
1
2
4
8
16
32
```

Iteratory

Napisz klasę mającą właściwości iteratora z jednym argumentem do konstruktora (liczbą n). W przypadku wrzucenia obiektu tej klasy na iterację do pętli for, powinna ona być w stanie wypisać liczby od 0 do n.