ΠΕΡΙΕΧΟΜΕΝΑ:

1. Generators
2. Συντομογραφίες Generators
3. .send(), .close() και .throw()
4. ΟΡΣ:
    1. Διάβασμα Αρχείων
    2. Αλφαβητική σειρά σε δένδρο

Ιωάννης Κ.

Χρυσός Χορηγός Μαθήματος

Πάνος Γ.

Ασημένιος Χορηγός Μαθήματος

- Ένας **generator**, είναι μία συνάρτηση που κατασκευάζει έναν iterator με πιο εύκολο τρόπο (από το να κατασκευάζουμε μία κλάση):
  - Πρακτικά **είναι μία συνάρτηση** που δεν καταστρέφεται όταν περατωθεί η λειτουργία της
  - αλλά **διατηρεί την κατάσταση της** (τιμές των τοπικών μεταβλητών της και σημείο στο οποίο τερματίστηκε η λειτουργία της), όταν ξανακληθεί.
- Είναι ακριβώς ίδια στο συντακτικό με μία συνηθισμένη συνάρτηση, αλλά όταν επιστρέφει την τιμή, χρησιμοποιεί τη λέξη-κλειδί **yield (αντί για τη λέξη κλειδί return)**
  - Η yield είναι αυτό που την κάνει να είναι generator και όχι μία απλή συνάρτηση

Παρατηρήσεις για τη σύνταξη και χρήση ενός generator

- Κατασκευάζουμε τη συνάρτηση generator χρησιμοποιώντας τη yield αντί της return, π.χ.:

```
yield value
```

- Έπειτα στο κυρίως πρόγραμμα:
  - Κατασκευάζουμε έναν iterator από τον generator (προσοχή δεν "καλούμε συνάρτηση", αλλά "δημιουργούμε iterator"):

```
iterator_name = generator_name()
```

  - Έπειτα, ακριβώς όπως ο iterator (με τη next() ή διατρέχοντας τα στοιχεία με μία for)

Παράδειγμα 1: generator.py

```
def f():
    yield 1
    yield 2
    yield 3

it = f()
for _ in range(3):
    print(next(it))
```

Παράδειγμα 2: generator2.py

```
def f():
    for i in range(10):
        yield i
it = f()
for i in it:
    print(i)

it = f()
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

Σημείωση: Δεδομένου ότι η for δημιουργεί iterator, μπορούμε να κάνουμε επανάληψη και απ'ευθείας, π.χ. **for elem in generator()**

- Η Python, μας δίνει τη δυνατότητα να έχουμε μια ακόμη πιο εύκολη σύνταξη generators με τις **συντομογραφίες generators** (αναφέρονται ως **generator expressions**)
- Έχουν ακριβώς την ίδια σύνταξη με τις περιγραφικές λίστες (list comprehensions), αλλά χρησιμοποιούν παρενθέσεις αντί για αγκύλες.

Παράδειγμα 3: gen.expression1.py

```
g = (i for i in range(4))

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

Παράδειγμα 4: gen.expression2.py

```
g = (i for i in range(4) if i%2==0)

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

Παράδειγμα 5: gen.expression3.py

```
g = (i if i%2==0 else i*i for i in range(4))

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

Παρατήρηση: Είναι σημαντικό να κατανοούμε τη διαφορά ανάμεσα στα 3 snippets τα οποία κάνουν την ίδια δουλειά:

```
g = [i for i in range(4)]
for i in g:
   print(i)
```

- Εδώ πρώτα κατασκευάζεται η λίστα και έπειτα διατρέχονται τα στοιχεία της.

```
for i in [1,2,3,4]:
   print(i)
```

- Εντελώς αντίστοιχα η λίστα προϋπάρχει της επανάληψης.

```
g = (i for i in range(4))
for i in g:
   print(i)
```

- Αντίθετα εδώ δεν έχει κατασκευαστεί η λίστα, αλλά αντίθετα κάθε αριθμός παράγεται όταν τον χρειαζόμαστε (μία τρομακτική διαφορά στις απαιτήσεις μνήμης του προγράμματος)

Άσκηση 1: Κατασκευάστε έναν iterator και διατρέξτε τα τετράγωνα των αριθμών από το 10 έως και το 20 με τρεις τρόπους:

- Μέσω iterator (class)
- Μέσω generator (function)
- Με συντομογραφία generator

- **<u>Μας δίνεται η δυνατότητα να "στείλουμε" στον generator τιμές</u>**, τις οποίες μπορεί να τις αξιοποιήσει συνεχίζοντας τον υπολογισμό του.
- Λαμβάνουμε την τιμή στον generator, παίρνοντας την επιστρεφόμενη τιμή της yield:

```
ret = yield val
```

- Στέλνουμε τιμή στον generator, χρησιμοποιώντας τη μέθοδο send() του αντικειμένου-επαναληπτή που δημιουργήσαμε.

```
it.send(val)
```

Προσοχή ότι η **<u>send</u>** ενεργοποιεί την next() δηλαδή πάντα θα επιστρέφεται σε αυτήν η επόμενη τιμή του generator.

<u>Παράδειγμα 6: send.py</u>

```
from random import randrange

def f():
    cnt=0
    for rounds in range(10):
        ret = yield cnt
        if ret is None:
            cnt+=1
        else:
            cnt+=1+ret
```

```
it = f()
for it_val in it:
    print(it_val)
    if randrange(2)==0:
        val = it.send(100)
        print(val)
```

- Η **close()** είναι μία μέθοδος του iterator που "κλείνει" αμέσως την επανάληψη, εξωτερικά, χωρίς να φτάσει ποτέ στο τέλος του.

<u>Παράδειγμα 7: close.py</u>

```
def f():
    i=1
    while True:
        yield i
        i+=2
```

```
it = f()
for i in it:
    print(i)
    if i==101:
        it.close()
```

- Η **throw()** είναι μία μέθοδος του iterator που τον αναγκάζει να προκαλέσει άμεσα την εξαίρεση που διοχετεύουμε ως όρισμα

<u>Παράδειγμα 8: throw.py</u>

```
def f():
    i=1
    while True:
        yield i
        i+=2
```

```
it = f()
for i in it:
    print(i)
    if i==101:
        it.throw(Exception)
```

OPC: Διάβασμα αρχείου σε κομμάτια

```
# source: https://stackoverflow.com/questions/519633/lazy-method-for-reading-big-file-in-python

def read_in_chunks(file_object, chunk_size=1024):
    """Lazy function (generator) to read a file piece by piece.
    Default chunk size: 1k."""
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break
        yield data


with open('romeonjuliet.txt') as f:
    for piece in read_in_chunks(f):
        print(piece)
```

OPC: Διάβασμα αρχείου .csv

```
# source: https://www.dataquest.io/blog/python-generators-tutorial/
data = "sample.csv"
lines = (line for line in open(data, encoding="ISO-8859-1"))
for line in lines:
    cells = line.split(";")
    for c in cells:
        print(c.strip(), end="\t")
    print("")
```

```
# A binary tree class.
# source (mod): https://www.python.org/dev/peps/pep-0255/
class Tree:

    def __init__(self, label, left=None, right=None):
        self.label = label
        self.left = left
        self.right = right

    def __repr__(self, level=0, indent="    "):
        s = level*indent + str(self.label)
        if self.left:
            s = s + "\n" + self.left.__repr__(level+1, indent)
        if self.right:
            s = s + "\n" + self.right.__repr__(level+1, indent)
        return s

    def __iter__(self):
        return inorder(self) # Create a Tree from a list.

def tree(list):
    n = len(list)
    if n == 0:
        return []
    i = n // 2
    return Tree(list[i], tree(list[:i]), tree(list[i+1:]))
```

```
# A non-recursive generator.
def inorder(node):
    stack = []
    while node:
        while node.left:
            stack.append(node)
            node = node.left
        yield node.label
        while not node.right:
            try:
                node = stack.pop()
            except IndexError:
                return
            yield node.label
        node = node.right

# Exercise the non-recursive generator.
t = tree("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
for x in t:
    print(x, end="")
```