11. ΕΜΒΑΘΥΝΣΗ ΚΩΔΙΚΑ – LAMBDA FUNCTIONS - MULTITHREADING

11.1 Ασκήσεις 10ης εβδομάδας

Άσκηση 1 (generate_dict_n.py)

Ζητήστε από τον χρήστη έναν ακέραιο αριθμό n. Γράψτε ένα πρόγραμμα για τη δημιουργία ενός λεξικού που περιέχει σαν κλειδί έναν αριθμό 'I' και σαν τιμή τον i * i (δηλ. το τετράγωνό του i x i) τέτοιο ώστε να είναι ένας ακέραιος αριθμός μεταξύ του 1 και του n (και τα δύο περιλαμβάνονται) που έχει δοθεί από τον χρήστη. και μετά το πρόγραμμα θα πρέπει να εκτυπώσει το λεξικό. Αν για παράδειγμα δώσουμε τον αριθμό 7, θα πρέπει να πάρουμε σαν έξοδο:

{1: 1}

{1: 1, 2: 4}

{1: 1, 2: 4, 3: 9}

{1: 1, 2: 4, 3: 9, 4: 16}

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49}

Κατόπιν, μετατρέψτε τον κώδικα σε dictionary comprehension.

Λύση 1

```
print('''Δώστε έναν αριθμό και θα τυπώσουμε ένα λεξικό

με τους ακέραιους μέχρι αυτόν, υψωμένους εις το τετράγωνο: ''')


n = int(input())

ans = {}

for i in range(1, n + 1):

    ans[i] = i * i

    print(ans)
```

Λύση 2

```
print('Δώστε έναν αριθμό ')

n = int(input())

ans = {i:i*i for i in range(1, n+1)}

print(ans)
```

Άσκηση 2 (print_dict.py)

Γράψτε μια συνάρτηση η οποία μπορεί να τυπώνει ένα λεξικό όπου τα κλειδιά είναι οι αριθμοί από το 1 μέχρι και το 20 και οι τιμές είναι τα τετράγωνά τους. Χρησιμοποιήστε την dictionary comprehension. Αν όλα πάνε σωστά, η εκτύπωσή σας θα είναι:

{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225, 16: 256, 17: 289, 18: 324, 19: 361, 20: 400}

Λύση

```
def printDict():
    dict = {i: i ** 2 for i in range(1, 21)}
    print(dict)
printDict()
```

Άσκηση 3 (count_freq.py)

Περαιτέρω σχόλια για τον παραπάνω κώδικα:

freq_dict.items(): Παίρνει όλα τα στοιχεία (items, δηλαδή τα ζεύγη key-value) από το freq_dict dictionary και τα επιστρέφει σαν μια σειρά από πλειάδες (tuples).

sorted(...): Αυτή η συνάρτηση παίρνει τη σειρά των tuples από το freq_dict.items() και τα ταξινομεί, σύμφωνα με τη συνάρτηση κλειδί (key).

key=take_second: Ορίζουμε πως ο τρόπος που θέλουμε να ταξινομήσουμε τα στοιχεία μας, είναι σύμφωνα με το δεύτερο στοιχείο (item[1]). Κάθε item αντιπροσωπεύει ένα tuple από το freq_dict.items και το item[1] αντιπροσωπεύει το δεύτερο στοιχείο, δηλ. τη συχνότητα του κάθε χαρακτήρα.

reverse=True: Το default στη συνάρτηση sorted() είναι False, δηλαδή αύξουσα σειρά. Με το True, ζητάμε το αντίστροφο, δηλαδή από το μεγαλύτερο προς το σύμφωνα με τη συχνότητα εμφάνισης.

{k: v for k, v in ...}: Εδώ έχουμε ένα dict comprehension το οποίο δημιουργεί ένα νέο λεξικό το οποίο βασίζεται σε ταξινομημένη σειρά από tuples. Διαπερνά όλες τις πλειάδες και για κάθε μια, αναθέτει το πρώτο στοιχείο (το k, που αντιπροσωπεύει τον χαρακτήρα) στο κάθε κλειδί (key) και το δεύτερο στοιχείο (v, που αντιπροσωπεύει τη συχνότητα εμφάνισης του κάθε χαρακτήρα) σαν την τιμή, στο νέο λεξικό.

Πώς λειτουργεί η sorted();

Σύνταξη

sorted(iterable, key=function, reverse=True/False)

Τιμές παραμέτρων

iterable Απαιτείται. Το αντικείμενο προς ταξινόμηση, λίστα, λεξικό, πλειάδα κ.λ.π..

key Προαιρετικό. Μία συνάρτηση που εκτελείται για να επιλέξουμε τη σειρά. Το Default είναι None

reverse Προαιρετικό. Μια Boolean μεταβλητή. Το False ταξινομεί με αύξουσα σειρά, το True με φθίνουσα. Το Default είναι False.

Παραδείγματα

1.

a = (1, 11, 2)

x = sorted(a)

print(x)

Έξοδος:

[1, 2, 11]

2.

```
a = ("h", "b", "a", "c", "f", "d", "e", "g")

x = sorted(a, reverse=True) # αντίστροφη ταξινόμηση, απ' το τέλος προς την αρχή

print(x)
```

Έξοδος:

['h', 'g', 'f', 'e', 'd', 'c', 'b', 'a']

Λύση 2 (με lamda function)

```
input_str = input("Please type a string: ")

# Δημιουργούμε ένα κενό λεξικό για να δεχθεί τη συχνότητα των γραμμάτων

freq_dict = {}


# Διαπερνάμε κάθε χαρακτήρα της εισόδου και ενημερώνουμε το λεξικό

for char in input_str:

    if char in freq_dict:

        freq_dict[char] += 1

    else:

        freq_dict[char] = 1
```

```
# Ταξινομούμε σύμφωνα με την τιμή (συχνότητα) σε φθίνουσα σειρά

# key=lambda item: item[1]: Εδώ χρησιμοποιούμε μια lambda function ή
# ανώνυμη συνάρτηση κι έχουμε το ίδιο αποτέλεσμα, δηλώνοντας ποιο
# στοιχείο θέλουμε να χρησιμοποιήσουμε για την ταξινόμηση
#(επιστρέφουμε και πάλι το item[1], δηλαδή το δεύτερο στοιχείο, όπως #
και με τη χρήση της συνάρτησης take_second().


sorted_dict = {k: v for k, v in sorted(freq_dict.items(),  key=lambda item:
item[1], reverse=True)}


# Εκτύπωση ταξινομημένου λεξικού

for key, value in sorted_dict.items():

    print(f" {key} {value}", end="")
```

11.1 Τι είναι οι lambda functions

Στην Python, οι συναρτήσεις lambda, επίσης γνωστές ως ανώνυμες συναρτήσεις, είναι μικρές συναρτήσεις που δημιουργούνται inline και δεν απαιτούν μια τυπική δήλωση def. Χρησιμοποιούνται συχνά όταν χρειαζόμαστε μια απλή συνάρτηση για ένα σύντομο χρονικό διάστημα, συνήθως ως όρισμα σε συναρτήσεις υψηλότερης προτεραιότητας ή σε καταστάσεις όπου η δημιουργία μιας ξεχωριστής συνάρτησης θα ήταν περιττή.

Μερικές περιπτώσεις χρήσης των lambda functions είναι οι παρακάτω:

1. Functional programming (Συναρτησιακός προγραμματισμός): Οι συναρτήσεις lambda χρησιμοποιούνται συχνά σε παραδείγματα συναρτησιακού προγραμματισμού, όπου οι συναρτήσεις θεωρούνται αντικείμενα πρώτης τάξης. Μπορούν να περάσουν ως ορίσματα σε συναρτήσεις υψηλότερης προτεραιότητας, όπως οι map(), filter() και reduce(), επιτρέποντας συνοπτικό και εκφραστικότερο κώδικα.

Παραδείγματα:

 # Χρήση λειτουργίας lambda με την map()

Ορισμός και χρήση της map():

Η map() χρησιμοποιείται για να εκτελέσει μια συνάρτηση σε κάθε στοιχείο ενός διατρεχόμενου(iterable) στοιχείου.

Σύνταξη:

map(function, iterables)

Παράδειγμα:

```
numbers = [1, 2, 3, 4, 5]

squared_numbers = list(map(lambda x: x ** 2, numbers))

print(squared_numbers)

# Εκτύπωση: [1, 4, 9, 16, 25]
```

 # Χρήση λειτουργίας lambda με την filter()

Ορισμός και χρήση της filter()

Η filter επιστρέφει έναν iterator όπου τα στοιχεία φιλτράρονται μέσω μιας συνάρτησης ελέγχοντας αν κάθε στοιχείο περνάει ή όχι από το φίλτρο.

Σύνταξη:

filter(function, iterable)

Παράδειγμα:

```
numbers = [1, 2, 3, 4, 5]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))

print(even_numbers)  # Εκτύπωση: [2, 4]
```

Ορισμός και χρήση της reduce()

Η reduce επιστρέφει έναν αριθμό, μειώνοντας κατά μια τις διαστάσεις ενός iterable, εφαρμόζοντας τη συνάρτηση σε κάθε στοιχείο του iterable.

Σύνταξη:

reduce(function, iterable)

Παράδειγμα:

```
from functools import reduce


nums = [1, 2, 3, 4]

ans = reduce(lambda x, y: x + y, nums)

print(ans) # Εκτύπωση: 10    Λύση της εξίσωσης: (((1+2)+3)+4)
```

2. Inline expressions (Ενσωματωμένες εκφράσεις): Όταν χρειαζόμαστε μια μικρή έκφραση ή υπολογισμό που δεν απαιτεί τη δημιουργία μιας κλασικής συνάρτησης, μια συνάρτηση lambda μπορεί να χρησιμοποιηθεί για να δημιουργήσει συμπαγή και αυτόνομο κώδικα. Για παράδειγμα, η ταξινόμηση μιας λίστας από tuples βασιζόμενη σε ένα συγκεκριμένο στοιχείο μπορεί να γίνει χρησιμοποιώντας μια συνάρτηση lambda ως όρισμα "key" για την sorted() (όπως στο παράδειγμά μας).

Παράδειγμα

```
# Χρήση λειτουργίας lambda για ταξινόμηση βάσει συγκεκριμένου στοιχείου

students = [("Alice", 25), ("Bob", 30), ("Charlie", 20)]

students_sorted_by_age = sorted(students, abc=lambda x: x[1])

print(students_sorted_by_age)  # Εκτύπωση: [('Charlie', 20), ('Alice', 25), ('Bob', 30)]
```

3. Callbacks (κλήσεις επιστροφής): Οι συναρτήσεις lambda μπορούν να είναι χρήσιμες όταν εργαζόμαστε με προγραμματισμό βασισμένο σε συμβάντα ή callbacks. Μπορούμε να ορίσουμε μικρές, ανώνυμες συναρτήσεις κατά τη διάρκεια εκτέλεσης για να εκτελεστούν όταν συμβεί ένα συγκεκριμένο γεγονός ή ως απόκριση σε μια συγκεκριμένη συνθήκη.

Παράδειγμα

```
# Χρήση λειτουργίας lambda ως κλήση επιστροφής κλήσης

def event_handler(callback):

    # Κάποια λογική εκτέλεσης

    callback()


event_handler(lambda: print("Εκτελέστηκε η κλήση επιστροφής."))
```

4. Μείωση πολυπλοκότητας του κώδικα: Σε ορισμένες περιπτώσεις, η χρήση μιας συνάρτησης lambda μπορεί να απλοποιήσει τον κώδικά μας αποφεύγοντας την ανάγκη για ορισμό μιας κλασικής συνάρτησης.

Παράδειγμα

```
# Χρήση λειτουργίας lambda για απλή υπολογιστική έκφραση
addition = lambda x, y: x + y
result = addition(5, 3)
print(result)  # Εκτύπωση: 8
```

Άλλα παραδείγματα:

Εφαρμογή συναρτήσεων σε όλα τα στοιχεία μιας λίστας:

```
numbers = [1, 2, 3, 4, 5]
processed_numbers = list(map(lambda x: x * 2, numbers))
print(processed_numbers)  # Εκτύπωση: [2, 4, 6, 8, 10]
```

Φιλτράρισμα στοιχείων μιας λίστας με βάση συγκεκριμένη συνθήκη:

```
numbers = [1, 2, 3, 4, 5]
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # Εκτύπωση: [2, 4]
```

Ανώνυμη συνάρτηση ως όρισμα σε μια άλλη συνάρτηση:

```
def process_numbers(numbers, func):
    processed_numbers = []
    for num in numbers:
        processed_numbers.append(func(num))
    return processed_numbers
```

```
numbers = [1, 2, 3, 4, 5]

result = process_numbers(numbers, lambda x: x ** 2)

print(result)  # Εκτύπωση: [1, 4, 9, 16, 25]
```

Με τα παραδείγματα βλέπουμε πως οι συναρτήσεις lambda μπορούν να χρησιμοποιηθούν για να κάνουν τον κώδικα πιο συνοπτικό, εκφραστικό και ευανάγνωστο σε περιπτώσεις όπου η δημιουργία ονομασμένης συνάρτησης δεν είναι απαραίτητη.

11.2 Multithreading – Multithreading

Το multithreading (πολυνηματικότητα) είναι μια ιδέα η οποία μας επιτρέπει να εκτελούμε πολλές διεργασίες ταυτόχρονα σε ένα πρόγραμμα Python. Σκεφτείτε το multithreading σαν να έχουμε πολλά threads που τρέχουν παράλληλα στο πρόγραμμά μας.

Στην κανονική ροή εκτέλεσης ενός προγράμματος Python, οι εντολές εκτελούνται μία-μία κατά σειρά. Αυτό σημαίνει ότι πρέπει να περιμένουμε μέχρι να ολοκληρωθεί μια εντολή προτού ξεκινήσει η

επόμενη. Με το multithreading, μπορούμε να ξεκινήσουμε πολλές εντολές ταυτόχρονα και να τις εκτελέσουμε παράλληλα.

Για να χρησιμοποιήσουμε multithreading σε ένα πρόγραμμα Python, πρέπει να χρησιμοποιήσουμε τη βιβλιοθήκη `threading`. Με αυτή τη βιβλιοθήκη, μπορούμε να δημιουργήσουμε threads και να τα εκτελέσουμε ταυτόχρονα.

Ένα παράδειγμα χρήσης multithreading μπορεί να είναι όταν έχουμε ένα μακροχρόνιο έργο που δεν θέλουμε να παρακωλύει την εκτέλεση του υπόλοιπου προγράμματος. Μπορούμε να δημιουργήσουμε ένα thread για το μακροχρόνιο έργο και να συνεχίσουμε την εκτέλεση του κύριου προγράμματος. Έτσι, το πρόγραμμά μας μπορεί να εκτελεί πολλές εργασίες ταυτόχρονα, βελτιώνοντας έτσι την απόδοση και την αποκρισιμότητά του.

Ωστόσο, πρέπει να είμαστε προσεκτικοί όταν χρησιμοποιούμε multithreading. Αν δεν διαχειριστούμε σωστά την κοινή πρόσβαση σε κοινούς πόρους, όπως μεταβλητές, μπορούν να προκύψουν προβλήματα ασυνέπειας δεδομένων. Πρέπει να εξασφαλίσουμε ότι τα threads διαχειρίζονται σωστά την κοινή πρόσβαση σε δεδομένα, χρησιμοποιώντας μηχανισμούς όπως τα κλειδώματα (locks) και οι συνθήκες (conditions).

Συνοψίζοντας, το multithreading επιτρέπει σε ένα πρόγραμμα Python να εκτελεί πολλαπλές διεργασίες ταυτόχρονα, βελτιώνοντας την απόδοση και την αποκρισιμότητά του. Χρησιμοποιούμε τη βιβλιοθήκη `threading` για να δημιουργήσουμε και να ελέγξουμε τα threads. Πρέπει να είμαστε προσεκτικοί κατά τη χρήση του multithreading για να αποφύγουμε προβλήματα ασυνέπειας δεδομένων.

Παράδειγμα single thread

```
# Παράδειγμα εκτέλεσης προγράμματος χωρίς multithreading
import time

# Δημιουργία μεταβλητής μεγέθους λίστας και λίστας
list_size = 500
list_of_ints = []

# Γέμισμα λίστας
for i in range(0, (list_size+1))
    list_of_ints.append(i)

# Συνάρτηση εκτύπωσης λίστας σε ένα thread
def single_print():
    for k in range(int(list_size+1)):
        print(list_of_ints[k])

# Αρχή καταμέτρησης χρόνου
singlethread_start_time = time.time()

# Εκτύπωση
single_print()

# Τέλος καταμέτρησης χρόνου
```

```
singlethread_end_time = time.time()


# Υπολογισμός χρόνου εκτέλεσης

singlethread_final_time = singlethread_end_time -
singlethread_start_time


print(f"\nΧρόνος εκτέλεσης συνάρτησης single thread:
{singlethread_final_time}")
```

Παράδειγμα multithread

```
# Παράδειγμα εκτέλεσης προγράμματος με multithreading

import time

import threading as th


# Δημιουργία μεγέθους λίστας και λίστας

list_size = 500

my_list = []


# Γέμισμα λίστας

for i in range(1, (list_size+1)):

    my_list.append(i)


# Συνάρτηση εκτύπωσης λίστας με παραμέτρους στη range()

def multi_print(offset, max_value):

    # offset = το τέλος του προηγούμενου thread

    # max_value = το τέλος του τρέχοντος thread


    # Εκτύπωση συγκεκριμένων κομματιών της λίστας
```

```
    for k in range((0+offset), max_value):
        print(f"{my_list[k]} ")

# Δημιουργία 4 threads (νημάτων) με ονόματα t1, t2, t3, t4 και με παραμέτρους:
# target = είναι η συνάρτηση που θέλουμε να εκτελεί το thread,
# args = τα ορίσματα που θέλουμε να περάσει το thread στη συνάρτηση
t1 = th.Thread(target=multi_print, args=(0, 125))
t2 = th.Thread(target=multi_print, args=(125, 250))
t3 = th.Thread(target=multi_print, args=(250, 375))
t4 = th.Thread(target=multi_print, args=(375, 500))

# Έναρξη καταμέτρησης χρόνου
start = time.time()

# Χωρίς start() δεν ξεκινούν τα threads
# Εκκίνηση των threads
t1.start()
t2.start()
t3.start()
t4.start()

# Αναμονή εκτέλεσης των threads για συνέχεια του προγράμματος και
# ολοκλήρωση των threads πριν το τέλος του προγράμματος
t1.join()
t2.join()
t3.join()
t4.join()

# Τέλος καταμέτρησης χρόνου
end = time.time()
```

print(f"Thread time = {end-start} ")

ΚΑΛΗ ΜΕΛΕΤΗ