ΠΕΡΙΕΧΟΜΕΝΑ:

1. Iterators
2. Iterables
3. Ενσωματωμένες Συναρτήσεις για Iterators
4. OPC:
    1. Fibonacci Numbers

Γιώργος Μ.

Σμαραγδένιος Χορηγός Μαθήματος

Πασχάλης Μ.

Σμαραγδένιος Χορηγός Μαθήματος

- Μία κλάση θεωρείται **iterator (επαναληπτής)**, όταν μπορεί να παράξει επαναληπτικά, ένα-ένα, στοιχεία μιας ακολουθίας με την εξής λειτουργικότητα:
  - **next()**: να επιστρέφει το επόμενο στοιχείο.
  - **for x in ob**: να διατρέχει τα στοιχεία.

- Κατασκευάζουμε έναν iterator ορίζοντας στην κλάση μας (επιπλέον) τις dunder methods:
  - **__iter__(self)**
    - Αρχικοποιεί ή ορίζει απαιτούμενες μεταβλητές
    - Πρέπει να επιστρέφει το ίδιο το αντικείμενο
  - **__next__(self)**
    - Επιστρέφει την επόμενη τιμή
- Στο κυρίως πρόγραμμα:
  - Κατασκευάζουμε ένα αντικείμενο της κλάσης μας:

```
obj = MyIterator()
```

  - Κατασκευάζουμε τον iterator μετατρέποντας ένα αντικείμενο της κλάσης μας με τη built-in συνάρτηση:

```
it = iter(object)
```

  - και έπειτα μπορούμε να χρησιμοποιήσουμε τη next() για να πάρουμε το επόμενο στοιχείο της ακολουθίας

```
next(it)
```

Παράδειγμα 1: iterator.py

```
class MyIterator:
    def __iter__(self):
        self.n = 0
        return self
    def __next__(self):
        self.n += 1
        return self.n
```

```
obj = MyIterator()
it = iter(obj)

print(next(it))
print(next(it))
print(next(it))
```

- Για να τερματίζει η επανάληψη (με τη for) πρέπει να κάνουμε raise την εξαίρεση **StopIteration** (δεν απαιτείται επαναληπτής – κατασκευάζεται αυτόματα από τη for)

Παράδειγμα 2: iterator.for.py

```
class MyIterator:
    def __iter__(self):
        self.n = 0
        return self
    def __next__(self):
        self.n += 1
        if self.n<=10:
            return self.n
        else:
            raise StopIteration
```

```
obj = MyIterator()
it = iter(obj)

for i in it:
    print(i)
```

```
obj = MyIterator()

for i in obj:
    print(i)
```

- **Iterable (αντικείμενο στο οποίο μπορούμε να κάνουμε επανάληψη)**, είναι κάθε αντικείμενο στο οποίο είναι δυνατό να παίρνουμε τιμές μία κάθε φορά (και το σύνολο τους να είναι πεπερασμένο).
- Iterables είναι:
  - Δικά μας αντικείμενα στα οποία έχουμε ορίσει την __next__() και την __iter__()
  - Όλες οι ακολουθίες (sequences) π.χ. list, tuple, range, string
  - μη ακολουθιακές δομές, όπως π.χ. dict

Παράδειγμα 3: iterator.list.dict.py

```
my_list = [1,2,3,4]
it = iter(my_list)
print(next(it))
print(next(it))
print(next(it))

my_dict = {"a":1, "b":2 }
it = iter(my_dict)
print(next(it))
print(next(it))
print(next(it))
```

Παρατηρήση 1: Iterators και for

- Η for κατασκευάζει εσωτερικά έναν επαναληπτή και έπειτα τον διατρέχει

- Δηλαδή το σχήμα που ακολουθεί η for είναι:

```
# for.implementation.py
iter_obj = iter([1,2,3])
while True:
    try:
        element = next(iter_obj)
        #.. do something with element
    except StopIteration:
        break
```

Παρατηρήση 2: Μία σημαντική χρήση των iterators

Παρατηρήστε τη διαφορά στις ακόλουθες δύο εκδοχές "επεξεργασίας δεδομένων" (βλ.και βίντεο):

```
# read.data.fail.py
for data in list(range(100000000)):
    print(data)
```

```
# read.data.success.py
class MyIterator:
    def __iter__(self):
        self.n = 0
        return self
    def __next__(self):
        self.n += 1000
        if self.n<=100000000:
            return list(range(self.n-1000,
                              self.n))
        else:
            raise StopIteration
for stream_data in MyIterator():
    for data in stream_data:
        print(data)
```

- Υπάρχουν **αρκετές ενσωματωμένες (built-in) συναρτήσεις που δουλεύουν πάνω σε iterators.**

1. Μετατροπές iterators σε σύνθετες δομές της Python:
   - **list(iter):** Μετατροπή σε λίστα
   - **tuple(iter):** Μετατροπή σε tuple
   - **set(iter):** Μετατροπή σε σύνολο
   - **frozenset(iter):** Μετατροπή σε λεξικό

Παράδειγμα 4: iterator.conversions.py

```
class MyIterator:
    ...

it = iter(MyIterator())
print(list(it))
print(tuple(it))
print(set(it))
print(frozenset(it))
print(dict(enumerate(it)))
```

Παρατηρήση 1:

- Οι παραπάνω μετατροπές δουλεύουν ακόμη κι αν επιδράσουν απ'ευθείας πανω στο αντικείμενο-iterator:

```
print(list(MyIterator()))
```

2. Συναρτήσεις υπολογισμού σε iterators

- **min**(iter): Επιστρέφει το ελάχιστο στοιχείο
- **max**(iter): Επιστρέφει το μέγιστο στοιχείο
- **sum**(iter): Επιστρέφει το άθροισμα των στοιχείων
- **all**(iter): True, αν όλα τα στοιχεία ερμηνεύονται σε True
- **any**(iter): True, αν τουλάχιστον ένα στοιχείο ερμηνεύεται σε True

Παρατηρήση για τις all και any:

- Τα 0 (int), 0.0 (float), κενή συμβολοσειρά (string) ερμηνεύονται σε False, ενώ όλες οι υπόλοιπες τιμές είναι True.

Παράδειγμα 5: iterator.builtin1.py

```
class MyIterator:
    ...

print(list(MyIterator()))
print(f"min={min(MyIterator())}")
print(f"max={max(MyIterator())}")
print(f"sum={sum(MyIterator())}")
print(f"all={all(MyIterator())}")
print(f"any={any(MyIterator())}")
```

Άσκηση 1: Κατασκευάστε έναν iterator που υπολογίζει τους πρώτους 100 περιττούς φυσικούς αριθμούς. Έπειτα υπολογίστε το άθροισμά τους.

3. Απαρίθμηση μελών:

- **enumerate(iter):** Επιστρέφει iterable από tuples που το καθένα έχει έναν αύξοντα αριθμό (ξεκινά από το μηδέν) και το τρέχον στοιχείο του iterator
- **enumerate(iter, start):** Ομοίως αλλά η αρίθμηση ξεκινά από το start

Παράδειγμα 6: enumerate1.py

```
class MyIterator:
    ...

print(enumerate(MyIterator()))
print(list(enumerate(MyIterator())))
print(list(enumerate(MyIterator(),100)))
```

Παρατήρηση: Ένα iterable από tuples μπορεί να χρησιμοποιηθεί με for 2 μεταβλητών, στις οποίες γίνεται unpacking του τρέχοντος στοιχείου της επανάληψης.

Παράδειγμα 7: enumerate2.py

```
class MyIterator:
    ...

for index, iter_value in enumerate(MyIterator(),100):
    print(index, iter_value)
```

4. Συνένωση iterators

- **zip**(iter1, iter2, …): Συνενώνει τα iterable σε μία ενιαία, η οποία αποτελείται από tuples που έχουν ένα στοιχείο από κάθε iterable (με τη σειρά)

Παράδειγμα 8: zip.py

```
class MyIterator:
    ...
a = MyIterator(10)
b = MyIterator(20)
c = MyIterator(30)
for x,y,z in zip(a,b,c):
    print(x,y,z)
```

5. Άλλες built-in συναρτήσεις για iterators:

- **reversed**(iter): Δουλεύει για ακολουθίες μόνο, αλλά μπορούμε να εμπλουτίσουμε την κλάση μας προσθέτοντας τη dunder method __reversed__
- **sorted**(iter): Επιστρέφει ένα iterable το οποίο περιέχει τα στοιχεία ταξινομημένα σε αύξουσα σειρά. Προαιρετικό όρισμα: reversed=true.
- **map**(iter) και **filter**(iter): Θα καλυφθούν σε επόμενο μάθημα της σειράς αυτής.
- Σημειώστε ότι υπάρχει και ειδικό πακέτο, το **itertools** το οποίο περιέχει επιπλέον συναρτήσεις για iterables.

```python
# opc.fibonacci.py
# source: https://www.journaldev.com/14620/python-iterator
class fibo:
    def __init__(self):
        # default constructor
        self.prev = 0
        self.cur = 1
        self.n = 1
    def __iter__(self):  # define the iter() function
        return self
    def __next__(self):  # define the next() function
        if self.n < 10:  # Limit to 10
            result = self.prev + self.cur
            self.prev = self.cur
            self.cur = result
            self.n += 1
            return result
        else:
            raise StopIteration  # raise exception

iterator = iter(fibo())

while True:
    # print the value of next fibonacci up to 10th fibonacci
    try:
        print(next(iterator))
    except StopIteration:
        print('First 9 fibonacci numbers have been printed already.')
        break  # break the loop
```

```python
# opc.primes.py
# modification on
# https://school.geekwall.in/p/r1halvojG/what-can-you-use-python-generator-functions-for
def check_prime(number):
    for divisor in range(2, int(number ** 0.5) + 1):
        if number % divisor == 0:
            return False
    return True
class Primes:
    def __init__(self):
        self.number = 1
    def __iter__(self):
        return self
    def __next__(self):
        self.number += 1
        if check_prime(self.number):
            return self.number
        else:
            return self.__next__()
primes = Primes()
print(primes)
for x in primes:
    print(x)
```