ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΛΟΠΟΝΝΗΣΟΥ
ΣΧΟΛΗ ΜΗΧΑΝΙΚΩΝ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

" ΕΦΑΡΜΟΓΕΣ ΤΗΣ ΓΛΩΣΣΑΣ PYTHON ΣΕ ΚΑΤΑΝΕΜΗΜΗΜΕΝΑ ΣΥΣΤΗΜΑΤΑ"

ΦΟΙΤΗΤΕΣ
Νεκτάριος Παναγόπουλος, AM: 2548
Σταύρος Κοντογιάννης, AM: 15137

ΕΠΙΒΛΕΠΩΝ: Βασίλειος Ταμπακάς, Καθηγητής

Πάτρα, 20 Μαΐου 2021

Εγκρίθηκε από την εξεταστική επιτροπή

Πάτρα, 20/5/21

Επιτροπή αξιολόγησης

1. Βασίλειος Ταμπακάς

Υπεύθυνη Δήλωση Φοιτητών

Βεβαιώνουμε ότι είμαστε συγγραφείς αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχαμε για την προετοιμασία της είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης έχουμε αναφέρει τις όποιες πηγές από τις οποίες κάναμε χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Επίσης βεβαιώνουμε ότι αυτή η εργασία προετοιμάστηκε από εμάς ειδικά για τη συγκεκριμένη εργασία.

Η έγκριση της διπλωματικής εργασίας από το Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών του Πανεπιστημίου Πελοποννήσου δεν υποδηλώνει απαραιτήτως και αποδοχή των απόψεων των συγγραφέων εκ μέρους του Τμήματος.

Η παρούσα εργασία αποτελεί πνευματική ιδιοκτησία του Νεκτάριου Παναγόπουλου και του Σταύρου Κοντογιάννη που την εκπόνησαν. Στο πλαίσιο της πολιτικής ανοικτής πρόσβασης οι συγγραφείς/δημιουργοί εκχωρούν στο Πανεπιστήμιο Πελοποννήσου, μη αποκλειστική άδεια χρήσης του δικαιώματος αναπαραγωγής, προσαρμογής, δημόσιου δανεισμού, παρουσίασης στο κοινό και ψηφιακής διάχυσής τους διεθνώς, σε ηλεκτρονική μορφή και σε οποιοδήποτε μέσο, για διδακτικούς και ερευνητικούς σκοπούς, άνευ ανταλλάγματος και για όλο το χρόνο διάρκειας των δικαιωμάτων πνευματικής ιδιοκτησίας. Η ανοικτή πρόσβαση στο πλήρες κείμενο για μελέτη και ανάγνωση δεν σημαίνει καθ' οιονδήποτε τρόπο παραχώρηση δικαιωμάτων διανοητικής ιδιοκτησίας των συγγραφέων/δημιουργών ούτε επιτρέπει την αναπαραγωγή, αναδημοσίευση, αντιγραφή, αποθήκευση, πώληση, εμπορική χρήση, μετάδοση, διανομή, έκδοση, εκτέλεση, «μεταφόρτωση» (downloading), «ανάρτηση» (uploading), μετάφραση, τροποποίηση με οποιονδήποτε τρόπο, τμηματικά ή περιληπτικά της εργασίας, χωρίς τη ρητή προηγούμενη έγγραφη συναίνεση των συγγραφέων/δημιουργών. Οι συγγραφείς/δημιουργοί διατηρούν το σύνολο των ηθικών και περιουσιακών τους δικαιωμάτων.

Περίληψη Διπλωματικής Εργασίας

Η διπλωματική εργασία αναφέρεται στη Γλώσσα Python, καθώς και στις εφαρμογές της στα κατανεμημένα συστήματα.
Αρχικά γίνεται μια εισαγωγή στη γλώσσα προγραμματισμού python όπου περιγράφονται τα βασικά σημεία της όπως δομές ελέγχου, δομές δεδομένων (πλειάδες, σύνολα, λεξικά κ.λ.π.) δίνοντας κατάλληλα παραδείγματα.
Στη συνέχεια μελετάται ο παραλληλισμός με νήματα και παρουσιάζονται τα νήματα όπως αυτά υλοποιούνται στην Python. Πιο συγκεκριμένα αναλύονται τεχνικές αμοιβαίου αποκλεισμού όπως σημαφόροι και κλειδώματα, τεχνικές χρονισμού κ.λ.π. και γίνεται μελέτη στο πρόβλημα παραγωγού-καταναλωτή, καθώς επίσης και στον παραλληλισμό με διεργασίες όπου περιγράφονται τα βασικά τους χαρακτηριστικά, όπως το πως δημιουργούνται και το πως εκτελούνται. Επίσης παρουσιάζονται οι διαφορές μεταξύ νημάτων και διεργασιών.
Επιπρόσθετα γίνεται μια αναλυτική παρουσίαση των γράφων τόσο θεωρητικά όσο και πρακτικά μέσω κατάλληλων κωδίκων σε Python. Ακόμα, παρουσιάζονται προγράμματα σε Python για απεικόνιση κορυφών και ακμών γράφου, για προσθήκη κορυφών και ακμών σε γράφο κ.λ.π.
Παρακάτω γίνεται επισκόπηση προγραμμάτων σε Python για παραλληλισμό πράξεων με μεγαλύτερο δυνατό βαθμό, για εκτέλεση νημάτων με κατάλληλο συγχρονισμό και για την υλοποίηση μιας εφαρμογής νημάτων που τοποθετούνται σε ουρά και εκτελούνται τυχαία.
Επιπλέον περιγράφονται τα sockets και συγκεκριμένα τα tcp και τα udp sockets, καθώς και οι διαφορές ανάμεσα τους. Επιδεικνύεται η δημιουργία tcp socket τόσο στο server όσο και στον client. Αντίστοιχα επιδεικνύεται η δημιουργία udp socket τόσο στο server όσο και στον client και παρουσιάζεται ένα πλήρες παράδειγμα με υλοποίηση ενός echo server και client.
Τέλος περιγράφονται και υλοποιούνται με κώδικα python οι αλγόριθμοι συγχρονισμού ρολογιού Cristian και Berkeley. Επίσης υλοποιούνται σε κάθε ένα από αυτούς ένας master clock server και ένας clock client αντίστοιχα.

Περιεχόμενα

1 ΒΑΣΙΚΑ ΧΑΡΑΚΤΗΡΙΣΤΙΚΑ PYTHON....6

1.1 ΤΡΟΠΟΙ ΔΙΕΡΜΗΝΕΙΑΣ ΚΩΔΙΚΑ PYTHON....6
1.2 ΜΕΤΑΒΛΗΤΕΣ....6
1.3 ΕΝΤΟΛΕΣ ΕΙΣΟΔΟΥ/ΕΞΟΔΟΥ....7
1.3.1 Εντολή Εξόδου Print....7
1.3.2 Εντολή Εισόδου Input....7
1.4 ΤΕΛΕΣΤΕΣ ΠΡΑΞΕΩΝ....8
1.4.1 Boolean (Λογικοί) Τελεστές....8
1.5 ΣΥΜΒΟΛΟΣΕΙΡΕΣ (STRINGS)....9
1.5.1 Πράξεις Συμβολοσειρών....12
1.6 ΔΟΜΕΣ ΕΛΕΓΧΟΥ ΣΤΗΝ PYTHON....13
1.6.1 Δομή Επιλογής - Εντολή if....13
1.6.2 Δομές Επανάληψης- Εντολή for....13
1.6.3 Δομές Επανάληψης- Εντολή while....15
1.7 ΠΛΕΙΑΔΕΣ (TUPLES)....16
1.8 ΣΥΝΟΛΑ (SETS)....18
1.9 ΛΙΣΤΕΣ (LISTS)....20
1.10 ΣΥΝΑΡΤΗΣΕΙΣ (FUNCTIONS)....22
1.11 ΛΕΞΙΚΑ (DICTIONARIES)....24
1.11.1 Πίνακες (Arrays)....25
1.12 ΑΝΑΔΡΟΜΗ ΣΕ PYTHON....26
1.13 ΚΛΑΣΕΙΣ/ΑΝΤΙΚΕΙΜΕΝΑ (CLASSES/OBJECTS)....28
1.14 ΚΛΗΡΟΝΟΜΙΚΟΤΗΤΑ ΣΤΗΝ PYTHON....31

2 ΠΑΡΑΛΛΗΛΙΣΜΟΣ ΜΕ ΝΗΜΑΤΑ....36

2.1 ΚΛΕΙΔΩΜΑΤΑ (LOCKS)....38
2.2 RLOCKS....40
2.3 ΣΗΜΑΦΟΡΟΙ (SEMAPHORES)....42
2.4 ΣΥΜΒΑΝΤΑ (EVENTS)....46
2.5 ΣΥΝΘΗΚΕΣ (CONDITIONS)....48
2.6 ΦΡΑΓΜΑΤΑ (BARRIERS)....50
2.7 ΧΡΟΝΙΣΤΕΣ (TIMERS)....52
2.8 ΧΑΡΑΚΤΗΡΙΣΤΙΚΑ ΠΡΟΒΛΗΜΑΤΑ ΣΥΓΧΡΟΝΙΣΜΟΥ ΜΕ ΝΗΜΑΤΑ....53
2.8.1 Πρόβλημα Παραγωγού-Καταναλωτή (Producer-Consumer) Problem....53

3 ΠΑΡΑΛΛΗΛΙΣΜΟΣ ΜΕ ΔΙΕΡΓΑΣΙΕΣ....56

3.1 ΔΗΜΙΟΥΡΓΙΑ ΔΙΕΡΓΑΣΙΑΣ....59
3.2 ΚΑΘΟΡΙΣΜΟΣ ΤΡΕΧΟΥΣΑΣ ΔΙΕΡΓΑΣΙΑΣ....60

4 ΓΡΑΦΟΙ ΣΤΗΝ PYTHON....65

4.1 ΕΝΤΟΠΙΣΜΟΣ ΔΙΕΡΓΑΣΙΑΣ....63
4.2 ΒΑΣΙΚΟΙ ΟΡΙΣΜΟΙ ΓΡΑΦΩΝ....65
4.3 1^{Ο} ΠΑΡΑΔΕΙΓΜΑ ΔΗΜΙΟΥΡΓΙΑΣ ΓΡΑΦΟΥ....66
4.4 ΟΙ ΓΡΑΦΟΙ ΩΣ ΜΙΑ ΚΛΑΣΗ ΤΙΣ PYTHON....68
4.5 2^{Ο} ΠΑΡΑΔΕΙΓΜΑ ΔΗΜΙΟΥΡΓΙΑΣ ΓΡΑΦΟΥ....72
4.5.1 Απεικόνιση Κορυφών Γράφου....73
4.5.2 Απεικόνιση Ακμών Γράφου....73
4.5.3 Προσθήκης Νέας Κορυφής στο Γράφο....75
4.5.4 Προσθήκης Νέας Ακμής στο Γράφο....76
4.6 ΜΟΝΟΠΑΤΙΑ ΣΤΗΝ PYTHON.....77

5 ΑΣΚΗΣΕΙΣ ΜΕ ΝΗΜΑΤΑ ΚΑΙ ΣΗΜΑΦΟΡΟΥΣ....81

5.1 ΆΣΚΗΣΗ 1....81
5.2 ΆΣΚΗΣΗ 2....86
5.3 ΆΣΚΗΣΗ 3....88
5.4 ΆΣΚΗΣΗ 4....90

6 ΕΡΓΑΣΤΗΡΙΑΚΗ ΆΣΚΗΣΗ 2 ΜΕ ΓΡΑΦΟΥΣ....96

6.1 ΔΙΑΔΙΚΑΣΙΑ ΔΗΜΙΟΥΡΓΙΑΣ ΓΡΑΦΟΥ ΣΤΗΝ PYTHON ... 96
6.2 ΓΕΝΙΚΟΙ ΓΡΑΦΟΙ ΣΤΗΝ PYTHON ... 98

7 SOCKETS ΣΤΗΝ PYTHON ... 101

7.1 ΠΕΡΙΓΡΑΦΗ SOCKET ... 101
7.2 TCP ΚΑΙ UDP SOCKETS ... 101
7.3 ΔΗΜΙΟΥΡΓΙΑ TCP SOCKET ΔΙΑΚΟΜΙΣΤΗ (SERVER) ... 103
7.4 ΔΗΜΙΟΥΡΓΙΑ TCP SOCKET ΠΕΛΑΤΗ (CLIENT) ... 106
7.5 UDP SERVER ... 107
7.6 UDP CLIENT ... 108
7.7 ΠΑΡΑΔΕΙΓΜΑ ECHO CLIENT AND SERVER ... 109
7.7.1 Υλοποίηση Echo Server ... 109
7.7.2 Υλοποίηση Echo Client ... 111
7.7.3 Εκτέλεση Echo Server και Client ... 111

8 ΕΡΓΑΣΤΗΡΙΑΚΗ ΆΣΚΗΣΗ 3 ΜΕ ΚΑΤΑΝΕΜΗΜΕΝΑ ΡΟΛΟΓΙΑ ... 112

8.1 ΑΛΓΟΡΙΘΜΟΣ BERKELEY ... 112
8.1.1 Ψευδοκώδικας Τελευταίου Βήματος Αλγορίθμου Berkeley ... 113
8.1.2 Master Clock server ... 115
8.1.3 Clock Client ... 119
8.2 ΑΛΓΟΡΙΘΜΟΣ CRISTIAN ... 121
8.2.1 Περιγραφή Αλγορίθμου Cristian ... 121
8.2.2 Λειτουργία/Αξιοπιστία του προηγούμενου τύπου ... 122
8.2.3 Master Clock server ... 123
8.2.4 Clock Client ... 124
8.2.5 Βελτίωση στο Συγχρονισμό Ρολογιού ... 125

9 ΒΙΒΛΙΟΓΡΑΦΙΑ ... 126

1 Βασικά Χαρακτηριστικά Python

Η γλώσσα Python είναι μία ισχυρή γλώσσα προγραμματισμού αρκετά εύκολη στην εκφώνηση της. Χρησιμοποιεί δομές δεδομένων υψηλού επιπέδου αλλά συνδυάζει και τις βασικές αρχές του αντικειμενοστραφούς προγραμματισμού. Επιτρέπει τη δημιουργία εφαρμογών σε πολλούς τομείς και στις περισσότερες πλατφόρμες. Τα βασικά χαρακτηριστικά της Python είναι τα ακόλουθα:

- Είναι γλώσσα υψηλού επιπέδου
- Φορητή γλώσσα. Αυτό σημαίνει ότι μπορεί να δουλέψει σε οποιαδήποτε πλατφόρμα χωρίς αλλαγές.
- Είναι γλώσσα ανοικτού κώδικα. Αυτό σημαίνει ότι μπορούμε να διαβάσουμε τον πηγαίο ελεύθερα τον πηγαίο κώδικα, να κάνουμε αλλαγές σε αυτόν, να δημιουργήσουμε αντίγραφα αυτού κλπ.
- Απλή στην εκμάθηση και επεκτάσιμη. Για παράδειγμα μπορούμε ένα τμήμα κώδικα να το προγραμματίσουμε σε άλλη γλώσσα (πχ σε c++) και στη συνέχεια να χρησιμοποιήσουμε αυτά τα τμήματα κώδικα σε πρόγραμμα python.

1.1 Τρόποι διερμηνείας κώδικα Python

Τα προγράμματα της python μπορούν να μεταφράζονται είτε από ένα μεταφραστή (compiler) είτε από ένα διερμηνευτή (interpreters) .Ο μεταφραστής λαμβάνει όλο το πηγαίο κώδικα και τον ελέγχει στο σύνολο του για συντακτικά λάθη ενώ ο διερμηνευτής λαμβάνει μία-μία γραμμή του πηγαίου κώδικα και τη μεταφράζει.

1.2 Μεταβλητές

Σε αντίθεση με άλλες γλώσσες προγραμματισμού, η Python δεν δηλώνει μεταβλητές Μια μεταβλητή δημιουργείται-δηλώνεται τη στιγμή που εκχωρείται μια τιμή σε αυτήν. Οι μεταβλητές δεν χρειάζεται να δηλώνονται σε κάποιο συγκεκριμένο τύπο και μπορούν να αλλάζουν τύπο μετά τη δημιουργία τους Το όνομα μιας μεταβλητής μπορεί να περιλαμβάνει λατινικούς χαρακτήρες ή αριθμούς αλλά δεν μπορεί να αρχίζει με αριθμό.

Οι βασικοί τύποι δεδομένων στην Python είναι:
<int> ακέραιος
<float> πραγματικός
<str> συμβολοσειρά

Παράδειγμα 1

- x = 5 →*η μεταβλητή x δηλώνεται αυτόματα τύπου int*
- y = "John" → *η μεταβλητή y δηλώνεται αυτόματα τύπου string*
- *print*(x) →*τυπώνεται το περιεχόμενο της x δηλ. το 5*
- *print*(y) →*τυπώνεται το περιεχόμενο της y δηλ. το John*
- x = 6.5 →*η μεταβλητή x μετατρέπεται αυτόματα σε float*
- *print*(x) →*τυπώνεται το περιεχόμενο της x δηλ. το 6.5*

Παράδειγμα 2

Για να χρησιμοποιήσουμε μια μεταβλητή, χρειαζόμαστε απλά να γνωρίζουμε το όνομα της. Για παράδειγμα το ακόλουθο πρόγραμμα υψώνει την μεταβλητή pi στο τετράγωνο (αφού την έχουμε αρχικοποιήσει με την τιμή 3.14) και στη συνέχεια τυπώνει το αποτέλεσμα της με την εντολή print

- pi = 3.14 →*Ο τελεστής = κάνει εκχώρηση σε μεταβλητή*
- *print*(pi**2) →*Ο τελεστής ** είναι τελεστής ύψωσης σε δύναμη στην Python*

1.3 Εντολές Εισόδου/Εξόδου

1.3.1 Εντολή Εξόδου Print

Η εντολή print() εμφανίζει το περιεχόμενο μιας μεταβλητής ή ένα μήνυμα. Για παράδειγμα:

- η εντολή *print*(a) τυπώνει το περιεχόμενο της μεταβλητής a,
- η εντολή *print*('Καλημέρα') τυπώνει το μήνυμα Καλημέρα

1.3.2 Εντολή Εισόδου Input

Η εντολή input() εμφανίζει ένα μήνυμα που είναι στην παρένθεση και περιμένει να δώσουμε μια είσοδο. Μετά καταχωρεί ότι εισάγαμε στη μεταβλητή που βρίσκεται αριστερά του τελεστή =. Για παράδειγμα:

- η εντολή name=*input*('Δώσε το όνομα σου') μας εμφανίζει το μήνυμα 'Δώσε το όνομα σου' και καταχωρεί στη μεταβλητή name το αλφαριθμητικό που εισάγαμε

Παράδειγμα με εντολές Input και Print

```
answer = int(input('What is the product of 2 * 5: '))

if (answer == 10):
    print('Correct!')

print('Exiting program')
```

Στο παράδειγμα αυτό η εντολή input μας επιτρέπει να εισάγουμε δεδομένα εμφανίζοντας σχετικό μήνυμα. Με τη συνάρτηση int μετατρέπουμε το δεδομένο εισόδου από αλφαριθμητικό σε int και το καταχωρούμε στη μεταβλητή answer. Μετά ελέγχουμε με τον τελεστή == αν είναι ίση με 10 και τυπώνουμε το μήνυμα 'Correct'.

1.4 Τελεστές πράξεων

- + Πρόσθεση
- - Αφαίρεση
- * Πολλαπλασιασμός
- / Διαίρεση π.χ. 5/3=1.66
- // Ακέραια διαίρεση (πηλίκο) π.χ. 5//1
- % υπόλοιπο διαίρεσης π.χ. 5%3=2
- ** Ύψωση σε δύναμη π.χ. 5**3=125

1.4.1 Boolean (Λογικοί) Τελεστές

Οι βασικοί τελεστές για εκφράσεις Boolean είναι

- not (άρνηση)
- or (διάζευξη)
- and (σύζευξη)

1.5 Συμβολοσειρές (Strings)

Μια συμβολοσειρά στην Python είναι μια ακολουθία χαρακτήρων που περικλείεται σε μονά ή διπλά εισαγωγικά. Το 'hello' είναι το ίδιο με το "hello". Τα αλφαριθμητικά μπορούν να εμφανιστούν στην οθόνη χρησιμοποιώντας την εντολή print. Για παράδειγμα print("hello") τυπώνεται στην οθόνη η λέξη hello. Η Python δεν έχει τύπο δεδομένων χαρακτήρα, ένας χαρακτήρας είναι μια συμβολοσειρά μήκους 1. Οι αγκύλες [] μπορούν να χρησιμοποιηθούν για την πρόσβαση στα στοιχεία μιας συμβολοσειράς.

Παράδειγμα 1 – Αποκοπή τμήματος αλφαριθμητικού

a = "abcdefghi"

0	1	2	3	4	5	6	7	8
a	b	c	d	e	f	g	h	i

a[:4] →*τυπώνεται 'abcd', δηλαδή τυπώνονται οι χαρακτήρες από τον 0ο μέχρι τον 3ο .Δεν συμπεριλαμβάνεται ο 4ος κατά σειρά χαρακτήρας*

a[4:] →*τυπώνεται 'efghi' δηλαδή τυπώνονται οι χαρακτήρες από τον 4ομέχρι τέλους. Συμπεριλαμβάνεται ο 4ος κατά σειρά χαρακτήρας*

a[4:7] →*τυπώνεται 'efg', δηλαδή τυπώνονται οι χαρακτήρες από τον 4ο μέχρι τον 6ο. Δεν συμπεριλαμβάνεται ο 7ος κατά σειρά χαρακτήρας*

>>>word='Python'

0	1	2	3	4	5
P	y	t	h	o	n
-6	-5	-4	-3	-2	-1

>>>word[0] *#ο χαρακτήρας του αλφαριθμητικού Python στη θέση 0*

'P'

>>>word[5] *#ο χαρακτήρας του αλφαριθμητικού Python στη θέση 5*

'n'

Οι δείκτες μπορεί να είναι αρνητικοί αριθμοί. Σε αυτή την περίπτωση αρχίζουν να μετρούν από τα δεξιά προς τα αριστερά:

>>>word[-1] *#ο τελευταίος χαρακτήρας του αλφαριθμητικού Python*

'n'

>>>word[-2] *#ο προτελευταίος χαρακτήρας του αλφαριθμητικού Python*

′o′

>>>word[-6] #*ο αρχικός χαρακτήρας του αλφαριθμητικού Python*
′P′

Παράδειγμα 2 – Υποσυμβολοσειρές-Τεμαχισμός

b="Hello, World!" →*το b μετατρέπεται αυτόματα σε πίνακα χαρακτήρων*

print(b[2:5]) →*Τυπώνονται* οι χαρακτήρες *llo*. *Με τον επόμενο κώδικα λαμβάνονται οι χαρακτήρες από τη θέση 2 έως τη θέση 5 του αλφαριθμητικού b (η θέση 5 δεν συμπεριλαμβάνεται):*

>>> word='Python'
word[0:2] →*οι χαρακτήρες από τη θέση 0 (συμπεριλαμβάνεται) μέχρι τη θέση 2 (εξαιρείται)*

- 'Py'

>>>word[2:5] → *οι χαρακτήρες από τη θέση 2 (συμπεριλαμβάνεται) μέχρι τη θέση 5 (εξαιρείται)*

- 'tho'

Τονίζουμε πως η αρχή περιλαμβάνεται πάντα ενώ το τέλος εξαιρείται. Αυτό εξασφαλίζει ότι s[: i] + s[i:] είναι ίσο με s

>>>word[:2]+ word[2:]

- 'Python'

>>>word[:4]+ word[4:]

- 'Python'

Ένα τμήμα δεικτών έχει χρήσιμες προεπιλογές. Είναι δυνατόν να παραλείπεται το πρώτο στοιχείο, όταν η τιμή του δείκτη ισούται με μηδέν ή να παραλείπεται το τέλος αυτού αφού το μέγεθος της συμβολοσειράς είναι σταθερό.

>>>word[:2] → *οι χαρακτήρες από τη θέση 0 (συμπεριλαμβάνεται) μέχρι τη θέση 2 (εξαιρείται)*

- 'Py'

>>> word[4:] → *οι χαρακτήρες από τη θέση 4 (συμπεριλαμβάνεται) μέχρι το τέλος*

- 'on'

>>>word[-2:] →*οι χαρακτήρες από την προτελευταία θέση (συμπεριλαμβάνεται) μέχρι το τέλος*

- 'on'

Εάν χρειαζόμαστε μια διαφορετική συμβολοσειρά, θα πρέπει να δημιουργήσουμε μια νέα.

>>>'J' + word[1:]

'Jython'

>>>word[:2] + 'py'

- 'Pypy'

Παράδειγμα 3 – Συνάρτηση strip

Με τη συνάρτηση strip() αφαιρούνται όλα τα κενά διαστήματα από την αρχή και το τέλος ενός αλφαριθμητικού

a = "   Hello, World!   "

print(a.*strip*()) →*επιστρέφει "Hello, World!" έχοντας διαγράψει όλα τα κενά από την αρχή και το τέλος της συμβολοσειράς*

Παράδειγμα 4 – Συνάρτηση len

Με τη συνάρτηση len() υπολογίζεται το μήκος ενός αλφαριθμητικού

a = "Hello, World!"

print(*len*(a)) →*τυπώνεται το μήκος της συμβολοσειράς που είναι 13 χαρακτήρες*

Παράδειγμα 5 – Συνάρτηση lower

Η συνάρτηση lower() επιστρέφει το αλφαριθμητικό με πεζούς χαρακτήρες

a = "Hello, World!"

print(a.*lower*())→*τυπώνεται το αλφαριθμητικό hello, world!*

1.5.1 Πράξεις Συμβολοσειρών

Οι συμβολοσειρές αποτελούν αμετάβλητες ακολουθίες χαρακτήρων και δεν επιτρέπεται η αλλαγή τους. Επιτρέπεται όμως η πρόσθεση συμβολοσειρών όπως και ο πολλαπλασιασμός συμβολοσειράς με ακέραιο.

τελεστής	αποτέλεσμα
<seq> + <seq>	συνένωση
<seq> * <int>	επανάληψη
<seq>[]	δείκτης
len(<seq>)	μήκος ακολουθίας
<seq>[:]	τεμαχισμός
for <var> in <seq>:	επανάληψη
<expr> in <seq>	συμμετοχή (Boolean)

Παράδειγμα 1 – Συνένωση Συμβολοσειρών

```
>>> a = 'Καλή '
>>> b = 'μέρα'
>>> c = a + b
>>> print (c)
```
→*Τυπώνεται Καλημέρα*

Παράδειγμα 2– Επανάληψη Συμβολοσειράς

Μπορούμε να πολλαπλασιάσουμε ένα string με αριθμό.

```
>>> a = ' Ηχώ '
>>> b = a * 5
>>> print (b)
```

ΗχώΗχώΗχώΗχώΗχώ

1.6 Δομές Ελέγχου στην Python

Υπάρχουν δύο δομές ελέγχου στην Python: Οι δομές επιλογής και οι δομές επανάληψης

1.6.1 Δομή Επιλογής - Εντολή if

```
x=input("Δώστε αριθμό")
if x<0:
    print('Αρνητικός Αριθμός')
elif x==0:
    print('Μηδέν')
else:
    print('Θετικός')
```

1.6.2 Δομές Επανάληψης- Εντολή for

Παράδειγμα 1

```
>>>li=[1, 3, 5, 7, 9]
>>>for i in li:
       print(i)
```

<u>**Αποτελέσματα Εκτέλεσης**</u>

- 1
- 3
- 5
- 7
- 9

```
>>>for i in li:
       print(i, li.index(i)))
```

<u>**Αποτελέσματα Εκτέλεσης**</u>

- 1 0

- 3 1
- 5 2
- 7 3
- 9 4

Παράδειγμα 2

```
>>>onoma=['John', 'Roger', 'Natalie', 'Tamara']
>>>for a in onoma:
        print(a, len(a))
```

- John 4
- Roger 5
- Natalie 7
- Tamara 6

Παράδειγμα 3

fruits = ["apple", "banana", "cherry"]

for x *in* fruits:

print(x) →*τυπώνεται*

apple

banana

cherry

Παράδειγμα 4

for x *in* "banana":

print(x) →*τυπώνονται οι τιμές*

b

a

n

a

n

a

1.6.3 Δομές Επανάληψης- Εντολή while

Να τυπωθούν οι όροι της ακολουθίας Fibonacci

```
>>>#Σειρά Fibonacci →αυτό είναι ένα σχόλιο
>>>a,b=0,1
>>>while b<50:
       print(b)
       a,b= b, a+b
```

- 1
- 1
- 2
- 3
- 5
- 8

Επεξήγηση αποτελεσμάτων

	a	b	Εκτύπωση
Αρχικά	0	1	1
$1^{η}$ επανάληψη	1	1	1
$2^{η}$ επανάληψη	1	2	2
$3^{η}$ επανάληψη	2	3	3 κ.λ.π.

1.7 Πλειάδες (Tuples)

Μια πλειάδα είναι ένα σύνολο στοιχείων ταξινομημένο και ΜΗ τροποποιήσιμο. Οι πλειάδες συμβολίζονται με ().

Παράδειγμα 1-Δημιουργία Πλειάδας

Δημιουργία και εκτύπωση Πλειάδας

thistuple = ("apple", "banana", "cherry")

print(thistuple) → *τυπώνεται ('apple', 'banana', 'cherry')*

Εναλακτικά μια πλειάδα δημιουργείται καλώντας το δημιουργό tuple() όπως δείχνει το επόμενο παράδειγμα

thistuple = *tuple*(("apple", "banana", "cherry"))

print(thistuple) → *τυπώνεται ('apple', 'banana', 'cherry')*

Παράδειγμα 2-Εκτύπωση 2ου στοιχείου Πλειάδας

thistuple = ("apple", "banana", "cherry")

print(thistuple [1]) → *τυπώνεται banana*

ΠΡΟΣΟΧΗ: ΑΠΟ ΤΗ ΣΤΙΓΜΗ ΠΟΥ ΔΗΜΙΟΥΡΓΕΙΤΑΙ ΜΙΑ ΠΛΕΙΑΔΑ ΔΕΝ ΤΡΟΠΟΠΟΙΕΙΑΙ. ΟΙ ΠΛΕΙΑΔΕΣ ΔΕΝ ΤΡΟΠΟΠΟΙΟΥΝΤΑΙ

Παράδειγμα 3-Εκτύπωση Στοιχείων Πλειάδας

Τυπώνονται όλα τα στοιχεία της πλειάδας (tuple)

thistuple =("apple", "banana", "cherry")

for x *in* thistuple:

    print(x) → *τυπώνεται η λίστα apple banana cherry*

Παράδειγμα 4-Εκτύπωση Μήκους Πλειάδας

Τυπώνεται το πλήθος των στοιχείων της πλειάδας

thistuple = ("apple", "banana", "cherry")

print(*len*(thistuple)) → *τυπώνεται η τιμή 3*

Παράδειγμα 5-Προσθήκη στοιχείου στην Πλειάδα

Ο ΑΚΟΛΟΥΘΟΣ ΚΩΔΙΚΑΣ ΕΙΝΑΙ ΛΑΘΟΣ ΓΙΑΤΙ ΤΑ ΣΤΟΙΧΕΙΑ ΤΗΣ ΠΛΕΙΑΔΑΣ ΔΕΝ ΤΡΟΠΟΠΟΙΟΥΝΤΑΙ.

thistuple = ("apple", "banana", "cherry")

thistuple[3] = "orange" →*Αυτό προκαλεί σφάλμα*

print(thistuple)

1.8 Σύνολα (sets)

Ένα σύνολο είναι μια συλλογή στοιχείων χωρίς ταξινόμηση και δεικτοδότηση. Τα σύνολα συμβολίζονται με {}. Σε ένα σύνολο μπορούμε να προσθέσουμε νέα στοιχεία αλλά ΔΕΝ ΜΠΟΡΟΥΜΕ ΝΑ ΤΡΟΠΟΠΟΙΗΣΟΥΜΕ ΥΠΑΡΧΟΝΤΑ ΣΤΟΙΧΕΙΑ ΤΟΥ ΣΥΝΟΛΟΥ.

Παράδειγμα 1-Δημιουργία Συνόλου

Δημιουργία και εκτύπωση Συνόλου

thisset = {"apple", "banana", "cherry"}

print(thisset)

Παράδειγμα 2-Εκτύπωση Στοιχείων Συνόλου

Τυπώνονται όλα τα στοιχεία του συνόλου

thisset = {"apple", "banana", "cherry"}

for x *in* thisset:

   print(x) → *τυπώνεται το σύνολο*

   apple

   banana

   cherry

Παράδειγμα 3-Εκτύπωση Πλήθους στοιχείων Συνόλου

Τυπώνεται το πλήθος των στοιχείων του συνόλου

thisset = {"apple", "banana", "cherry"}

print(*len*(thisset)) → *τυπώνεται η τιμή 3*

Παράδειγμα 4-Προσθήκη νέου στοιχείου και νέων στοιχείων στο σύνολο

Για να προσθέσουμε ένα στοιχείο στο σύνολο χρησιμοποιούμε τη μέθοδο add(). Για να προσθέσουμε πολλά στοιχεία στο σύνολο χρησιμοποιούμε τη μέθοδο update().

Προσθήκη ενός στοιχείου στο σύνολο

thisset = {"apple", "banana", "cherry"}

thisset.*add*("orange")

print(thisset) →*Τυπώνεται {'apple', 'banana', 'orange', 'cherry'}*

Προσθήκη πολλών στοιχείων στο σύνολο

thisset = {"apple", "banana", "cherry"}

thisset.*update*(["orange", "mango", "grapes"])

print(thisset) →*Τυπώνεται {'orange', 'mango', 'cherry', 'grapes', 'banana', 'apple'}*

Παράδειγμα 5-Διαγραφή συγκεκριμένου στοιχείου Συνόλου

Η μέθοδος remove() διαγράφει ένα συγκεκριμένο στοιχείο του συνόλου

thisset = {"apple", "banana", "cherry"}

thisset.*remove*("banana")

print(thisset) → *τυπώνεται ['apple', 'cherry']*

1.9 Λίστες (Lists)

Μια λίστα είναι ένα σύνολο στοιχείων ταξινομημένο και τροποποιήσιμο. Οι λίστες συμβολίζονται με []. Η λίστα ομαδοποιεί δεδομένα και είναι ένας κατάλογος τιμών που διαχωρίζονται με κόμματα και όλες βρίσκονται μέσα σε []. Δεν είναι υποχρεωτικό όλες οι τιμές της λίστας να είναι του ίδιου τύπου π.χ. a=['London','Rome', 1452, 9]. Κάθε μέλος μιας λίστας έχει μια θέση μέσα σε αυτή. Η θέση του $1^{ου}$ μέλους είναι η 0. π.χ.

>>>a[0]

- 'London'

>>>a[2]

- 1452

Τα μέλη μιας λίστας μπορεί να είναι διαφορετικού τύπου, η Python όμως τα αντιμετωπίζει το καθένα με τον τύπο του. Στο προηγούμενο παράδειγμα η λίστα έχει τα δύο πρώτα μέλη της (London, Rome) τύπου string και τα δύο επόμενα (1451, 9) τύπου int.

>>> a[0]+a[1]

- 'LondonRome'

>>> a[2]+α[3]

- 1461

Παράδειγμα 1-Δημιουργία Λίστας

Δημιουργία και εκτύπωση Λίστας

thislist = ["apple", "banana", "cherry"]

print(thislist) → *τυπώνεται ['apple', 'banana', 'cherry']*

Παράδειγμα 2-Εκτύπωση 2ου στοιχείου Λίστας

thislist = ["apple", "banana", "cherry"]

print(thislist[1]) → *τυπώνεται banana*

Παράδειγμα 3-Τροποποίηση 2ου στοιχείου Λίστας

thislist = ["apple", "banana", "cherry"]

thislist[1] = "blackcurrant"→*τροποποιούμε το στοιχείο στη θέση 1 θέτοντας του τιμή blackcurrant*

print(thislist) → *τυπώνεται ['apple', 'blackcurrant', 'cherry']*

Παράδειγμα 4-Εκτύπωση Στοιχείων Λίστας

Τυπώνονται όλα τα στοιχεία της λίστας

thislist = ["apple", "banana", "cherry"]

for x *in* thislist:

  print(x) → *τυπώνεται η λίστα*

apple

banana

cherry

Παράδειγμα 5-Εκτύπωση Μήκους Λίστας

Υπολογίζεται το πλήθος των στοιχείων της λίστας.

thislist = ["apple", "banana", "cherry"]

print(*len*(thislist)) → *τυπώνεται η τιμή 3*

+Με δεδομένη τη λίστα ['a', 'b', 3, 'd', 1, 'a'] εκτελώντας τη συνάρτηση len() σε αυτή θα πάρουμε τα ακόλουθα̃:

>>>*len*(a)

- 6

Η len είναι μια συνάρτηση που έχει εφαρμογή σε όλα τα αντικείμενα της Python

>>>k='asdfghjkl'

>>>*len*(k)

- 9

1.10 Συναρτήσεις (Functions)

Στην Python μια συνάρτηση δηλώνεται με την εντολή def:

Παράδειγμα 1-Δημιουργία και Κλήση Συνάρτησης

Η δημιουργία μιας συνάρτησης γίνεται με την εντολή def ενώ η κλήση της γίνεται γράφοντας το όνομα της ακολουθούμενο από παρενθέσεις:

Δήλωση-Ορισμός Συνάρτησης

def my_function():
    print("Hello from a function")

Κλήση συνάρτησης

my_function() → *έτσι γίνεται η κλήση της συνάρτησης με όνομα my_function και τυπώνεται το μήνυμα Hello from a function*

Παράδειγμα 2-Παράμετροι Συνάρτησης

def my_function(fname):
    print(fname + " Refsnes")

my_function("Emil") → *καλείται η συνάρτηση my_function με όρισμα την τιμή Emil και τυπώνεται το μήνυμα Emil Refsnes*

my_function("Tobias") → *καλείται η συνάρτηση my_function με όρισμα την τιμή Tobias και τυπώνεται το μήνυμα Tobias Refsnes*

my_function("Linus") → *καλείται η συνάρτηση my_function με όρισμα την τιμή Linus και τυπώνεται το μήνυμα Linus Refsnes*

Παράδειγμα 3-Εξορισμού Παράμετροι Συνάρτησης

Αν καλέσουμε μια συνάρτηση χωρίς παραμέτρους τότε θα ισχύουν οι προεπιλεγμένες τιμές των παραμέτρων

def my_function(country = "Norway"):

    print("I am from " + country)

my_function("Sweden") → *καλείται η συνάρτηση my_function με όρισμα Sweden και τυπώνεται το μήνυμα I am from Sweden*

my_function("India") → *καλείται η συνάρτηση my_function με όρισμα India και τυπώνεται το μήνυμα I am from India*

my_function() → *καλείται η συνάρτηση my_function χωρίς όρισμα και τυπώνεται το μήνυμα I am from Norway*
my_function("Brazil") → *καλείται η συνάρτηση my_function με όρισμα Brazil και τυπώνεται το μήνυμα I am from Brazil*

Παράδειγμα 4-Μεταβίβαση Λίστας ως Παράμετρο Συνάρτησης

```
def my_function(food):
  for x in food:
    print(x)


fruits = ["apple", "banana", "cherry"]
```

my_function(fruits) → *καλείται η συνάρτηση my_function με όρισμα τη λίστα fruits και τυπώνεται η ακόλουθη λίστα:*

apple

banana

cherry

Παράδειγμα 5-Επιστροφή Τιμής από συνάρτηση

```
def my_function(x):
  return 5*x
```

print(my_function(3)) →*καλείται η συνάρτηση my_function με όρισμα την τιμή 3. Αυτή υπολογίζει και επιστρέφει το 15 που τυπώνεται*

print(my_function(5)) →*καλείται η συνάρτηση my_function με όρισμα την τιμή 5. Αυτή υπολογίζει και επιστρέφει το 25 που τυπώνεται*

print(my_function(9)) →*καλείται η συνάρτηση my_function με όρισμα την τιμή 9. Αυτή υπολογίζει και επιστρέφει το 45 που τυπώνεται*

1.11 Λεξικά (Dictionaries)

Ένα λεξικό (dictionary) είναι μια συλλογή στοιχείων μη ταξινομημένη, τροποποιήσιμη και δεικτοδοτημένη. Τα λεξικά στην Python συμβολίζονται με άγκιστρα και έχουν κλειδιά και τιμές.

Παράδειγμα 1-Δημιουργία Λεξικού

Δημιουργία και εκτύπωση Λεξικού

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

print(thisdict) → *Τυπώνεται {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}*

Παράδειγμα 2-Λήψη στοιχείων Λεξικού

Μπορούμε να προσπελάσουμε τα στοιχεία ενός λεξικού γράφοντας όνομα λεξικού[πεδίο] όπως στο επόμενο παράδειγμα όπου θέλουμε να πάρουμε το κλειδί model:

x = thisdict["model"]
print(x) →*Τυπώνεται Mustang*

Εναλλακτικά μπορούμε να χρησιμοποιήσουμε τη μέθοδο get() για να πάρουμε το ίδιο αποτέλεσμα όπως στο επόμενο παράδειγμα:

x = thisdict.*get*("model")
print(x) →*Τυπώνεται Mustang*

Παράδειγμα 3-Αλλαγή Τιμής Λεξικού

Μπορούμε να αλλάξουμε την τιμή ενός στοιχείου μέσω της τιμής του κλειδιού του. Για παράδειγμα θέλουμε να αλλάξουμε το year σε 2018.

```
thisdict = {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
thisdict["year"] = 2018
```

print(thisdict) → *Τυπώνεται {'brand': 'Ford', 'model': 'Mustang', 'year': 2018}*

1.11.1 Πίνακες (Arrays)

Οι πίνακες χρησιμοποιούνται για να αποθηκεύσουν πολλές τιμές σε μια μοναδική μεταβλητή.

Παράδειγμα 1-Κατασκευή Πίνακα

Δημιουργία και εκτύπωση πίνακα
cars = ["Ford", "Volvo", "BMW"]
print(cars) → *Τυπώνεται ['Ford', 'Volvo', 'BMW']*

Παράδειγμα 2-Κατασκευή Πίνακα

Τροποποιείται το 2ο στοιχείο του πίνακα όπως φαίνεται στο επόμενο παράδειγμα:
cars = ["Ford", "Volvo", "BMW"]
cars[0] = "Toyota"
print(cars) → *Τυπώνεται ['Toyota', 'Volvo', 'BMW']*

Παράδειγμα 3-Εκτύπωση Πλήθους στοιχείων Πίνακα

Με τη συνάρτηση len() εκτυπώνουμε το πλήθος των στοιχείων ενός πίνακα όπως φαίνεται στο επόμενο παράδειγμα:
x = *len*(cars)
print(cars) → *Τυπώνεται 3*

Παράδειγμα 4-Εκτύπωση Πίνακα

for x *in* cars:
    print(x) → *Τυπώνεται ['Ford', 'Volvo', 'BMW', 'Honda']*

Παράδειγμα 5-Προσθήκη στοιχείων Πίνακα

Με τη συνάρτηση append() μπορούμε να προσθέσουμε ένα στοιχείο στον πίνακα όπως φαίνεται στο επόμενο παράδειγμα:
cars.*append*("Honda")
    print(x) → *Τυπώνεται ['Ford', 'Volvo', 'BMW', 'Honda']*

1.12 Αναδρομή σε Python

Παράδειγμα 1 – Υπολογισμός του n!

Το παράδειγμα αυτό παρουσιάζει μια αναδρομική συνάρτηση για τον υπολογισμό του n! Υπενθυμίζουμε ότι ο αναδρομικός τύπος υπολογισμού του n! είναι: factorial(n)= factorial(n-1)*n με αρχική συνθήκη 0!=1.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return factorial(n-1)*n
```

Ο υπολογισμός του n! με επαναληπτική συνάρτηση γίνεται ως εξής:

```
def iterative_factorial(n):
    result = 1
    for i in range(2, n+1):
        result*= i
    return result
```

Παράδειγμα 2 – Υπολογισμός του n-οστού όρου της ακολουθίας Fibonacci

Το παράδειγμα αυτό παρουσιάζει μια αναδρομική συνάρτηση για τον υπολογισμό του n-οστού όρου της ακολουθίας Fibonacci. Υπενθυμίζουμε ότι ο αναδρομικός τύπος υπολογισμού του n-οστού όρου της ακολουθίας Fibonacci είναι fib(n)=fib(n-1)+fib(n-2) με αρχικές συνθήκες fib(0)=0 και fib(1)=1

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

Ο υπολογισμός του του n-οστού όρου της ακολουθίας Fibonacci με επαναληπτική συνάρτηση γίνεται ως εξής:

```
def fib(n):
    a, b = 0, 1
    for i in range(n):
```

```
        a, b = b, a + b
    return a
```

Παράδειγμα 3 – Υπολογισμός αναδρομικού τύπου

Υπολογισμός του αναδρομικού τύπου f(n+1) = f(n) + 3 με αρχική συνθήκη f(1) = 3,

```
def mult3(n):
    if n == 1:
        return 3
    else:
        return mult3(n-1) + 3

    for i in range(1,10):
        print(mult3(i))
```

1.13 Κλάσεις/Αντικείμενα (Classes/Objects)

Παράδειγμα 1-Δημιουργία Κλάσης

Για να δημιουργήσουμε μια νέα κλάση χρησιμοποιούμε την εντολή class και ορίζουμε κάποιες ιδιότητες σε αυτή την κλάση. Στη συνέχεια δηλώνουμε αντικείμενα στην κλάση αυτή. Στο επόμενο παράδειγμα ορίζουμε μια νέα κλάση με όνομα MyClass με την ιδιότητα (μέλος) x και μετά ορίζουμε το αντικείμενο p1 στην κλάση αυτή και τυπώνουμε την τιμή της ιδιότητας x.

class MyClass:
  x = 5 →*το x είναι μέλος της κλάσης MyClass και αρχικοποιείται με την τιμή 5. Τα μέλη μιας κλάσης στην Python είναι εξορισμού public*

p1 = MyClass()→*δημιουργείται το αντικείμενο p1 καλώντας το δημιουργό της κλάσης MyClass*
print(p1.x) →*Τυπώνεται 5*

Παράδειγμα 2-Δημιουργός Κλάσης

Όλες οι κλάσεις έχουν ένα δημιουργό. Ο δημιουργός κάθε κλάσης στην Python είναι η εξορισμού συνάρτηση __init__(). Ο δημιουργός καλείται αυτόματα κάθε φορά που δηλώνουμε ένα αντικείμενο στην κλάση και αρχικοποιεί τα μέλη του. Στο επόμενο παράδειγμα ορίζουμε μια κλάση με όνομα Person και καλούμε τη συνάρτηση __init__() (δηλ. το δημιουργό) για να δώσει αρχικές τιμές στις ιδιότητες-μέλη name και age

class Person:
  def __init__(*self*, name, age):
    self.name = name →*στο μέλος name που προσδιορίζεται από το self καταχωρείται το όρισμα name*
    self.age = age→*στο μέλος age που προσδιορίζεται από το self καταχωρείται το όρισμα age*

p1 = Person("John", 36) →*ορίζουμε το αντικείμενο p1 και καλείται ο δημιουργός της κλάσης Person για να αρχικοποιήσει τα μέλη του p1. Επειδή γράψαμε δημιουργό στην κλάση όταν τον καλούμε πρέπει να του δίνουμε 2 ορίσματα. Αν δεν γράφαμε δημιουργό τότε θα υπήρχε ένας προεπιλεγμένος κενός δημιουργός χωρίς ορίσματα και θα τον καλούσαμε γράφοντας Person()*

print(p1.name)→*τυπώνεται η τιμή John*
print(p1.age) → *τυπώνεται η τιμή 36*

Παράδειγμα 3-Μέθοδοι Κλάσης

Τα αντικείμενα μπορεί να περιέχουν μεθόδους. Οι μέθοδοι των αντικειμένων είναι συναρτήσεις που ανήκουν στα αντικείμενα. Στο επόμενο παράδειγμα ορίζουμε μια μέθοδο στην κλάση Person που τυπώνει ένα μήνυμα. Το αντικείμενο p1 που διαθέτει αυτή τη μέθοδο τυπώνει το μήνυμα.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def myfunc(self):→Η myfunc είναι μέθοδος της κλάσης Person. Ο λόγος που βάζουμε όρισμα τη λέξη self (this) είναι για να προσπελάσουμε μέσα σε αυτή το μέλος name
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc() →τυπώνεται Hello my name is John
```

Παράδειγμα 4-Η παράμετρος self

Η παράμετρος self είναι μια αναφορά στο τρέχον αντικείμενο της κλάσης και χρησιμοποιείται για την προσπέλαση μεταβλητών που ανήκουν στην κλάση. Η παράμετρος self μπορεί να έχει οποιοδήποτε όνομα, αλλά πρέπει να είναι πάντα η 1η παράμετρος οποιασδήποτε συνάρτησης στην κλάση.

Στο επόμενο παράδειγμα που είναι ίδιο με το προηγούμενο χρησιμοποιούμε τις λέξεις mysillyobject και abc εναλλακτικά αντί για τη λέξη self για να δείξουμε στην πράξη ότι η παράμετρος self μπορεί να έχει οποιοδήποτε όνομα.

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc() → τυπώνεται Hello my name is John
```

Παράδειγμα 5-Τροποποίηση Ιδιοτήτων-Μελών

Μπορούμε να τροποποιήσουμε τις ιδιότητες των αντικειμένων μιας κλάσης. Στο επόμενο παράδειγμα τροποποιούμε την ηλικία του στιγμιοτύπου p1 σε 40.

```
class Person:
    def__init__(self, name, age):
        self.name = name
        self.age = age

    def myfunc(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.age = 40
print(p1.age) →τυπώνεται 40
```

1.14 Κληρονομικότητα στην Python

Η Κληρονομικότητα μας επιτρέπει να ορίζουμε μια κλάση που κληρονομεί όλες τις μεθόδους και τις ιδιότητες της υπερκλάσης της.

- Πατρική Κλάση (Parent class) είναι η κλάση από την οποία γίνεται η κληρονομικότητα. Ονομάζεται και κλάση Βάσης
- Κλάση Παιδί (Child class) είναι η κλάση η οποία κληρονομείται από άλλη κλάση (από μια υπερκλάση). Λέγεται και Παραγόμενη κλάση.

Δημιουργία Αρχικής και Παραγόμενης Κλάσης

Ορίζουμε μια κλάση με όνομα Person με τις ιδιότητες firstname και lastname και τη μέθοδο printname

```
class Person:
    def __init__(self, fname, lname):
        self.firstname = fname
        self.lastname = lname

    def printname(self):
        print(self.firstname, self.lastname)
```

x = Person("John", "Doe") →*ορίζουμε το αντικείμενο x καλούμε τη μέθοδο printname() γιαυτό*
x.printname()

Η Κλάση Student είναι παραγόμενη της κλάσης Person και κληρονομεί τα μέλη και τις μεθόδους της κλάσης Person:

class Student(Person): →*Στην παρένθεση γράφεται η υπερκλάση Person την οποία κληρονομεί η παραγόμενη κλάση Student*
    pass →*Η λέξη pass δηλώνει ότι δεν θέλουμε να προσθέσουμε καμία άλλη ιδιότητα ή καμία άλλη μέθοδο στην κλάση αυτή. Άρα η κλάση Student έχει τις ίδιες ιδιότητες και τις ίδιες μεθόδους με την κλάση Parent*

x = Student("Mike", "Olsen") →*Δηλώνουμε ένα αντικείμενο στην κλάση Student και καλούμε τη μέθοδο printname() γιαυτό*
x.printname() →*Τυπώνεται το μήνυμα Mike Olsen*

Η κλάση Child κληρονομεί τις ιδιότητες και τις μεθόδους από την κλάση Parent. Στο επόμενο παράδειγμα προσθέτουμε τη συνάρτηση __init__() στην κλάση Child αντί

για το keyword pass και το μέλος graduationyear. Η συνάρτηση __init__() (δημιουργός) καλείται αυτόματα κάθε φορά που δημιουργούμε ένα αντικείμενο.

class Person:
  def __init__(*self*, fname, lname): →*δημιουργός της κλάσης Person*
    self.firstname = fname
    self.lastname = lname

  def printname(*self*): →*Μέθοδος printname της κλάσης Person. Η μέθοδος αυτή κληρονομείται στην κλάση Student*
    print(*self*.firstname, *self*.lastname)

class Student(Person):
  def __init__(*self*, fname, lname): →*δημιουργός της κλάσης Student*
    Person.__init__(*self*, fname, lname) →*ο δημιουργός της κλάσης Student καλεί το δημιουργό της κλάσης Person*
    self.graduationyear = 2019 →*Το μέλος graduationyear της κλάσης Student αρχικοποιείται με την τιμή 2019*
x = Student("Mike", "Olsen") →*δηλώνεται το αντικείμενο x τύπου Student και καλείται ο δημιουργός της κλάσης Student*
print(x.graduationyear) →*Τυπώνεται η τιμή 2019*

Προσθήκη Μεθόδων στην Παραγόμενη Κλάση

class Person:
  def __init__(*self*, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(*self*):
    print(*self*.firstname, *self*.lastname)

class Student(Person):
  def __init__(*self*, fname, lname, year):
    Person.*__init__*(*self*, fname, lname)
    self.graduationyear = year

  def welcome(*self*): →*Η μέθοδος welcome προστίθεται στην παραγόμενη κλάση Student*
    print("Welcome", *self*.firstname, *self*.lastname, " to the class of ", *self*.graduationyear)

x = Student("Mike", "Olsen", 2019)
x.welcome() →*Τυπώνεται το μήνυμα Welcome Mike Olsen to the class of 2019*

Β Παράδειγμα Κληρονομικότητας

Στο επόμενο παράδειγμα ορίζουμε την υπερκλάση Person. Διαθέτει ένα δημιουργό που αρχικοποιεί τα μέλη firstname και lastname. Επίσης διαθέτει και τη μέθοδο Name που επιστρέφει τα μέλη firstname και lastname. Στη συνέχεια ορίσουμε την παραγόμενη κλάση Employee που κληρονομεί την κλάση Person. Διαθέτει ένα δημιουργό που καλεί πρώτα το δημιουργό της υπερκλάσης Person και μετά αρχικοποιεί το μέλος staffnumber. Επίσης η κλάση Employee διαθέτει και τη μέθοδο GetEmployee που επιστρέφει το αποτέλεσμα της μεθόδου Name() και σε αυτό συνενώνει και το μέλος staffnumber.

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        Person.__init__(self, first, last)
        self.staffnumber = staffnum



    def GetEmployee(self):
        return self.Name() + ", " +  self.staffnumber

x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")

print(x.Name()) →Τυπώνεται το μήνυμα Marge Simpson
print(y.GetEmployee()) → Τυπώνεται το μήνυμα Homer Simpson 1007
```

Παρατήρηση 1

Η μέθοδος __init__ της (παραγόμενης) κλάσης Employee καλεί τη μέθοδο __init__ της (υπερ) κλάσης Person. Θα μπορούσαμε εναλλακτικά να γράψουμε super().__init__(first, last) για να την καλέσουμε δηλαδή:

```
def __init__(self, first, last, staffnum):
```

```
    super().__init__(first, last)
    self.staffnumber = staffnum
```

Παρατήρηση 2

Αντί να χρησιμοποιήσουμε τις μεθόδους Name" and "GetEmployee" του προηγούμενου παραδείγματος θα ήταν προτιμότερο να συμπεριλάβουμε αυτή τη λειτουργικότητα στη μέθοδο __str__ όπως δείχνουμε στη συνέχεια:

```
class Person:
    def __init__(self, first, last):
        self.firstname = first
        self.lastname = last

    def __str__(self):
        return self.firstname + " " + self.lastname
    def Name(self):
        return self.firstname + " " + self.lastname

class Employee(Person):
    def __init__(self, first, last, staffnum):
        super().__init__(first, last)
        self.staffnumber = staffnum
x = Person("Marge", "Simpson")
y = Employee("Homer", "Simpson", "1007")
print(x)→τυπώνεται το μήνυμα Mare Simpson
print(y) → τυπώνεται το μήνυμα Homer Simpson
```

2 Παραλληλισμός με Νήματα

Το multithreading είναι μια μέθοδος προγραμματισμού που χρησιμοποιεί νήματα. Ένα νήμα εκτέλεσης είναι η μικρότερη ακολουθία προγραμματισμένων εντολών που μπορεί να διαχειρισθεί ανεξάρτητα, από το λειτουργικό σύστημα. Ένα νήμα είναι μια ελαφριά διεργασία. Η υλοποίηση των νημάτων και των διεργασιών διαφέρει από το ένα λειτουργικό σύστημα στο άλλο. Στις περισσότερες όμως περιπτώσεις ένα νήμα εμπεριέχεται σε μια διεργασία. Μπορούν να υπάρχουν πολλαπλά νήματα μέσα στην ίδια διεργασία τα οποία μπορούν να μοιράζονται πόρους από το σύστημα, όπως μνήμη. Διαφορετικές διεργασίες δεν μπορούν να μοιράζονται τους ίδιους πόρους. Συγκεκριμένα, τα νήματα μιας διεργασίας περιέχουν τις εντολές προς την εκτελούμενη διεργασία (δηλαδή τον κώδικα της) και το εννοιολογικό της πλαίσιο (οι τιμές των μεταβλητών της σε οποιαδήποτε χρονική στιγμή. Σε ένα απλό επεξεργαστή, η πολυνημάτωση (multithreading) πραγματοποιείται με τη μέθοδο της πολυπλεξίας με διαίρεση χρόνου (όπως στην πολυεπεξεργασία): ο επεξεργαστής μεταπηδάει μεταξύ των διάφορων νημάτων. Αυτή η εναλλαγή μεταξύ των διεργασιών ονομάζεται context switch, και συμβαίνει σε πολύ τακτά χρονικά διαστήματα, τέτοια έτσι ώστε ο χρήστης έχει την εντύπωση ότι τα νήματα εκτελούνται την ίδια στιγμή. Μόνο σε έναν επεξεργαστή με πολλούς επεξεργαστικούς πυρήνες, τα νήματα εκτελούνται πραγματικά ταυτόχρονα και κάθε πυρήνας εκτελεί ένα συγκεκριμένο νήμα ή εργασία.

Υπάρχουν όμως και προφανή προβλήματα που μπορεί κανείς να φανταστεί με το multithreading. Για παράδειγμα πολλά νήματα που προσπαθούν να αποκτήσουν πρόσβαση στο ίδιο τμήμα δεδομένων μπορεί να οδηγήσουν σε προβλήματα όπως η ασυνεπής παραγωγή δεδομένων ή η φθορά της εξόδου (όπως το HWeolrldo στη θέση του Hello World στην κονσόλα μας). Τέτοια προβλήματα μπορεί να προκύψουν όταν δεν λέμε στον υπολογιστή πώς να κάνουμε νήματα με οργανωμένο τρόπο.

Αλλά πώς μπορούμε να «πούμε» στον υπολογιστή να διατηρήσει συγχρόνως τα νήματα του προγράμματος μας; Το κάνουμε χρησιμοποιώντας πρότυπα συγχρονισμού. Αυτοί είναι απλοί μηχανισμοί λογισμικού για να διασφαλίσουν ότι τα νήματά λειτουργούν αρμονικά μεταξύ τους

Σε αυτό το κεφάλαιο παρουσιάζει μερικά από τα πιο δημοφιλή πρότυπα συγχρονισμού στο Python, που ορίζονται στο module threading.py. Οι περισσότερες από τις μεθόδους αποκλεισμού (δηλαδή, οι μέθοδοι που εμποδίζουν την εκτέλεση ενός συγκεκριμένου νήματος έως ότου ικανοποιηθεί κάποια συνθήκη) παρέχουν την προαιρετική λειτουργικότητα του χρονικού ορίου, αλλά δεν το έχουμε συμπεριλάβει εδώ για λόγους απλότητας

Θα ορίσουμε ένα νήμα μεταβιβάζοντάς τον έναν αριθμό, ο οποίος αντιπροσωπεύει τον αριθμό του και ακολουθώντας τα επόμενα βήματα

1. Εισάγουμε αρχικά το module threading με την εντολή import threading

2. Στο κύριο πρόγραμμα, δημιουργείται ένα αντικείμενο νήματος και συσχετίζεται με μια συνάρτηση στόχου (target function) που πρέπει να εκτελεστεί από το νήμα

3. Στη συνέχεια, μεταβιβάζεται ένα όρισμα τη συνάρτηση αυτή οποίο θα συμπεριληφθεί στο μήνυμα εξόδου με την εντολή t = threading.Thread(target=function , args=(i,))

3. Το νήμα δεν αρχίζει να εκτελείται μέχρις ότου κληθεί η μέθοδοςs start έως και αντίστοιχα η συνάρτηση join που συνενώνει όλα τα νήματα αναγκάζει το καλόν νήμα να περιμένει και την ολοκλήρωση των υπολοίπων

Η δημιουργία νημάτων επεξηγείται στο ακόλουθο πρόγραμμα:

```
import threading
def my_func(thread_number):
    return print('my_func called by thread N°\
        {}'.format(thread_number))
def main():
    threads = []
    for i in range(10):
        t = threading.Thread(target=my_func, args=(i,))
        threads.append(t)
        t.start()
        t.join()
if __name__ == "__main__":
    main()
```

Επεξηγήσεις Κώδικα

import threading *#Εισαγωγή του module threading που περιλαμβάνει συναρτήσεις διαχείρισης νημάτων*

def my_func(thread_number): *#Ορισμός συνάρτησης που πρέπει να εκτελέσουν τα νήματα*

return print('my_func called by thread N°\
{}'.***format***(thread_number))*#Κάθε νήμα τυπώνει τον αριθμό του λαμβάνει ως όρισμα*

def main(): *#ορισμός κύριου προγράμματος (συνάρτηση main)*

threads = [] *#δήλωση πίνακα νημάτων*

for i ***in range***(10):

t = threading.***Thread***(target=my_func, ***args***=(i,)) *#δημιουργούνται 10 νήματα και στο καθένα από αυτά ανατίθεται η εκτέλεση της συνάρτησης my_func με όρισμα ένα ακέραιο από 0 μέχρι 9 που αντιπροσωπεύει τον αριθμό του νήματος*

threads.***append***(t) *#όλα τα νήματα προστίθενται στον πίνακα νημάτων*

t.***start***() *#κάθε νήμα αρχίζει να εκτελείται κα καλεί τη συνάρτηση my_func*

t.***join***() *#κάθε νήμα που ολοκληρώνει τη my_func συνενώνεται με τα υπόλοιπα νήματα στο τέλος της παράλληλης περιοχής*

if __name__ == "__main__": *#με την εντολή αυτή εκτελείται αυτόματα η συνάρτηση main()*

main()

Αποτελέσματα Εκτέλεσης

```
my_func called by thread N°    0
my_func called by thread N°    1
my_func called by thread N°    2
my_func called by thread N°    3
my_func called by thread N°    4
my_func called by thread N°    5
my_func called by thread N°    6
my_func called by thread N°    7
my_func called by thread N°    8
my_func called by thread N°    9
```

2.1 Κλειδώματα (Locks)

Τα κλειδώματα είναι ίσως οι απλούστεροι μηχανισμοί συγχρονισμού στην Python. Το Lock έχει μόνο δύο καταστάσεις: κλειδωμένο (lock) και ξεκλείδωτο (unlock) Δημιουργείται σε κατάσταση ξεκλειδώματος και έχει δύο κύριες μεθόδους acquire() και release(). Η μέθοδος acquire () κλειδώνει το Lock και αποκλείει την εκτέλεση του νήματος έως ότου η μέθοδος release() σε κάποια άλλη ρουτίνα να την ξεκλειδώσει. Στη συνέχεια κλειδώνει ξανά το Lock και επιστρέφει True. Η μέθοδος release () μπορεί να καλείται μόνο σε κατάσταση κλειδώματος, θέτει την κατάσταση σε unlock και τερματίζει αμέσως. Εάν καλείται η release() σε κατάσταση ξεκλειδώματος, εμφανίζεται ένα RunTimeError.

Ακολουθεί ο κώδικας που χρησιμοποιεί ένα Lock για την προσπέλαση μιας διαμοιραζόμενης/Κοινόχρηστης (shared) μεταβλητής

```
#lock_tut.py

from threading import Lock, Thread
lock = Lock()
g = 0

def add_one():

   global g
   lock.acquire()
   g += 1
   lock.release()

def add_two():
   global g
   lock.acquire()
   g += 2
   lock.release()

threads = []
for func in [add_one, add_two]:
   threads.append(Thread(target=func))
   threads[-1].start()

for thread in threads:

   thread.join()

print(g)
```

Επεξηγήσεις Κώδικα

from threading import Lock, Thread *#Εισάγουμε τα modules Lock και Thread*

lock = ***Lock***()

g = 0

def add_one():

  global g

  lock.***acquire***() *#κλείδωμα του lock*

  g += 1

  lock.***release***() *#ξεκλείδωμα του lock*

def add_two():

  global g

  lock.***acquire***() *#κλείδωμα του lock*

  g += 2

  lock.***release***() *#ξεκλείδωμα του lock*

threads = []

for func ***in*** [add_one, add_two]:

threads.***append***(Thread(target=func)) *#δημιουργούνται 2 νήματα, προστίθενται στον πίνακα threads και σε κάθε νήμα ανατίθεται η εκτέλεση της συνάρτησης func*

threads[-1].***start***() *#αρχίζει η εκτέλεση κάθε νήματος*

for thread ***in*** threads:

thread.***join***()

print(g)

Η έξοδος του προγράμματος δίνει 3 και έτσι σιγουρευόμαστε ότι οι δύο συναρτήσεις δεν αλλάζουν ταυτόχρονα την τιμή της καθολικής μεταβλητής g αν και τρέχουν σε δύο διαφορετικά νήματα. Έτσι, το Locks μπορεί να χρησιμοποιηθεί για την αποφυγή ασυνεπούς εξόδου επιτρέποντας μόνο σε ένα νήμα να τροποποιεί δεδομένα κάθε φορά.

2.2 RLocks

Το τυπικό Lock δεν γνωρίζει ποιο νήμα κρατάει το κλείδωμα. Εάν διατηρηθεί το κλείδωμα, οποιοδήποτε νήμα που προσπαθεί να το αποκτήσει θα μπλοκάρει, ακόμη και αν το ίδιο νήμα κρατά ήδη το κλείδωμα. Σε τέτοιες περιπτώσεις, χρησιμοποιείται RLock (re-entrant lock - κλείδωμα επανα-εισόδου). Επεκτείνουμε τον προηγούμενο κώδικα προσθέτοντας δηλώσεις εξόδου για να δείξουμε τον τρόπο με τον οποίο το RLock μπορεί να αποτρέψει τον ανεπιθύμητο αποκλεισμό.

```
import threading

num = 0
lock = Threading.Lock()

lock.acquire()
num += 1
lock.acquire() # This will block.
num += 2
lock.release()


# With RLock, that problem doesn't happen.
lock = Threading.RLock()

lock.acquire()
num += 3
lock.acquire() # This won't block.
num += 4
lock.release()
lock.release() # You need to call release once for each call to acquire.
```

<u>Επεξηγήσεις Κώδικα</u>

import threading

```
num = 0
lock = Threading.Lock() #εκτελούμε τη συνάρτηση Lock()

lock.acquire() () #κλείδωμα του lock
num += 1
lock.acquire() #Αυτή η εκτέλεση της acquire μπλοκάρει το νήμα
num += 2
lock.release() #ξεκλείδωμα του lock

#Με τη χρήση RLock το πρόβλημα αυτό δεν εμφανίζεται
lock = Threading.RLock()

lock.acquire()#κλείδωμα του lock
num += 3
lock.acquire() #Το νήμα εδώ δεν μπλοκάρει
num += 4
lock.release()
lock.release() # Πρέπει για κάθε κλήση της acquire να υπάρχει αντίστοιχη κλήση της release
```

Μία καλή περίπτωση χρήσης για RLocks είναι η αναδρομή (recursion), όταν μια γονική κλήση μιας συνάρτησης θα απέκλειε σε διαφορετική περίπτωση την εμφωλευμένη (nested) κλήση της. Έτσι, η κύρια χρήση του RLocks είναι η εμφωλευμένη πρόσβαση σε κοινόχρηστους πόρους.

2.3 Σημαφόροι (Semaphores)

Ένα βασικό αντικείμενο που αφορά νήματα είναι ο σημαφόρος. Ένας σημαφόρος είναι ένας μετρητής με μερικές χαρακτηριστικές ιδιότητες. Η πρώτη ιδιότητα είναι ότι ο μετρητής αυτός είναι ατομικός (atomic). Αυτό σημαίνει ότι το λειτουργικό σύστημα εγγυημένα δεν θα κάνει swap out ένα νήμα στη μέση της αύξησης ή της μείωσης ενός μετρητή. Ο εσωτερικός μετρητής αυξάνεται με την κλήση της συνάρτησης release() και μειώνεται με την κλήση της συνάρτησης acquire().

Μια δεύτερη ιδιότητα είναι ότι ένα νήμα καλεί την acquire() ενώ ο μετρητής έχει την τιμή μηδέν, τότε το νήμα θα μπλοκάρει μέχρις ότου ένα άλλο νήμα καλέσει τη συνάρτηση release() και αυξήσει την τιμή του μετρητή στο ένα.

Μια τιμή λάθους θα εμφανιστεί μόνο αν η οι κλήσεις της συνάρτηση release() προσπαθήσουν να αυξήσουν το μετρητή πάνω από το μέγιστη τιμή του (που είναι ο αριθμός νημάτων που μπορούν να εκτελέσουν την συνάρτηση acquire() στο σημαφόρο πριν εμφανιστεί το block.

Οι σημαφόροι χρησιμοποιούνται συχνά για την προστασία ενός πόρου που έχει περιορισμένη χωρητικότητα. Ένα παράδειγμα θα ήταν αν έχουμε μια ομάδα συνδέσεων και θέλετε να περιορίσετε το μέγεθος αυτής της ομάδας σε έναν συγκεκριμένο αριθμό.

Ο κώδικας που ακολουθεί δείχνει τη χρήση των σηματοφόρων σε ένα απλό πρόβλημα παραγωγού-καταναλωτή.

```
import random, time
from threading import BoundedSemaphore, Thread

max_items = 5

container = BoundedSemaphore(max_items)

def producer(nloops):
    for i in range(nloops):
        time.sleep(random.randrange(2, 5))
        print(time.ctime(), end=": ")
        try:
            container.release()
            print("Produced an item.")
        except ValueError:
            print("Full, skipping.")

def consumer(nloops):
    for i in range(nloops):
        time.sleep(random.randrange(2, 5))
        print(time.ctime(), end=": ")

        if container.acquire(False):
            print("Consumed an item.")
        else:
            print("Empty, skipping.")

threads = []
nloops = random.randrange(3, 6)
print("Starting with %s items." % max_items)
threads.append(Thread(target=producer, args=(nloops,)))
threads.append(Thread(target=consumer, args=(random.randrange(nloops, nloops+max

for thread in threads:
    thread.start()
for thread in threads:
    thread.join()

print("All done.")
```

<u>Επεξηγήσεις Κώδικα</u>

import random, time

from threading import BoundedSemaphore, Thread

max_items = 5

container = ***BoundedSemaphore***(max_items) *#δημιουργούμε ένα buffer δηλ. μια μνήμη 5 θέσεων. Αν ο παραγωγός επιχειρήσει μια κλήση release() πέραν του μεγέθους του buffer δηλ. αν επιχειρήσει να βάλει περισσότερα στοιχεία από το μέγεθος του buffer θα προκληθεί λάθος*

def producer(nloops): *#ορισμός συνάρτησης παραγωγού*

for i ***in range***(nloops):

time.sleep(***random.randrange***(2, 5)) *#κάθε νήμα γίνεται sleep για ένα τυχαίο χρόνο από 2 έως 5 msec*

print(***time.ctime***(), ***end***=": ") *#τυπώνεται η τρέχουσα χρονική στιγμή*

```
        try: #ο κώδικας στο try μπορεί να προκαλέσει error
            container.release()#το νήμα ξεκλειδώνει το κλειδί και τοποθετεί
        (βάζει) στοιχείο στο buffer
            print("Produced an item.")
        except ValueError: #αν το container.release() προκαλέσει σφάλμα κατά
    την εκτέλεση του δηλ. αν ο buffer είναι γεμάτος και δεν χωρά άλλο στοιχείο
            print("Full, skipping.") #τυπώνεται μήνυμα

def consumer(nloops):
        for i in range(nloops):
            time.sleep(random.randrange(2, 5))
            print(time.ctime(), end=": ")

            if container.acquire(False):#το νήμα κλειδώνει το κλειδί και
        αφαιρεί (καταναλώνει) στοιχείο από το buffer
                print("Consumed an item.")
            else:
                print("Empty, skipping.")

threads = []
nloops = random.randrange(3, 6)
print("Starting with %s items." % max_items)
threads.append(Thread(target=producer, args=(nloops,)))
threads.append(Thread(target=consumer, args=(random.randrange(nloops,
nloops+max_items+2),)))

for thread in threads:  # Εδώ αρχίζουν όλα τα νήματα. σε κάθε επανάληψη στη
μεταβλητή thread αναθέτει ένα νήμα
        thread.start() //με τη συνάρτηση start() #εκτελείται το κάθε νήμα

for thread in threads:  # Με το for αυτό το πρόγραμμα περιμένει όλα τα νήματα
να τελειώσουν
    thread.join()

print("All done.")
```

Αποτελέσματα Εκτέλεσης

Starting with 5 items.
Fri Apr 24 20:53:17 2020Fri Apr 24 20:53:17 2020: : Full, skipping. Consumed an item
Fri Apr 24 20:53:19 2020: Produced an item.
Fri Apr 24 20:53:20 2020: Consumed an item.

Fri Apr 24 20:53:21 2020: Produced an item.
Fri Apr 24 20:53:22 2020: Consumed an item.
Fri Apr 24 20:53:25 2020: Consumed an item.
Fri Apr 24 20:53:29 2020: Consumed an item.
All done.

Το threading module διαθέτει την κλάση Semaphore. Η κλάση αυτή διαθέτει ένα μετρητή που μας επιτρέπει να καλέσουμε την συνάρτηση release() οσοδήποτε αριθμό φορών για αύξηση. Για να αποφευχθούν προγραμματιστικά λάθη είναι μια λογική κίνηση είναι να χρησιμοποιήσουμε τον τύπο BoundedSemaphore που προκαλεί error αν μια κλήση release() επιχειρήσει να αυξήσει το μετρητή πέραν της μέγιστης τιμής του.

Οι σηματοφόροι χρησιμοποιούνται για να περιορίσουν την πρόσβαση σε πόρους όπως π.χ. μόνο 10 clients ανταγωνίζονται π.χ. για ένα server.

2.4 Συμβάντα (Events)

Η αρχή συγχρονισμού Event λειτουργεί ως ένας απλός communicator μεταξύ νημάτων. Τα events βασίζονται σε ένα εσωτερικό flag για το οποίο τα νήματα μπορούν να εκτελέσουν τις συναρτήσεις set() και clear(). Τα άλλα νήματα μπορούν να εκτελέσουν τη συνάρτηση wait() και να περιμένουν το εσωτερικό flag να "τεθεί" δηλ. να εκτελέσει τη συνάρτηση set(). Η συνάρτηση wait() μπλοκάρει μέχρι το flag να γίνει true. Το επόμενο τμήμα κώδικα παρουσιάζει τη χρήση των Events:

```
import random, time
from threading import Event, Thread

event = Event()

def waiter(event, nloops):
    for i in range(nloops):
        print("%s. Waiting for the flag to be set." % (i+1))
        event.wait()
        print("Wait complete at:", time.ctime())
        event.clear()
        print()

def setter(event, nloops):
    for i in range(nloops):
        time.sleep(random.randrange(2, 5))
        event.set()

threads = []
nloops = random.randrange(3, 6)

threads.append(Thread(target=waiter, args=(event, nloops)))
threads[-1].start()
threads.append(Thread(target=setter, args=(event, nloops)))
threads[-1].start()

for thread in threads:
    thread.join()

print("All done.")
```

<u>Επεξηγήσεις Κώδικα</u>

import random, time

from threading import Event, Thread

event = ***Event***()

def waiter(event, nloops):

    for i ***in range***(nloops):

        print("%s. Waiting for the flag to be set." % (i+1))

        event.***wait***() *#Μπλοκάρει μέχρι το flag να γίνει true*

```
        print("Wait complete at:", time.ctime())
        event.clear() #Κάνει reset to flag δηλ. το κάνει false
        print()

def setter(event, nloops):
    for i in range(nloops):
        time.sleep(random.randrange(2, 5)) #Κάνει Sleep για ένα τυχαίο χρονικό
    διάστημα
        event.set() #κάνει το flag true

threads = []
nloops = random.randrange(3, 6)

threads.append(Thread(target=waiter, args=(event, nloops))) #δημιουργούμε ένα νήμα. Του αναθέτουμε την εκτέλεση της συνάρτησης waiter. Η συνάρτηση έχει 2 ορίσματα. Το event και το nloops και το νήμα προστίθεται στον πίνακα threads
threads[-1].start() #το νήμα εκτελείται

threads.append(Thread(target=setter, args=(event, nloops)))
threads[-1].start()

for thread in threads:
    thread.join()

print("All done.")
```

Αποτελέσματα Εκτέλεσης

1. Waiting for the flag to be set.
Wait complete at: Sat Apr 25 12:40:36 2020

2. Waiting for the flag to be set.
Wait complete at: Sat Apr 25 12:40:39 2020

3. Waiting for the flag to be set.
Wait complete at: Sat Apr 25 12:40:42 2020

4. Waiting for the flag to be set.
Wait complete at: Sat Apr 25 12:40:44 2020

2.5 Συνθήκες (Conditions)

Ένα αντικείμενο συνθήκης (Condition object) είναι απλά μια πιο εξελιγμένη έκδοση του αντικειμένου Event. Λειτουργεί επίσης ως communicator μεταξύ των νημάτων και μπορεί να καλέσει τη συνάρτηση notify() ειδοποιώντας τα άλλα νήματα ότι έγινε μια αλλαγή στην κατάσταση του προγράμματος. Για παράδειγμα μπορεί να χρησιμοποιηθεί για να σηματοδοτήσει τη διαθεσιμότητα ενός πόρου. Τα άλλα νήματα αποκτούν με την εκτέλεση της συνάρτησης acquire() τη συνθήκη (condition) και την κλειδώνουν χωρίς να περιμένουν (με την εκτέλεση της wait()) τη συνθήκη να ικανοποιηθεί. Επίσης ένα νήμα πρέπει να καλέσει τη συνάρτηση release() για να αποδεσμεύσει μια συνθήκη όταν έχει ολοκληρώσει τις απαιτούμενες ενέργειες έτσι ώστε τα άλλα νήματα να αποκτήσουν με τη σειρά τους το condition για τους δικούς τους λόγους. Ο επόμενος κώδικας δείχνει την υλοποίηση του προβλήματος παραγωγού καταναλωτή με χρήση του Object Condition

```
import random, time
from threading import Condition, Thread

condition = Condition()

box = []

def producer(box, nitems):
    for i in range(nitems):
        time.sleep(random.randrange(2, 5))
        condition.acquire()
        num = random.randint(1, 10)
        box.append(num)
        condition.notify()
        print("Produced:", num)
        condition.release()

def consumer(box, nitems):
    for i in range(nitems):
        condition.acquire()
        condition.wait()  .
        print("%s: Acquired: %s" % (time.ctime(), box.pop()))
        condition.release()

threads = []

nloops = random.randrange(3, 6)
for func in [producer, consumer]:
    threads.append(Thread(target=func, args=(box, nloops)))
    threads[-1].start()  # Starts the thread.

for thread in threads:
    thread.join()

print("All done.")
```

Επεξηγήσεις Κώδικα

```
import random, time
from threading import Condition, Thread

condition =Condition()

box = []

def producer(box, nitems):
	for i in range(nitems):
		time.sleep(random.randrange(2, 5))  #Γίνεται sleep για ένα τυχαίο διάστημα
		condition.acquire() #εκτελώντας τη συνάρτηση acquire() ανεβάζει το κλειδί ώστε
	να αποκτήσει πρόσβαση στο box δηλ στην κοινή μνήμη
		num =random.randint(1, 10) #το στοιχείο που θα μπει στο box
		box.append(num)  # ένα στοιχείο μπαίνει στο buffer για κατανάλωση
		condition.notify()  #ενημερώνει τον consumer ότι υπάρχει τέτοιο στοιχείο για
	κατανάλωση
		print("Produced:", num)
		condition.release()

def consumer(box, nitems):
	for i in range(nitems):
		condition.acquire()#εκτελώντας τη συνάρτηση acquire() ανεβάζει το κλειδί ώστε
	να αποκτήσει πρόσβαση στο box δηλ στην κοινή μνήμη
		condition.wait() #O consumer μπλοκάρει μέχρι να υπάρχει στοιχείο προς κατανάλωση
		print("%s: Acquired: %s" % (time.ctime(), box.pop()))
		condition.release()

threads = []

nloops = random.randrange(3, 6)
for func in [producer, consumer]:
	threads.append(Thread(target=func, args=(box, nloops)))
	threads[-1].start() # το νήμα αρχίζει

for thread in threads:
	thread.join()

print("All done.")
```

Αποτελέσματα Εκτέλεσης

Produced: 2
Sat Apr 25 14:11:56 2020: Acquired: 2
Produced: 7
Sat Apr 25 14:11:59 2020: Acquired: 7
Produced: 8
Sat Apr 25 14:12:02 2020: Acquired: 8
All done.

2.6 Φράγματα (Barriers)

Ένα Φράγμα (barrier) είναι μια βασική δομή που μπορεί να χρησιμοποιηθεί από πολλά νήματα με στόχο ώστε όταν ένα νήμα φτάσει σε ένα συγκεκριμένο τμήμα του κώδικα να περιμένει και τα υπόλοιπα νήματα να φτάσουν σε αυτό. Κάθε νήμα προσπαθεί να περάσει το φράγμα εκτελώντας τη συνάρτηση wait() η οποία το μπλοκάρει μέχρις όρου όλα τα νήματα να φτάσουν στο φράγμα. Όταν συμβεί αυτό όλα τα νήματα συνεχίζουν ταυτόχρονα. Τα φράγματα έχουν πολλές εφαρμογές. Μια χαρακτηριστική εφαρμογή είναι ο συγχρονισμός ενός client και ενός server καθώς ο server πρέπει να περιμένει τον client αφού πρώτα ξεκινήσει ο ίδιος. Το επόμενο τμήμα κώδικα παρουσιάζει τη χρήση των Barriers.

```
from random import randrange
from threading import Barrier, Thread
from time import ctime, sleep

num = 4

b = Barrier(num)
names = ["First", "Second", "Third", "Lst"]

def player():
    name = names.pop()
    sleep(randrange(2, 5))
    print("%s reached the barrier at: %s\n" % (name, ctime()))
    b.wait()

threads = []
print("Race starts now…")

for i in range(num):
    threads.append(Thread(target=player))
    threads[-1].start()

for thread in threads:
    thread.join()
print()
print("Race over!")
```

Επεξηγήσεις Κώδικα

```
from random import randrange
from threading import Barrier, Thread
from time import ctime, sleep

num = 4
#4 νήματα θα φτάσουν στο φράγμα
b =Barrier(num)
names = ["First", "Second", "Third", "Last"]

def player():
	name = names.pop()
	sleep(randrange(2, 5))
	print("%s reached the barrier at: %s" % (name, ctime()))
	b.wait()

threads = []
print("Race starts now…")

for i in range(num):
	threads.append(Thread(target=player))
	threads[-1].start()

for thread in threads:
	thread.join()
print()
print("Race over!")
```

Αποτελέσματα Εκτέλεσης

Race starts now…

Third reached the barrier at: Sat Apr 25 22:16:02 2020

First reached the barrier at: Sat Apr 25 22:16:02 2020

Second reached the barrier at: Sat Apr 25 22:16:03 2020

Lst reached the barrier at: Sat Apr 25 22:16:04 2020

Race over!

2.7 Χρονιστές (Timers)

Τα αντικείμενα Timer χρησιμοποιούνται για τον προγραμματισμό ενεργειών που πρέπει να προγραμματιστούν για εκτέλεση μετά την πάροδο συγκεκριμένου χρόνου. Αυτά τα αντικείμενα προγραμματίζονται να εκτελούνται από ξεχωριστό νήμα που εκτελείται. Η Timer είναι μια υποκλάση της κλάσης Thread που ορίζεται στην Python. Αρχίζει με την εκτέλεση της συνάρτησης start() που αναφέρεται ρητά στον Timer. Η δημιουργία του timer γίνεται με την επόμενη εντολή:

threading.Timer(interval, function, args = None, kwargs = None)

Δημιουργείται ένας timer που θα εκτελέσει τη συνάρτηση function με τα ορίσματα args και kwargs μετά την πάροδο χρονικού διαστήματος ίσο με interval seconds.

Παράδειγμα με Timer

Στο επόμενο παράδειγμα προγραμματίζεται η συνάρτηση myfunc() για να εκτελεστεί 5 sec μετά την κλήση της start()

```
import threading
def myfunc():
        print("Testing Timer\n")

timer = threading.Timer(2.0, myfunc)
timer.start() #αρχίζει το νήμα να εκτελείται μετά την πάροδο 2 sec, δηλ, μετά από 2 sec
θα εκτελέσει τη συνάρτηση myfunc
print("Exit\n")
```

Αποτελέσματα Εκτέλεσης

Exit

Testing Timer

2.8 Χαρακτηριστικά Προβλήματα Συγχρονισμού με Νήματα

2.8.1 Πρόβλημα Παραγωγού-Καταναλωτή (Producer-Consumer) Problem

2.8.1.1 Βοηθητικό Module myThread.py

```
import threading
from time import ctime

class MyThread(threading.Thread):
    def __init__(self, func, args, name=''):
        threading.Thread.__init__(self)
        self.name = name
        self.func = func
        self.args = args

    def getResult(self):
        return self.res

    def run(self):
        print("starting", self.name, 'at:', ctime())
        self.res = self.func(*self.args)
        print(self.name, "finished at:", ctime())
```

2.8.1.2 Κύριο Module prod_cons.py

```
from myThread import MyThread
from queue import Queue
from random import randint
from time import sleep

def writeQ(queue):
    print("producing object for Q...", end=" ")
    queue.put("xxx", 1)
    print("size now:", queue.qsize())

def readQ(queue):
    val = queue.get(1)
    print("consumed object from Q... size now", queue.qsize())

def writer(queue, loops):
    for i in range(loops):
```

```
        writeQ(queue)
    sleep(randint(1, 3))

def reader(queue, loops):
    for i in range(loops):
        readQ(queue)
        sleep(randint(2, 5))

funcs = [writer, reader]
nfuncs = range(len(funcs))

def main():
    nloops = randint(2, 5)
    q = Queue(32)

    threads = []
    for i in nfuncs:
        t = MyThread(funcs[i], (q, nloops), funcs[i].__name__)
        threads.append(t)

    for i in nfuncs:
        threads[i].start()

    for i in nfuncs:
        threads[i].join()

    print("all DONE")

if __name__ == "__main__":
    main()
```

Αποτελέσματα Εκτέλεσης

```
startingstarting  writerreader at: at:  Sat Apr 25 22:38:22 2020Sat Apr 25 22:38:22 2020

producing object for Q... size now: consumed object from Q... size now  10

producing object for Q... size now: 1
consumed object from Q... size now 0
producing object for Q... size now: 1
consumed object from Q... size nowproducing object for Q...  0size now:
 1
```

```
producing object for Q... size now: 2
writer finished at: Sat Apr 25 22:38:30 2020
consumed object from Q... size now 1
consumed object from Q... size now 0
reader finished at: Sat Apr 25 22:38:40 2020
all DONE
```

3 Παραλληλισμός με Διεργασίες

3.1 Ορισμός Διεργασίας

- Διεργασία (Process) είναι ο μηχανισμός εκτέλεσης ενός προγράμματος

- Η Διεργασία είναι μια ενεργή οντότητα (περιέχει ένα σύνολο από συσχετισμένους με αυτή πόρους προκειμένου να εκτελεστεί). Το πρόγραμμα είναι παθητική οντότητα. Σημαντικό: ένα πρόγραμμα από μόνο του δεν αποτελεί διεργασία

- Κάθε διεργασία εκτελεί ένα μοναδικό πρόγραμμα. Το ίδιο πρόγραμμα μπορεί να εκτελείται από πολλές διεργασίες (με διαφορετικές ταχύτητες και ακολουθίες εκτέλεσης των εντολών)

- Η εκτέλεση μιας διεργασίας γίνεται σειριακά. Το λειτουργικό μπορεί όμως να εκτελεί πολλές διεργασίες «παράλληλα» μεταξύ τους.

- Στα παραδοσιακά λειτουργικά συστήματα η «μονάδα χρήσης» της CPU είναι η διεργασία (process).

- Διεργασία είναι μια ανεξάρτητη μονάδα εκτέλεσης που εκτελείται και διαχειρίζεται από το λειτουργικό σύστημα. Κάθε διεργασία είναι ένα εκτελούμενο πρόγραμμα που έχει τα ακόλουθα χαρακτηριστικά:

  - ιδιωτική περιοχή διευθύνσεων
  - δικό της program counter
  - δικό της stack
  - δικούς της registers
  - Πληροφορίες κατάστασης (State information)
  - Ιδεατό χώρο μνήμης (Virtual memory)

- Ο χώρος κάθε διεργασίας είναι προστατευμένος από τις υπόλοιπες διεργασίες του συστήματος.

- Κάθε διεργασία αποτελείται από ένα μοναδικό νήμα ελέγχου (thread of control) μια και έχει ένα program counter

- Η Επικοινωνία μεταξύ διεργασιών γίνεται με σαφώς προκαθορισμένους τρόπους

3.2 Ορισμός Νήματος

- Οι Διεργασίες είναι πολυνηματικές. Κάθε διεργασία διαθέτει τουλάχιστον ένα νήμα.

Τα νήματα (threads) εκτελούν διαφορετικά τμήματα της ίδιας διεργασίας επιταχύνοντας την εκτέλεση της **Πιο τυπικά νήμα είναι ένα μονοπάτι εκτέλεσης εντολών εντός μιας διεργασίας με τα ακόλουθα χαρακτηριστικά:**

- Όλα τα νήματα μοιράζονται τους πόρους της διεργασίας. Πολύ σημαντικός πόρος είναι ο χώρος διευθύνσεων ιδεατής μνήμης δηλ. τα νήματα μοιράζονται την ίδια μνήμη

- Όλα τα νήματα μίας διεργασίας μοιράζονται:

  –τον χώρο διευθύνσεων

  –τους πόρους συστήματος (πχ ανοιχτά αρχεία, semaphores, signals, child processes) της διεργασίας στην οποία περιέχονται

- Όλα τα νήματα έχουν άμεση επικοινωνία μεταξύ τους και έχουν άμεση πρόσβαση σε όλες τις μεταβλητές

- Το νήμα μπορεί να υποστεί διαχείριση:

  ✓ είτε από το λειτουργικό σύστημα (Kernel-level thread)

  ✓ είτε στο χώρο διευθύνσεων μνήμης του χρήστη (User-level thread). Αυτό γίνεται συνήθως σε μια βιβλιοθήκη νημάτων

  ✓ Τα νήματα έχουν πρόσβαση σε μια κοινή περιοχή της μνήμης

  ✓ Η διαδικασία εναλλαγής τους στην ΚΜΕ είναι απλούστερη από την αντίστοιχη διαδικασία εναλλαγή διεργασιών

  ✓ Κάθε νήμα μίας διεργασίας έχει τα δικά του:

    –program counter

–Stack

–Registers

3.3 Διαφορές Διεργασιών-Νημάτων

Νήματα

- Διαμοιράζονται τον χώρο διευθύνσεων, τα δεδομένα, τον κώδικα και τα αρχεία εφόσον ανήκουν στην ίδια διεργασία
- Θεωρούνται ως lightweight σε σχέση με τη δημιουργία, τον τερματισμό και την εναλλαγή πλαισίου
- Ένα νήμα δεν διαθέτει data segment ή heap
- Ένα νήμα δεν μπορεί να υπάρξει μόνο του, πρέπει να υπάρχει στα πλαίσια μιας διεργασίας
- Υπάρχουν περισσότερα από ένα νήματα σε μια διεργασία. Το 1^{ο} νήμα είναι το κύριο και κατέχει τη στοίβα της διεργασίας
- Η δημιουργία νημάτων και η εναλλαγή τους δεν είναι δαπανηρή
- Αν ένα νήμα εκλείψει η στοίβα του επιστρέφεται στους πόρους του συστήματος

Διεργασίες

- Είναι οντότητες που δεσμεύουν οποιονδήποτε πόρο τις αφορά όπως τον χώρο διευθύνσεων, τα δεδομένα, τον κώδικα και τα αρχεία
- Θεωρούνται ως heavyweight σε σχέση με τη δημιουργία, τον τερματισμό και την εναλλαγή πλαισίου
- Μια διεργασία έχει κώδικα, δεδομένα, στοίβα καθώς και άλλα segments
- Υπάρχει ένα τουλάχιστον νήμα σε μια διεργασία
- Τα νήματα σε μια διεργασία διαμοιράζονται κώδικα, δεδομένα, στοίβα και I/O αλλά το καθένα έχει τη δική του στοίβα και τους δικούς του καταχωρητές
- Η δημιουργία των διεργασιών και η εναλλαγή τους είναι δαπανηρές
- Αν μια διεργασία εκλείψει οι πόροι της επιστρέφεται στο σύστημα και όλα τα νήματα της εκλείπουν

3.4 Δημιουργία Διεργασίας

Η δημιουργία μιας διαδικασίας αφορά τη δημιουργία μια νέας διεργασίας που ονομάζεται διεργασία παιδί (child process) από μια πατρική διεργασία (parent process). Μετά τη δημιουργία του παιδιού η πατρική διεργασία συνεχίζει την εκτέλεση της ασύγχρονα ή περιμένει έως ότου τελειώσει η διεργασία παιδί. Η πατρική διεργασία και η διεργασία παιδί θεωρούνται δύο παράλληλες διεργασίες οι οποίες εκτελούνται ταυτόχρονα.

Η βιβλιοθήκη πολλαπλών διεργασιών επιτρέπει τη δημιουργία παράλληλων διεργασιών ακολουθώντας τα επόμενα βήματα:

Ορίζουμε

1. Το αντικείμενο διεργασίας
2. Καλούμε τη συνάρτηση start() της διεργασίας για να την εκτελέσουμε
3. Καλούμε τη συνάρτηση join() της διεργασίας. Η συνάρτηση περιμένει μέχρι να ολοκληρωθεί η διαδικασία και μετά βγαίνει

<u>Κώδικας για Δημιουργία Διεργασιών</u>

```
import multiprocessing
def myFunc(i):
    print ('calling myFunc from process n°: %s' %i)
    for j in range (0,i):
        print('output from myFunc is :%s' %j)

if __name__ == '__main__':
    for i in range(6):
        process = multiprocessing.Process(target=myFunc, args=(i,))
        process.start()
        process.join()
```

<u>Επεξηγήσεις Κώδικα</u>

import multiprocessing #*Εισαγωγή του module multiprocessing για τη δημιουργία διεργασιών*

def myFunc(i): #*Κάθε διεργασία συσχετίζεται με τη συνάρτηση myfunc(i)*

print(′ calling myFunc from process no: %s;′ %i)

for j ***in range***(0,i): #*εκτελείται μια επανάληψη από 0 μέχρι i-1*

print('output from myfunc is :%s' %j) #*η συνάρτηση εμφανίζει τους αριθμούς από 0 μέχρι i, όπου i είναι το ID που σχετίζεται με τον αριθμό διεργασίας*

if__name__==′__main__′:

for i ***in range***(6): #*εκτελείται μια επανάληψη από 0 μέχρι 5*

process = ***multiprocessing.Process***(***target***=myFunc, ***args***=(i,)) #*σε κάθε επανάληψη δημιουργείται μια νέα διεργασία και της ανατίθεται η εκτέλεση της*

συνάρτησης myFunc με όρισμα το i που είναι ο αριθμό της. Άρα το main που είναι η αρχική διεργασία (διεργασία πατέρας) δημιουργεί 6 διεργασίες παιδιά και τους αναθέτει την εκτέλεση της συνάρτησης myFunc

*process.**start**() #ξεκινά η κάθε διεργασία παιδί τη λειτουργία της η οποία περιλαμβάνει την εκτέλεση της συνάρτησης myFunc*

process.***join***() *#η συνάρτηση join είναι απαραίτητη διότι αλλιώς οι διεργασίες παιδιά δεν θα μπορούσαν αν τερματίσουν και θα έπρεπε χειροκίνητα να τις κάνουμε kill*

Αποτελέσματα Εκτέλεσης

C:\>python th2.py
calling myFunc from process n°: 0
calling myFunc from process n°: 1
output from myFunc is :0
calling myFunc from process n°: 2
output from myFunc is :0
output from myFunc is :1
calling myFunc from process n°: 3
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
calling myFunc from process n°: 4
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
output from myFunc is :3
calling myFunc from process n°: 5
output from myFunc is :0
output from myFunc is :1
output from myFunc is :2
output from myFunc is :3
output from myFunc is :4

3.5 Ονοματισμός Διεργασίας

Η μεταβίβαση ορισμάτων για την αναγνώριση και τον ονοματισμό διεργασιών είναι περιττή και άσκοπη. Κάθε διεργασία έχει ένα όνομα και μια προεπιλεγμένη τιμή που

μπορεί να τροποποιείται καθώς δημιουργείται μια διεργασία. Ο ονοματισμός διεργασιών είναι χρήσιμος προκειμένου να τις παρακολουθούμε ειδικά σε εφαρμογές με πολλαπλούς τύπους διαδικασιών που εκτελούνται ταυτόχρονα

import multiprocessing
import time

def worker():
name = ***multiprocessing.current_process***().***name*** *#καταχωρείται στην τοπική μεταβλητή name το όνομα της διεργασίας που παρέχεται από το πεδίο name της συνάρτησης current_process(). Άρα κάθε διεργασία έχει το δικό της όνομα*

print name, 'Starting' *#τυπώνεται το όνομα κάθε διεργασίας ακολουθούμενο από τη λέξη starting*

time.sleep(2) *#κάθε διεργασία σταματά να εκτελείται για 2 msec. Αυτό υποτίθεται ότι είναι η λειτουργία της συνάρτησης*

print name, 'Exiting' *τυπώνεται το όνομα κάθε διεργασίας ακολουθούμενο από τη λέξη Exiting*

def my_service():
name = ***multiprocessing.current_process***().***name*** *#καταχωρείται στην τοπική μεταβλητή name το όνομα της διεργασίας που παρέχεται από το πεδίο name της συνάρτησης current_process(). Άρα κάθε διεργασία έχει το δικό της όνομα*

print name, 'Starting' *#τυπώνεται το όνομα κάθε διεργασίας ακολουθούμενο από τη λέξη starting*

time.sleep(3) *#κάθε διεργασία σταματά να εκτελείται για 3 msec. Αυτό υποτίθεται ότι είναι η λειτουργία της συνάρτησης*

print name, 'Exiting' *τυπώνεται το όνομα κάθε διεργασίας ακολουθούμενο από τη λέξη Exiting*

if __name__ == '__main__':

service = ***multiprocessing.Process***(***name***='my_service', ***target***=my_service) *#δημιουργείται μια νέα διεργασία με όνομα 'my_service' και της ανατίθεται η εκτέλεσης της συνάρτησης my_service*

worker_1 = ***multiprocessing.Process***(***name***='worker 1', ***target***=worker) *#δημιουργείται μια νέα διεργασία με όνομα 'worker1' και της ανατίθεται η εκτέλεσης της συνάρτησης worker*

worker_2 = ***multiprocessing .Process***(***target***=worker) #δ*ημιουργείται μια νέα διεργασία με το προεπιλεγμένο όνομα από το σύστημα και της ανατίθεται η εκτέλεσης της συνάρτησης worker*

#εκκίνηση και των 3 διεργασιών
worker_1.***start***()
worker_2.***start***()
service.***start***()

Αποτελέσματα Εκτέλεσης

$ python multiprocessing_names.py

worker 1 Starting
worker 1 Exiting
Process-3 Starting
Process-3 Exiting
my_service Starting
my_service Exiting

3.6 Εντοπισμός Διεργασίας

Αν θέλουμε σε κάποιο σημείο του κώδικα μας να γνωρίζουμε ποια διεργασία εκτελείται την τρέχουσα χρονική στιγμή θα χρησιμοποιήσουμε τη συνάρτηση current_process() από το module multiprocessing. Αυτή η συνάρτηση χρησιμοποιεί το όρισμα name για να αναγνωρίζει κάθε στιγμή την εκτελούμενη διεργασία

Κώδικας για Εύρεση Εκτελούμενης Διεργασίας

```
import multiprocessing
import time

def myFunc():
    name = multiprocessing.current_process().name
    print ("Starting process name = %s \n" %name)
    time.sleep(3)
    print ("Exiting process name = %s \n" %name)

if __name__ == '__main__':
    process_with_name = multiprocessing.Process(name='myFunc process',target=myFunc)
    process_with_default_name = multiprocessing.Process(target=myFunc)
    process_with_name.start()
    process_with_default_name.start()
    process_with_name.join()
    process_with_default_name.join()
```

Επεξηγήσεις Κώδικα

import multiprocessing *# Εισαγωγή του module multiprocessing για τη δημιουργία διεργασιών*
import time *# Εισαγωγή του module time*

def myFunc(): #*Κάθε διεργασία συσχετίζεται με τη συνάρτηση myFunc(i)*

name = ***multiprocessing.current_process***().***name*** *#Απο το module multiprocessing λαμβάνουμε τη συνάρτηση current_process() που επιστρέφει την τρέχουσα εκτελούμενη διεργασία και το γνώρισμα name περιέχει το όνομα της*

print("Starting process name = %s \n" %name) #*εμφανίζεται το όνομα της εκτελούμενης διεργασίας*

time.sleep(3) #*η εκτελούμενη διεργασία τίθεται σε κατάσταση αναμονής για 3 msec*

print("Exiting process name = %s \n" %name) #*εμφανίζεται το όνομα της διεργασίας που τερματίζει*

if __name__ == '__main__':

process_with_name = ***multiprocessing.Process***(***name***='myFunc process', ***target***=myFunc) #*η διεργασία αυτή λαμβάνει ως όνομα το 'myFunc process' και της ανατίθεται η εκτέλεση της συνάρτησης myFunc*

process_with_default_name = ***multiprocessing.Process***(***target***=myFunc) #*η διεργασία αυτή δεν έχει όνομα (παίρνει ένα εξορισμού όνομα) και της ανατίθεται η εκτέλεση της συνάρτησης myFunc*

process_with_name.***start***() #*η πρώτη διεργασία αρχίζει να εκτελείται*

process_with_default_name.***start***() #*η δεύτερη διεργασία αρχίζει να εκτελείται*

process_with_name.***join***() #*η πρώτη διεργασία τερματίζει*

process_with_default_name.***join***()#*η δεύτερη διεργασία τερματίζει*

4 Γράφοι στην Python

4.1 Βασικοί Ορισμοί Γράφων

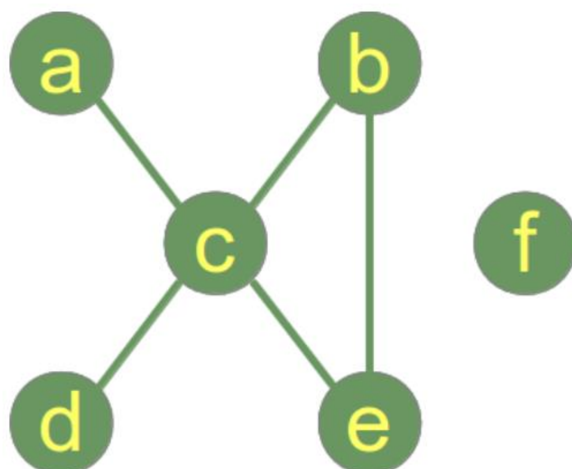
Πριν μελετήσουμε τους τρόπους υλοποίησης των γράφων στην python, θα περιγράψουμε μερικές βασικές έννοιες.

- Γράφος είναι το διατεταγμένο ζεύγος G=(V, E) όπου V σύνολο σημείων και E διμελής σχέση πάνω στο V. Τα στοιχεία του V καλούνται κορυφές (ή κόμβοι). Τα στοιχεία του E καλούνται ακμές οι οποίες είναι ευθύγραμμα ή καμπύλα τμήματα με άκρα ένα ή δύο στοιχεία του συνόλου V.
- Δυο κορυφές λέγονται προσκείμενα εφόσον συνδέονται με μία ή περισσότερες ακμές. Οι προσκείμενες κορυφές αναφέρονται και ως γειτονικές ή ακόμα και ως γειτνιάζουσες
- Μια ακμή μπορεί να έχει την ίδια κορυφή ως αρχή και τέλος. Σε αυτήν την περίπτωση η ακμή καλείται «βρόγχος»
- Ψευδογράφος καλείται ένας γράφος που περιέχει πολλαπλές ακμές ή/και βρόγχους.
- Τα διατεταγμένα ζεύγη G=(V,E) που δεν είναι ψευδογράφοι καλούνται απλοί γράφοι (ή γράφοι)
- Όταν τα στοιχεία του E είναι διατεταγμένα ζεύγη (v1,v2), όπου v1∈V, ο γράφος ονομάζεται προσανατολισμένος(ή κατευθυνόμενος)
- Βαθμός κορυφής είναι ο αριθμός που εκφράζει το πλήθος των παρακείμενων κορυφών της.
- Ο αριθμός των κορυφών του γράφου καλείται τάξη(order) του γράφου.

Ένας γράφος αποτελείται από κόμβους ή κορυφές οι οποίες μπορεί να συνδέονται μεταξύ τους η όχι .Στο γράφο της επόμενης εικόνας η κορυφή A συνδέεται με την κορυφή C αλλά η κορυφή A δεν συνδέεται με την κορυφή B. Η γραμμή που συνδέει δυο κορυφές ονομάζεται ακμή. Αν οι ακμές είναι μη διευθυνόμενες τότε ο γράφος χαρακτηρίζεται συνολικά ως μη διευθυνόμενος, ενώ αν οι ακμές είναι διευθυνόμενες τότε ο γράφος ονομάζεται διευθυνόμενος και οι ακμές του γράφου αυτού ονομάζονται τόξα. Παρόλου που οι γράφοι μοιάζουν ως κάτι πολύ θεωρητικό εντούτοις μπορούμε να αναπαραστήσουμε πολλές πρακτικές εφαρμογές με αυτούς. Για παράδειγμα οι γράφοι χρησιμοποιούνται για την αναπαράσταση δικτύου επικοινωνίας, για την οργάνωση δεδομένων, για την απεικόνιση της ροής ενός υπολογισμού κ.λ.π.

4.2 1º Παράδειγμα Δημιουργίας Γράφου

Έστω ότι δίνεται ο ακόλουθος γράφος:



Εικόνα 1 Παράδειγμα Γράφου

Ο γράφος της προηγούμενης εικόνας μπορεί να υλοποιηθεί με τον ακόλουθο τρόπο που αντιπροσωπεύει τη δομή ενός λεξικού στην Python:
graph ={"a" :["c"], "b":["c", "e"], "c": ["a", "b", "d", "e"],"d":["c"], "e":["c", "b"], "f":[]}

Τα κλειδιά του λεξικού είναι οι κόμβοι του γράφου. Οι αντίστοιχες τιμές κάθε κλειδιού είναι λίστες που περιέχουν κόμβους που συνδέονται με ακμές από την κορυφή αυτή. Αυτός είναι ένα ιδιαίτερα απλός και χρήσιμος τρόπος για την αναπαράσταση ενός γράφου. Μια ακμή μπορεί να αναπαρασταθεί ως μια πλειάδα κόμβων όπως για παράδειγμα:(“a”,”b”). Στο παράδειγμα αυτό έχουμε:

Κλειδιά Λεξικού= Κόμβοι-Κορυφές Γράφου. Άρα οι κόμβοι είναι:

- "a"
- "b"
- "c"
- "d"
- "e"
- "f"

Τιμές Λεξικού= Ακμές Γράφου. Άρα οι ακμές είναι:

- :["c"] μια ακμή που ξεκινά από την κορυφή a
- ["c", "e"] δύο ακμές που ξεκινούν από την κορυφή b

- ["a", "b", "d", "e"]," τέσσερεις που ξεκινούν από την κορυφή c
- ["c"] μια ακμή που ξεκινά από την κορυφή d
- ["c", "b"] δύο ακμές που ξεκινούν από την κορυφή e
- [] καμία ακμή δεν ξεκινά από την κορυφή f

Η συνάρτηση που δημιουργεί τη λίστα όλων των άκμων του προηγούμενου γράφου είναι η ακόλουθη.

def generate_edges(graph):
edges = [] #*λίστα ακμών*

for node ***in*** graph: #*εκτελείται επανάληψη για κάθε κορυφή του γράφου. Σε κάθε επανάληψη καταχωρείται στη μεταβλητή node ένα στοιχείο από το λεξικό graph*

for neighbour ***in graph***[node]:#*εκτελείται μια εμφωλευμένη επανάληψη για κάθε κορυφή του γράφου. Σε κάθε επανάληψη καταχωρείται στη μεταβλητή* neighbour *ένα στοιχείο από τη λίστα κορυφών αυτού του κλειδιού*

edges.***append***((node, neighbour)) #*στη λίστα edges προστίθεται μια ακμή σε κάθε επανάληψη*
return edges #*επιστρέφεται η λίστα ακμών*

print(generate_edges(graph)) #*στο κύριο πρόγραμμα εκτυπώνεται η λίστα ακμών*

Η εκτέλεση του κώδικα αυτού παράγει το ακόλουθο αποτέλεσμα:

Αποτελέσματα Εκτέλεσης

[('a', 'c'), ('c', 'a'), ('c', 'b'), ('c', 'd'), ('c', 'e'), ('b', 'c'), ('b', 'e'), ('e', 'c'), ('e', 'b'), ('d', 'c')]

Όπως παρατηρούμε δεν υπάρχει καμία ακμή που συνδέει την κορυφή f. Η κορυφή f είναι μια **απομονωμένη κορυφή** (isolated node) του γράφου.

Η επόμενη συνάρτηση python υπολογίζει τις απομονωμένες κορυφές του γράφου.

#Η συνάρτηση αυτή επιστρέφει μια λίστα απομονωμένων κορυφών
def find_isolated_nodes(graph):
isolated = []

for node ***in*** graph:
if not graph[node]: # *αν η κορυφή δεν περιέχεται στο γράφο graph που δίνεται ως όρισμα τότε η κορυφή αυτή συνενώνεται στη μεταβλητή isolated*
isolated += node

return isolated

4.3 Οι γράφοι ως μια κλάση τις python

Ας δούμε τώρα το πως υλοποιείται μια κλάση στην python. Ο επόμενος κώδικας περιέχει ένα δημιουργό δηλ. τη μέθοδο __init__method για την αποθήκευση των κορυφών και των αντίστοιχων γειτονικών κορυφών τους def find_isolated_nodes(graph):

class Graph(object):
def __init__(***self***, graph_dict=***None***): # *Η κλάση αυτή στην Python δείχνει τα βασικά στοιχεία και τις λειτουργίες των γραφημάτων και αρχικοποιεί ένα graph object. Αν δεν δίνεται λεξικό ή δίνεται None. Η συνάρτηση αυτή είναι ο δημιουργός της κλάσης Graph*
if graph_dict == ***None***:
graph_dict = { }

self.__graph_dict = graph_dict #*στο μέλος graph_dict καταχωρείται το όρισμα graph_dict. Το μέλος graph_dict είναι ένα λεξικό. Οι γράφοι στην Python δημιουργούνται μέσω λεξικού*

def vertices(self): #*ορίζουμε τη μέθοδο vertices που επιστρέφει τις κορυφές του γράφου*
return list(***self***.__graph_dict.***keys***()) //*για το μέλος graph_dict καλείται η συνάρτηση keys() του λεξικού που επιστρέφει όλα τα κλειδιά δηλ. τις κορυφές του γράφου*

def edges(self): #*επιστρέφει τις ακμές του γράφου*
return self.__generate_edges()

def add_vertex(***self***, vertex): #*συνάρτηση προσθήκης νέας κορυφής στο γράφο. Κάθε κορυφή είναι ένα κλειδί στο λεξικό. Αν η κορυφή vertex δεν βρίσκεται στο λεξικό self.__graph_dict με τις κορυφές τότε μια νέα κορυφή "vertex" με μια κενή λίστα ακμών ως τιμή προστίθεται στο λεξικό*
if vertex ***not in self***.__graph_dict:
self.__graph_dict[vertex] = []

```
def add_edge(self, edge): # συνάρτηση προσθήκης νέας ακμής στο γράφο.
Υποθέτουμε ότι μια ακμή είναι τύπου set, tuple ή list. Μεταξύ δύο κορυφών μπορεί να
υπάρχουν πολλαπλές ακμές
        edge = set(edge) #στο μέλος edge καταχωρείται το όρισμα edge
        (vertex1, vertex2) = tuple(edge) #δημιουργείται μια νέα ακμή ως μια
tuple που συνδέει τις κορυφές vertex1 και vertex2

        if vertex1 in self.__graph_dict: #αν η κορυφή vertex1 ανήκει στο
λεξικό
            self.__graph_dict[vertex1].append(vertex2)# τότε στη λίστα ακμών
της κορυφής προστίθεται η κορυφή vertex2 δηλ. προστίθεται συνολικά η ακμή (vertex1,
vertex2)
        else:
            self.__graph_dict[vertex1] = [vertex2] αν η κορυφή vertex1 δεν
ανήκει στο λεξικό τότε προστίθεται στο λεξικό ως κλειδί με τιμή vertex2

def __generate_edges(self): #Μια στατική μέθοδος δημιουργίας των άκρων του
γραφήματος "γράφημα". Τα άκρα αντιπροσωπεύονται ως σύνολα με ένα ή δύο κορυφές
(βρόχο πίσω στην κορυφή)
        edges = []
        for vertex in self.__graph_dict: #εκτελείται επανάληψη για κάθε κλειδί
του λεξικού δηλ. για κάθε κορυφή
            for neighbour in self.__graph_dict[vertex]: εκτελείται επανάληψη
για κάθε άλλο κλειδί του λεξικού δηλ. για κάθε άλλη κορυφή
                if {neighbour, vertex} not in edges: #αν αυτό το ζεύγος
κορυφών δεν είναι ακμή τότε με τη συνάρτηση append προστίθεται στις ακμές του
γράφου
                    edges.append({vertex, neighbour})
        return edges

def __str__(self): #επιστρέφει όλο το γράφο ως ένα string
        res = "vertices: "
        for k in self.__graph_dict: #σε αυτή την επανάληψη συνενώνουμε όλες
τις κορυφές στο αλφαριθμητικό res
            res += str(k) + " "
       res += "\nedges: "

        for edge in self.__generate_edges():σε αυτή την επανάληψη
συνενώνουμε όλες τις ακμές στο αλφαριθμητικό res

          res += str(edge) + " "
```

```
        return res #επιστρέφεται όλος ο γράφος ως ένα string

if __name__ == "__main__":
g={"a" : ["d"],"b" : ["c"],"c" : ["b", "c", "d", "e"], "d" : ["a", "c"],"e" : ["c"],"f" : []}

graph = Graph(g)

print("Vertices of graph:")
print(graph.vertices())

print("Edges of graph:")
print(graph.edges())

print("Add vertex:")
graph.add_vertex("z")

print("Vertices of graph:")
print(graph.vertices())

print("Add an edge:")
graph.add_edge({"a", "z"})

print("Vertices of graph:")
print(graph.vertices())

print("Edges of graph:")
print(graph.edges())

print('Adding an edge {"x", "y"} with new vertices:')
graph.add_edge({"x", "y"})

print("Vertices of graph:")
print(graph.vertices())
print("Edges of graph:")
print(graph.edges())
```

Αποτελέσματα Εκτέλεσης

$ python3 graph.py
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f']
Edges of graph:
[{'a', 'd'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}]

Add vertex:
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f', 'z']

Add an edge:
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f', 'z']
Edges of graph:
[{'a', 'd'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'a', 'z'}]

Adding an edge {"x", "y"} with new vertices:

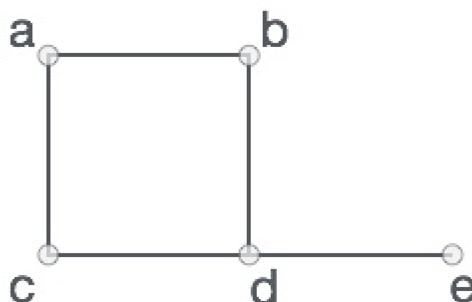
Vertices of graph:
['a', 'c', 'b', 'e', 'd', 'f', 'y', 'z']

Edges of graph:
[{'a', 'd'}, {'c', 'b'}, {'c'}, {'c', 'd'}, {'c', 'e'}, {'a', 'z'}, {'y', 'x'}]

4.4 2ο Παράδειγμα Δημιουργίας Γράφου

Ένας εναλλακτικός που μπορεί να δοθεί στη έννοια του γράφου είναι ότι αποτελεί μία αναπαράσταση ενός συνόλου αντικειμένων, όπου μερικά ζεύγη αντικειμένων συνδέονται μέσω συνδέσμων.

Τα διασυνδεδεμένα αντικείμενα αντιπροσωπεύονται από σημεία που ορίζονται ως κορφές ενώ οι σύνδεσμοι που συνδέουν τις κορφές αυτές ονομάζονται ακμές. Ένας γράφος μπορεί να αναπαρασταθεί χρησιμοποιώντας τον τύπο δεδομένων του λεξικού. Αντιπροσωπεύουμε τις κορφές ως τα κλειδιά του λεξικού(keys) και τη σύνδεση μεταξύ των κορφών, τις επονομαζόμενες κορφές, ως τιμές του λεξικού. Ας θεωρήσουμε τον ακόλουθο γράφο



Εικόνα 2 Παράδειγμα Γράφου

Στον γράφο αυτό το σύνολο των κορυφών είναι το v = {a,b,c,d,e} και το σύνολο των ακμών είναι E = {ab,ac,bd,cd,de). Μπορούμε να δημιουργήσουμε τον γράφο αυτό με τον ακόλουθο κώδικα.

```
# Δημιουργία Γράφου
graph={"a":["b", "c"], "b": ["a", "d"], "c" : ["a", "d"], "d" : ["e"], "e":["d"]}

#Εκτύπωση του Γράφου
print(graph)
```

Εκτελώντας τον προηγούμενο κώδικα παράγεται το ακόλουθο αποτέλεσμα:

{'c': ['a', 'd'], 'a': ['b', 'c'], 'e': ['d'], 'd': ['e'], 'b': ['a', 'd']}

4.4.1 Απεικόνιση Κορυφών Γράφου

Για την απεικόνιση των ακμών του γράφου εντοπίζουμε απλώς τα κλειδιά (keys) του λεξικού του γράφου χρησιμοποιώντας τη μέθοδο keys().

```
class graph:
    def __init__(self, gdict=None):
        if gdict is None:
            gdict = []
        self.gdict = gdict

        # Λήψη Κλειδιών Λεξικού
        def getVertices(self):
            return list(self.gdict.keys())

# Δημιουργία του λεξικού με τα γραφικά αντικείμενα
graph_elements = { "a" : ["b", "c"], "b" : ["a", "d"], "c" : ["a", "d"], "d" : ["e"], "e":["d"]}
g = graph(graph_elements)
print(g.getVertices())
```

Η εκτέλεση του προηγούμενου κώδικα δίνει το ακόλουθο αποτέλεσμα:

['d', 'b', 'e', 'c', 'a']

4.4.2 Απεικόνιση Ακμών Γράφου

Για τον εντοπισμό των ακμών γράφου πρέπει να σκεφτούμε λίγο διαφορετικά καθώς είναι πιο δύσκολο να εντοπίσουμε όλα τα ζεύγη κορυφών που συνδέονται μεταξύ τους με ακμή. Αρχικά δημιουργούμε μια κενή λίστα ακμών και μετά εκτελούμε μια επανάληψη για τις ακμές που συσχετίζονται με αυτές τις κορυφές. Δημιουργείται μια λίστα που περιέχει το διακριτό σύνολο ακμών που προκύπτουν από τις κορυφές αυτές.

```
class graph:
    def __init__(self,gdict=None):
        if gdict is  None:
            gdict = {}
        self.gdict = gdict

    def edges(self):
        return self.findedges()
```

```
#Εύρεση Διακριτής Λίστας Ακμών
def findedges(self):
    edgename = []
    for vrtx in self.gdict:
        for nxtvrtx in self.gdict[vrtx]:
            if {nxtvrtx, vrtx} not in edgename:
                edgename.append({vrtx, nxtvrtx})
    return edgename

#Δημιουργία Λεξικού με τα αντικείμενα του γράφου
graph_elements ={"a": ["b", "c"], "b": ["a", "d"], "c":["a", "d"], "d" :["e"], "e":["d"]}

g = graph(graph_elements)
print(g.edges())
```

Αποτελέσματα Εκτέλεσης

[{'b', 'a'}, {'b', 'd'}, {'e', 'd'}, {'a', 'c'}, {'c', 'd'}]

4.4.3 Προσθήκης Νέας Κορυφής στο Γράφο

Η προσθήκη μιας νέας κορυφής στο γράφο μπορεί να γίνει άμεσα απλά προσθέτοντας ένα επιπλέον κλειδί στο λεξικό του γράφου.

```
class graph:
    def __init__(self,gdict=None):
        if gdict is None:
            gdict = { }
        self.gdict = gdict

    def getVertices(self):
        return list(self.gdict.keys())

    # Προσθήκη Κορυφής ως κλειδί
    def addVertex(self, vrtx):
        if vrtx not  in self.gdict:
            self.gdict[vrtx] = []

#Δημιουργία του dictionary με τα στοιχεία του γράφου
graph_elements ={ "a":["b", "c"],"b": "a", "d"], "c":["a", "d"], "d":["e"],"e" :["d"]}

g = graph(graph_elements)

g.addVertex("f")

print(g.getVertices())
```

Η εκτέλεση του προηγούμενου κώδικα παράγει το ακόλουθο αποτέλεσμα:

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

4.4.4 Προσθήκης Νέας Ακμής στο Γράφο

Η Προσθήκη μιας νέας ακμής σε ένα υπάρχον γράφο αφορά τη 'μεταχείριση' της νέας κορυφής ως μια πλειάδα (tuple) και τον έλεγχο αν η ακμή είναι ήδη παρούσα ή όχι. Αν δεν είναι παρούσα τότε προστίθεται στο γράφο

```
class graph:
    def__init__(self,gdict=None):
        if gdict is  None:
            gdict = {}
        self.gdict = gdict

    def edges(self):
        return self.findedges()

    #Προσθήκη της νέας ακμής
    def AddEdge(self, edge):
        edge = set(edge)
        (vrtx1, vrtx2) = tuple(edge)
        if vrtx1 in self.gdict:
          self.gdict[vrtx1].append(vrtx2)
        else:
          self.gdict[vrtx1] = [vrtx2]

     #Λίστα των ονομάτων ακμών
     def findedges(self):
         edgename = []
         for vrtx in self.gdict:
             for nxtvrtx in self.gdict[vrtx]:
                 if {nxtvrtx, vrtx} not in edgename:
                     edgename.append({vrtx, nxtvrtx})
         return edgename

#Δημιουργία Λεξικού με Γραφικά Αντικείμενα
graph_elements={"a" : ["b", "c"],"b" : ["a", "d"], "c" : ["a", "d"],"d" : ["e"],"e" : ["d"]}
g = graph(graph_elements)
g.AddEdge({'a', 'e'})
g.AddEdge({'a', 'c'})
print(g.edges())
```

Αποτελέσματα Εκτέλεσης

[{'e', 'd'}, {'b', 'a'}, {'b', 'd'}, {'a', 'c'}, {'a', 'e'}, {'c', 'd'}]

4.5 Μονοπάτια στην Python.

Θέλουμε να εντοπίσουμε το συντομότερο μονοπάτι από μια κορυφή σε άλλη. Πριν δούμε τον κώδικα python που επιλύει το συγκεκριμένο πρόβλημα θα πρέπει να προσθέσουμε μερικούς τυπικούς ορισμούς.

Προσκείμενες κορυφές (adjacent vertices)

Δυο κορυφές είναι γειτονικές αν συνδέονται με ακμή.

Μονοπάτι σε μη διευθυνόμενο γράφο.

Ένα μονοπάτι σε ένα μη διευθυνόμενο γράφο είναι μια ακολουθία κορυφών $P=(V_1, V_2, \ldots, V_n) \in V \times V \times \ldots \times V$ έτσι ώστε η κορυφή vi να είναι γειτονική στη κορυφή v_{i+1} για $1 \leq i < n$. Ένας τέτοιος γράφος ονομάζεται μονοπάτι μήκους n από την κορυφή v_1 στην κορυφή v_n.

Απλή διαδρομή

Μια διαδρομή χωρίς επαναλαμβανόμενες κορυφές ονομάζεται απλή διαδρομή

Παράδειγμα

Η (a, c, e) είναι μια απλή διαδρομή στο γράφημα μας, ενώ η (a, c, e, b). (a, c, e, b, c, d) είναι μια διαδρομή αλλά όχι μια απλή διαδρομή, επειδή ο κόμβος c εμφανίζεται δύο φορές.

Η επόμενη μέθοδος εντοπίζει ένα μονοπάτι από μια αρχική κορυφή

```
def find_path(self, start_vertex, end_vertex, path=None): #εντοπίζει ένα μονοπάτι από μια start_vertex στην end_vertex στο γράφο
    if path == None:
        path = []

    graph = self.__graph_dict
    path = path + [start_vertex]

    if start_vertex == end_vertex:
        return path

    if start_vertex not in graph:
        return None

    for vertex in graph[start_vertex]:
        if vertex not in path:
            extended_path = self.find_path(vertex, end_vertex, path)
```

```
            if extended_path:
                return extended_path

        return None
```

Εάν αποθηκεύσουμε την κλάση γραφημάτων, συμπεριλαμβανομένης της μεθόδου find_path ως "graphs.py", μπορούμε να ελέγξουμε τον τρόπο λειτουργίας της συνάρτησης find_path:

```
from graphs import Graph
g={"a" : ["d"], "b": ["c"], "c" : ["b", "c", "d", "e"], "d" : ["a", "c"], "e":["c"],"f":[]}

graph = Graph(g)

print("Vertices of graph:")
print(graph.vertices())

print("Edges of graph:")
print(graph.edges())

print('The path from vertex "a" to vertex "b":')
path = graph.find_path("a", "b")
print(path)

print('The path from vertex "a" to vertex "f":')
path = graph.find_path("a", "f")
print(path)

print('The path from vertex "c" to vertex "c":')
path = graph.find_path("c", "c")
print(path)
```

Αποτελέσματα Εκτέλεσης

Vertices of graph:
['e', 'a', 'd', 'f', 'c', 'b']
Edges of graph:
[{'e', 'c'}, {'a', 'd'}, {'d', 'c'}, {'b', 'c'}, {'c'}]
The path from vertex "a" to vertex "b":
['a', 'd', 'c', 'b']
The path from vertex "a" to vertex "f":
None
The path from vertex "c" to vertex "c":
['c']

Η μέθοδος find_all_paths βρίσκει όλες τις διαδρομές μεταξύ μιας κορυφής έναρξης προς μια κορυφή κορυφής:

```
def find_all_paths(self, start_vertex, end_vertex, path=[]):#εύρεση όλων των μονοπατιών στο γράφο από την κορυφή start_vertex στην κορυφή end_vertex
    graph = self.__graph_dict
    path = path + [start_vertex]

    if start_vertex == end_vertex:
        return [path]

    if start_vertex not in graph:
        return []

    paths = [] #πίνακας μονοπατιών
    for vertex in graph[start_vertex]:
        if vertex not in path:
            extended_paths = self.find_all_paths(vertex, end_vertex, path)

            for p in extended_paths:
                paths. Append(p)
    return paths
```

Τροποποιούμε το προηγούμενο παράδειγμα προσθέτοντας άκρα από "a" σε "f" και από "f" σε "d" για να δοκιμάσουμε την προηγούμενη μέθοδο. Ο νέος κώδικας που προκύπτει είναι ο ακόλουθος:

```
from graphs import Graph
g={"a":["d", "f"], "b":["c"], "c":["b", "c", "d", "e"], "d":["a", "c"], "e" : ["c"], "f":["d"]}

graph = Graph(g)

print("Vertices of graph:")
print(graph.vertices())

print("Edges of graph:")
print(graph.edges())

print('All paths from vertex "a" to vertex "b":')
path = graph.find_all_paths("a", "b")
print(path)

print('All paths from vertex "a" to vertex "f":')
path = graph.find_all_paths("a", "f")
print(path)

print('All paths from vertex "c" to vertex "c":')
path = graph.find_all_paths("c", "c")
print(path)
```

Αποτελέσματα Εκτέλεσης

```
Vertices of graph:
['d', 'c', 'b', 'a', 'e', 'f']
Edges of graph:
[{'d', 'a'}, {'d', 'c'}, {'c', 'b'}, {'c'}, {'c', 'e'}, {'f', 'a'}, {'d', 'f'}]
All paths from vertex "a" to vertex "b":
[['a', 'd', 'c', 'b'], ['a', 'f', 'd', 'c', 'b']]
All paths from vertex "a" to vertex "f":
[['a', 'f']]
All paths from vertex "c" to vertex "c":
[['c']]
```

5 Ασκήσεις με Νήματα και Σημαφόρους

5.1 Άσκηση 1

Υλοποιείστε ένα πρόγραμμα σε Python το οποίο με τον μεγαλύτερο δυνατό βαθμό παραλληλισμού να υλοποιεί τις παρακάτω πράξεις. Αρχικά θα πρέπει να σχεδιαστεί ο γράφος προτεραιότητας και στην συνέχεια να υλοποιηθεί το πρόγραμμα και τα κατάλληλα threads χρησιμοποιώντας τον κατάλληλο αριθμό σημαφόρων.

E1: A:=10;

E2: B:=20;

E3: C:=A + 5;

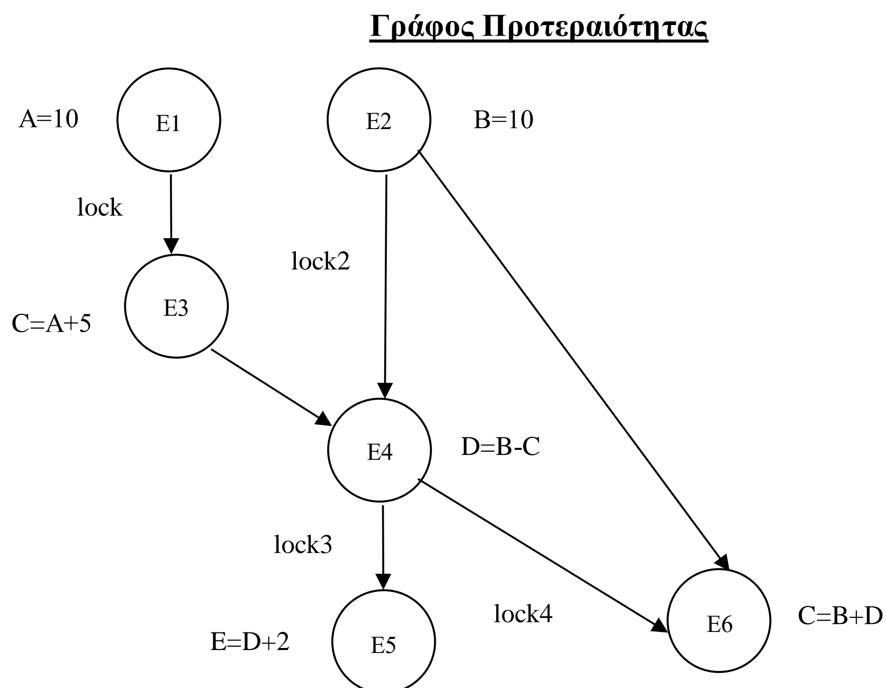
E4: D:=B – C;

E5: E:=D+2;

E6: C:=B+D;

<u>Απάντηση</u>

<u>Γράφος Προτεραιότητας</u>



Ψευδο-Κώδικας συγχρονισμού νημάτων με χρήση σημαφόρων

var

lock, lock2, lock3, lock4: ***semaphores***;

A, B, C, D : ***shared integer***;

lock=lock2=lock3=lock4=0;

cobegin

E1:***begin*** A=10; ***end***;

E2:***begin*** B=10; ***end***;

coend;

Υλοποίηση σε Python

```
import threading #Εισαγωγή του module threading που περιλαμβάνει συναρτήσεις
διαχείρισης νημάτων
import random, time
from threading import Thread

lock = threading.Lock() #δημιουργία 1ου κλειδιού
lock2 = threading.Lock()#δημιουργία 2ου κλειδιού
lock3 = threading.Lock()#δημιουργία 3ου κλειδιού
lock4 = threading.Lock()#δημιουργία 4ου κλειδιού

#αρχικοποίηση μεταβλητών
A=0
B=0
C=0
D=0
E=0

def E1():
    lock.acquire() #κλείδωμα του κλειδιού lock
    print('\n YPOLOGISMOS A')
    global A
    A=10
    print('A=', A, end='') #εκτύπωση A
    lock.release() #ξεκλείδωμα του lock
```

```
def E2():
    time.sleep(1) #θέτουμε μια χρονοκαθυστέρηση για λόγους εμφάνισης των αποτελεσμάτων στην οθόνη. Κάθε αποτέλεσμα θέλουμε να τυπώνεται μια μικρή καθυστέρηση σε σχέση με τα προηγούμενα
    lock2.acquire() #κλείδωμα του κλειδιού lock
    print('\n YPOLOGISMOS B')
    global B
    B=20
    print('B=', B, end='')
    lock2.release() #ξεκλείδωμα του κλειδιού lock2

def E3():
    time.sleep(2)
    lock.acquire() #κλείδωμα του κλειδιού lock
    global A, B, C
    print('\n YPOLOGISMOS C')
    C=A+5 #υπολογισμός C
    print('C=', C, end='')
    lock.release() #ξεκλείδωμα του κλειδιού lock

def E4():
    time.sleep(3)
    lock2.acquire() #κλείδωμα του κλειδιού lock2
    print('\n YPOLOGISMOS D')
    global A,B,C,D
    D=B-C #υπολογισμός D
    print('D=', D, end='')
    lock2.release() #ξεκλείδωμα του κλειδιού lock2

def E5():
    time.sleep(4)
    lock3.acquire() #κλείδωμα του κλειδιού lock3
    print('\n YPOLOGISMOS E')
    global A,B,C,D,E
    E=D+2#υπολογισμός E
    print('E=', E, end='')
    lock3.release() #ξεκλείδωμα του κλειδιού lock3
```

```
def E6():
    time.sleep(5)
    lock4.acquire() #κλείδωμα του κλειδιού lock4
    print('\n EPANAYPOLOGISMOS A')
    global B, C, D
    C=B+D #εκ νέου υπολογισμός του C
    print('C=', C, end='')
    lock4.release() #ξεκλείδωμα του κλειδιού lock4

def main(): #ορισμός κύριου προγράμματος (συνάρτηση main)
     t1= threading.Thread(target=E1) #δημιουργία νήματος t1 και ανάθεση της συνάρτησης E1 σε αυτό

     t2= threading.Thread(target=E2) #δημιουργία νήματος t2 και ανάθεση της συνάρτησης E2 σε αυτό

     t3= threading.Thread(target=E3) #δημιουργία νήματος t3 και ανάθεση της συνάρτησης E3 σε αυτό

     t4= threading.Thread(target=E4) #δημιουργία νήματος t4 και ανάθεση της συνάρτησης E4 σε αυτό

     t5= threading.Thread(target=E5) #δημιουργία νήματος t5 και ανάθεση της συνάρτησης E5 σε αυτό

     t6= threading.Thread(target=E6) δημιουργία νήματος t6 και ανάθεση της
συνάρτησης E6 σε αυτό

     t1.start() #εκκίνηση νήματος t1
     t2.start() #εκκίνηση νήματος t2
     t3.start() #εκκίνηση νήματος t3
     t4.start() #εκκίνηση νήματος t4
     t5.start() #εκκίνηση νήματος t5
     t6.start() #εκκίνηση νήματος t6

     t1.join() #συνένωση νήματος t1
     t2.join() #συνένωση νήματος t2
     t3.join( )#συνένωση νήματος t3
     t4.join() #συνένωση νήματος t4
     t5.join() #συνένωση νήματος t5
     t6.join() #συνένωση νήματος t6
```

```
if __name__ == "__main__": #με την εντολή αυτή εκτελείται αυτόματα η συνάρτηση
    main()
```

Αποτελέσματα Εκτέλεσης

YPOLOGISMOS A
A= 10
YPOLOGISMOS B
B= 20
YPOLOGISMOS C
C= 15
YPOLOGISMOS D
D= 5
YPOLOGISMOS E
E= 7
EPANAYPOLOGISMOS A
C= 25

5.2 Άσκηση 2

Φανταστείτε πως υπάρχουν 3 νήματα A, B, C, τα οποία εκτελούν τις συναρτήσεις study(), drink_coffee() και take_exam() αντίστοιχα. Χρησιμοποιώντας μόνο ένα σημαφόρο, να γράψετε πρόγραμμα σε Python το οποίο να εξασφαλίζει πως τα νήματα A και B θα εκτελεστούν πριν από το νήμα C.

Απάντηση

```
import threading #Εισαγωγή του module threading που περιλαμβάνει συναρτήσεις διαχείρισης νημάτων
import random, time
from threading import BoundedSemaphore, Thread

max_items = 2
container = BoundedSemaphore(max_items) #δημιουργία σημαφόρου

def study(): #υλοποίηση συνάρτησης study
    print('study now\n') #εκτύπωση μηνύματος

def drink_coffee(): #υλοποίηση συνάρτησης drink_coffee
    print('drink coffee\n') #εκτύπωση μηνύματος

def take_exam(): #υλοποίηση συνάρτησης take_exam
    container.acquire() #κατεβάζει το σημαφόρο container
    container.acquire() #κατεβάζει το σημαφόρο container
    container.release() #ανεβάζει το σημαφόρο container
    container.release() #ανεβάζει το σημαφόρο container
    print('take exam\n') #εκτύπωση μηνύματος

def main(): #ορισμός κύριου προγράμματος (συνάρτηση main)
    threads = [] #δήλωση πίνακα νημάτων

    #δημιουργία των 3 νημάτων
    t1= threading.Thread(target=study) #δημιουργία 1ου νήματος και ανάθεση της συνάρτησης study σε αυτό

    t2= threading.Thread(target=drink_coffee) #δημιουργία 2ου νήματος και ανάθεση της συνάρτησης drink_coffee σε αυτό

    t3= threading.Thread(target=take_exam) #δημιουργία 3ου νήματος και ανάθεση της συνάρτησης take_exam σε αυτό
```

```
    #εκκίνηση των 3 νημάτων

    t1.start()
    t2.start()
    t3.start()

    t1.join()
    t2.join()
    t3.join()

if __name__ == "__main__":
    main() #με την εντολή αυτή καλείται και εκτελείται αυτόματα η συνάρτηση main
```

Αποτελέσματα Εκτέλεσης

study now

drink coffee

take exam

5.3 Άσκηση 3

Να γραφεί πρόγραμμα σε Python το οποίο να δημιουργεί N νήματα (το N είσοδος από το πληκτρολόγιο). Το κάθε νήμα θα πρέπει να επιλέγει δύο τυχαίους πραγματικούς αριθμούς α, β στο διάστημα 0-10. Το νήμα θα «κοιμάται» για α secs και μετά προσθέτει σε μία κοινή μεταβλητή sum τον αριθμό β. Η sum θα πρέπει να εκτυπώνεται μόνο όταν όλα τα νήματα έχουν προσθέσει την μεταβλητή β τους.

Απάντηση

```
import threading
import time
import random
from threading import Thread

#δημιουργία σημαφόρου
lock = threading.Lock()

sum=0

def sumall(i):
    lock.acquire() #ο σημαφόρος κατεβαίνει για την προστασία της καθολικής μεταβλητής sum

    a=(int)(random.randint(0,10)) #παραγωγή τυχαίας τιμής από 0 έως 10
    b=random.randint(0,10) #παραγωγή τυχαίας τιμής από 0 έως 10

    print("Thread ",i," sleeps for ",a," seconds") #εκτύπωση αριθμού νήματος και του
χρόνου που κάνει sleep
    print("Thread ",i," adds ",b," as value") #εκτύπωση αριθμού νήματος και της τιμής
που θα προσθέσεις

    time.sleep(a) #το νήμα "κοιμάται" για a second
    global sum

    sum=sum+b #πρόσθεση του b στο άθροισμα

    print("sum=", sum)

    lock.release() #ο σημαφόρος ανεβαίνει

    #Εισαγωγή πλήθους νημάτων
    T= int(input("Enter number of threads: "))
```

```
    threads = [] #δήλωση πίνακα νημάτων
    for n in range(T): #εκτέλεση επανάληψης για κάθε νήμα
        t = Thread(target=sumall, args=(n,)) #δημιουργία κάθε νήματος και ανάθεση της συνάρτησης sumall σε αυτό
        t.start() #εκκίνηση κάθε νήματος
        threads.append(t) #προσθήκη κάθε νήματος στον πίνακα νημάτων

    #Αναμονή τερματισμού νημάτων
    for t in threads:
        t.join()
```

Αποτελέσματα Εκτέλεσης

```
Enter number of threads: 5

Thread  0  sleeps for  9  seconds
Thread  0  adds  4  as value
sum= 4
Thread  1  sleeps for  0  seconds
Thread  1  adds  4  as value
sum= 8
Thread  2  sleeps for  6  seconds
Thread  2  adds  5  as value
sum= 13
Thread  3  sleeps for  7  seconds
Thread  3  adds  3  as value
sum= 16
Thread  4  sleeps for  9  seconds
Thread  4  adds  7  as value
sum= 23
```

5.4 Άσκηση 4

Να γραφεί πρόγραμμα σε Python το οποίο προσομοιώνει ένα μανάβικο, ως εξής:
Ένας μανάβης (νήμα) προμηθεύεται τέσσερα είδη φρούτων (μήλα, πορτοκάλια, αχλάδια και μπανάνες) και να διαθέτει προς πώληση. Η διαθεσιμότητα των καλαθιών είναι 100 τεμάχια για κάθε φρούτο και ο μανάβης γεμίζει τα καλάθια κάθε δύο λεπτά. Οι καταναλωτές (νήματα) μπορούν να έρχονται στο μανάβικο οποιαδήποτε στιγμή (τυχαίος αριθμός μεταξύ 1 και 30 sec) και να αγοράζουν το επιθυμητό φρούτο τους (n τεμάχια από τυχαίο είδος φρούτου, όπου n τυχαίος αριθμός στο διάστημα 10-30). Αν τα φρούτα που θέλουν να αγοράσουν έχουν τελειώσει, περιμένουν έως ότου η προμήθεια αυτού του είδους είναι έτοιμη. Η προσομοίωση να τερματίζει όταν ο μανάβης γεμίσει τα καλάθια 10 φορές. Σε κάθε γεγονός (γέμισμα καλαθιών, άφιξη πελάτη, αγορά, αναμονή) να τυπώνονται κατάλληλα μηνύματα. Σημειώνεται πως κάθε χρονική στιγμή μόνο ένας πελάτης μπορεί να αγοράζει φρούτα, ενώ οι υπόλοιποι περιμένουν είτε την σειρά τους είτε τον ανεφοδιασμό των καλαθιών.

<u>Απάντηση</u>

```
import sys
import time
import random
import threading
import queue

#δήλωση dictionary με φρούτα
baskets = {
   'apples': 100,
   'oranges': 100,
   'pears': 100,
   'bananas': 100,
}

basket_sema = threading.BoundedSemaphore(1) #δημιουργία σημαφόρου

client_queue = queue.Queue() #Κλήση Κατασκευαστή για μια ουρά FIFO. Επειδή δεν καθορίζεται μέγεθος ουράς, το οριζόμενο μέγεθος ουράς είναι άπειρο

has_fruit = threading.Event() #εξορισμού συνάρτηση που επιστρέφει ένα νέο αντικείμενο συμβάντος. Ένα συμβάν διαχειρίζεται μια σημαία (flag) που μπορεί να οριστεί σε true με τη μέθοδο set() και να επαναρυθμιστεί σε false με τη μέθοδο clear(). Η μέθοδος wait() μπλοκάρει έως ότου to το flag γίνει true

do_work = True #αρχικοποίηση με true ώστε να ξεκινήσει
refills = 10
```

```
def thread_grocer():
    global do_work #δήλωση καθολικής μεταβλητής do_work
    global refills #δήλωση καθολικής μεταβλητής refills

    while refills > 0: #όσο υπάρχουν ακόμα refills για να γίνουν (το πρόγραμμα σταματά στα 10 refills)
        time.sleep(120) #κάνουμε sleep το πρόγραμμα για 2 min=120sec

        with basket_sema: #με το σημαφόρο basket_sema κάνε
            for b in baskets:
                baskets[b] = 100 #θέτει όλες τις ποσότητες στο 100 στο καλάθι σε κάθε είδος
            refills -= 1 #σε κάθε νέο γέμισμα refill μειώνεται η μεταβλητή refills κατά 1
            print("Grocer: Refilled baskets. {} refills remaining".format(refills)) #δείχνει τα εναπομείναντα refills

            print("Baskets contains ", baskets)

            has_fruit.set() #ο grocer ενημερώνει το Thread του πελάτη ότι το καλάθι έχει φρούτα

        do_work =False #θέτουμε στη do_work false για να τελειώσει το πρόγραμμα

        return

def thread_dispatcher(): #το νήμα αυτό βάζει στην ουρά τους πελάτες
    while do_work: #όσο η μεταβλητή do_work είναι true δηλ. όσο γίνονται refills
        c = threading.Thread(target=thread_client) #δημιουργία ενός νέου πελάτη (για την ακρίβεια ενός νέου νήματος πελάτη)

        client_queue.put(c) #ο πελάτης μπαίνει στην ουρά

        print("Dispatcher: There are {} clients waiting".format(client_queue.qsize())) #τυπώνεται το μέγεθος ουράς

        w = random.randint(1, 30) #παράγεται ένας τυχαίος integer από 1 μέχρι 30. Είναι ο χρόνος άφιξης του επόμενου πελάτη

        print("Client: Next client in arrives in {} seconds".format(w)) #τυπώνεται ο χρόνος άφιξης του επόμενου πελάτη
```

```
        while w>0: #όσο υπολείπεται χρόνος για τον πελάτη
            if not do_work: #αν η do_work γίνει false τερματίζει
                return

            time.sleep(0.2) #κάθε 0.2 sec ελέγχεται η do_work έτσι ώστε σε
        περίπτωση που ο grocer τελειώσει να μην χρειάζεται να περιμένουμε τον επόμενο
        πελάτη και να τερματίσει κατευθείαν

            w -= 0.2 #επιλέξαμε εμείς τυχαία αυτή τη μείωση
        return

def thread_service(): #λαμβάνει τον επόμενο πελάτη από την ουρά
        while do_work: #όσο η do_work είναι true σημαίνει ότι η λειτουργία του grocer
δεν έχει τελειώσει
            if client_queue.not_empty: #αν υπάρχουν πελάτες στην ουρά
                c = client_queue.get() #λαμβάνουμε την επόμενο πελάτη από την ουρά

            c.start() #αρχίζει το νήμα του πελάτη

            c.join() #το νήμα του πελάτη γίνεται join
        return

def thread_client():
    b = random.choice(list(baskets.keys())) #ο πελάτης επιλέγει τυχαία ένα στοιχείο
από το καλάθι δηλ. ένα τυχαίο φρούτο

    a = random.randint(10, 30) #επιλέγεται τυχαίος αριθμός από 10 έως 30 που είναι
τα φρούτα που θα πάρει

    if baskets[b] < a: #ελέγχεται αν υπάρχει απόθεμα από το συγκεκριμένο φρούτο
        has_fruit.clear() #αν δεν υπάρχει απόθεμα κάνει clear το event. Με το clear αυτό
το wait στη συνέχεια περιμένει

    has_fruit.wait()#ο πελάτης περιμένει να κάνει set το event ο grocer

    with basket_sema:
        baskets[b] -= a #αφαίρεση ποσότητας αγοράς από το καλάθι

        print("Client: Bought {} from {}".format(a, b)) #τυπώνονται η ποσότητα
αγοράς και το υπόλοιπο από το κάθε φρούτο που απομένει
```

```
        print("Baskets: Contain ", baskets) #εκτυπώνει το καλάθι

        return

if __name__ == '__main__':
    print("Baskets: Contain ", baskets)
    if all(v == 100 for v in baskets.values()): #ελέγχονται όλα τα αποθέματα αν είναι ίσα με 100
        has_fruit.set() #σημαίνει ότι υπάρχουν φρούτα στο καλάθι ώστε να εξυπηρετηθούν οι πελάτες μέχρι το 1ο refill δηλ. ξεκινάμε με φρούτα

        grocer = threading.Thread(target=thread_grocer) #δημιουργία νήματος Grocer
        grocer.start()#εκκίνηση νήματος Grocer

        dispatcher = threading.Thread(target=thread_dispatcher) ) #δημιουργία νήματος dispatcher

        dispatcher.start()#εκκίνηση νήματος dispatcher

        service = threading.Thread(target=thread_service) #δημιουργία νήματος service
        service.start() #εκκίνηση νήματος service

        #με τις επόμενες 3 εντολές join συνενώνουμε όλα τα νήματα που δημιουργήσαμε ώστε να τερματίσουν
        service.join()

        dispatcher.join()

        grocer.join()

        sys.exit(0) #τερματισμός προγράμματος
```

Αποτελέσματα Εκτέλεσης

Baskets: Contain {'apples': 100, 'oranges': 100, 'pears': 100, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 20 from pears

Client: Next client in arrives in 10 seconds Basket contains
{'apples': 100, 'oranges': 100, 'pears': 80, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 17 from pears

Client: Next client in arrives in 17 seconds Basket contains
{'apples': 100, 'oranges': 100, 'pears': 63, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 10 from oranges

Client: Next client in arrives in 24 seconds Basket contains
{'apples': 100, 'oranges': 90, 'pears': 63, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 23 from apples

Client: Next client in arrives in 16 seconds Basket contains
{'apples': 77, 'oranges': 90, 'pears': 63, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 29 from pears

Client: Next client in arrives in 15 seconds Basket contains
{'apples': 77, 'oranges': 90, 'pears': 34, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 23 from pears

Client: Next client in arrives in 15 seconds Basket contains
{'apples': 77, 'oranges': 90, 'pears': 11, 'bananas': 100}
Dispatcher: There are 1 clients waiting
Client: Next client in arrives in 3 seconds
Dispatcher: There are 1 clients waiting
Client: Next client in arrives in 11 seconds
Grocer: Refilled baskets. 9 refills remaining
Baskets: Contain {'apples': 100, 'oranges': 100, 'pears': 100, 'bananas': 100}
Client: Bought 20 from pears
Baskets: Contain {'apples': 100, 'oranges': 100, 'pears': 80, 'bananas': 100}
Client: Bought 14 from apples
Baskets: Contain {'apples': 86, 'oranges': 100, 'pears': 80, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 21 from oranges

Client: Next client in arrives in 3 seconds Basket contains
{'apples': 86, 'oranges': 79, 'pears': 80, 'bananas': 100}

Dispatcher: There are 1 clients waiting Client: Bought 20 from oranges

Client: Next client in arrives in 17 seconds Basket contains
{'apples': 86, 'oranges': 59, 'pears': 80, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 18 from oranges

Client: Next client in arrives in 28 seconds Basket contains
{'apples': 86, 'oranges': 41, 'pears': 80, 'bananas': 100}
Dispatcher: There are 1 clients waiting Client: Bought 23 from bananas

6 Εργαστηριακή Άσκηση 2 με Γράφους

6.1 Διαδικασία Δημιουργίας Γράφου στην Python

Για να δημιουργήσουμε γράφο στην Python ακολουθούμε τα παρακάτω βήματα:

1. Σχεδιάζουμε το γράφο που θέλουμε να προσομοιώσουμε.

2. Αριθμούμε τους κόμβους του γράφου με τους ακεραίους {1, 2, …, N} όπου N το πλήθος κορυφών

3. Καταστρώνουμε ένα πίνακα με 3 στήλες όπως παρακάτω:

Κόμβος	Πλήθος Γειτόνων	Ακμές

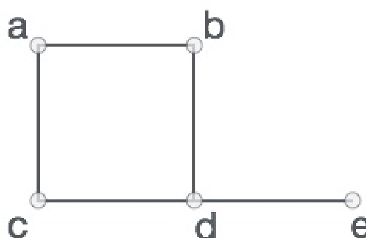
4. Γεμίζουμε τον πίνακα ως εξής.

a. Στη στήλη «**Κόμβος**» τοποθετούμε τους ακεραίους {1, 2, …, N}. Συνεπώς, κάθε γραμμή αντιστοιχεί σε έναν κόμβο του γράφου.

b. Στη στήλη «**Πλήθος Γειτόνων**» τοποθετούμε το πλήθος των γειτόνων κάθε κόμβου. Σημειώνεται πως ο κόμβος 'λ' είναι γείτονας του κόμβου 'κ', όταν υπάρχει ακμή από τον 'κ' στον 'λ'.

c. Στη στήλη «**Ακμές**» γράφουμε τους αριθμούς των κόμβων που γειτονεύουν με τον τρέχοντα κόμβο.

Για παράδειγμα για το γράφο του επόμενου σχήματος



ο πίνακας συμπληρώνεται ως εξής:

Κόμβος	Πλήθος Γειτόνων	Ακμές
a	2	b, c
b	2	a, d
c	2	a. d
d	3	c, d, e

5. Γράφουμε τον κώδικα που υλοποιεί τον παραπάνω γράφο.

```
# Δημιουργία του Γράφου
graph={"a" : ["b", "c"],"b" : ["a", "d"],"c" : ["a", "d"],"d" : ["e"],"e" :["d"]}

#Εκτύπωση του Γράφου
print(graph)
```

Αποτελέσματα Εκτέλεσης

{'a': ['b', 'c'], 'b': ['a', 'd'], 'c': ['a', 'd'], 'd': ['e'], 'e': ['d']}

6.2 Γενικοί Γράφοι στην Python

Θα δοκιμάσουμε να εκτελέσουμε κάποια προγράμματα σε Python προκειμένου να γίνει πλήρως κατανοητή η λειτουργία τους.

Βήμα 1: Δημιουργία τοπολογίας αστέρα

```
#Δημιουργία του Λεξικού με τα στοιχεία του Γράφου
graph={"a" : ["e"], "b" : ["e"], "c" : ["e"],"d" : ["e"],"e" : ["a", "b", "c", "d"],}

#Εκτύπωση Γράφου
print(graph)
```

Βήμα 3: Κατανοώντας ένα ολοκληρωμένο παράδειγμα

Σε αυτό το βήμα θα δημιουργήσουμε ένα πρόγραμμα σε Python, στο οποίο θα δημιουργείται και θα ελέγχεται ένας γράφος σε τοπολογία Αστέρα

```
#Πρόγραμμα που ελέγχει αν ένας γράφος αποτελεί τοπολογία αστέρα
def addEdge(adj, u, v): #Η συνάρτηση αυτή προσθέτει μια ακμή σε ένα η διευθυνόμενο γράφο
	adj[u].append(v)
	adj[v].append(u)

def printGraph(adj, V): #Η συνάρτηση αυτή εκτυπώνει το γράφο (συγκεκριμένα εκτυπώνει τη λίστα γειτνίασης του γράφου)
	for v in range(V):
		print("Adjacency list of vertex ",v,"\n head ")

	for x in adj[v]:
		print("-> ",x,end=" ")
```

```
def checkStarTopologyUtil(adj, V, E): #Συνάρτηση που επιστρέφει true αν ο γράφος που αναπαριστάνεται από την λίστα γειτνίασης αντιπροσωπεύει μια τοπολογία αστέρα
        if (E != (V - 1)): #Το πλήθος ακμών πρέπει να είναι ίσο με το πλήθος κορυφών -1
                return False

        #ένας μοναδικός κόμβος είναι τοπολογία διαύλου
        if (V == 1):
           return True

        vertexDegree = [0]*(V + 1)

        #Υπολογισμός Βαθμού κάθε κορυφής
        for i in range(V+1):
                for v in adj[i]:
                        vertexDegree[v] += 1

#Η μεταβλητή countCentralNodes αποθηκεύει το πλήθος τω κόμβων βαθμού V – 1 το οποίο πρέπει να είναι ίσο με 1 στην περίπτωση της τοπολογίας αστέρα
        countCentralNodes = 0
        centralNode = 0

        for i in range(1, V + 1):
                if (vertexDegree[i] == (V - 1)):
                      countCentralNodes += 1

        #Αποθήκευση δείκτη κεντρικού κόμβου
        centralNode = i

        #Θα υπάρχει μόνο ένας κεντρικός κόμβος στην τοπολογία αστέρα
        if (countCentralNodes != 1):
                return False

        for i in range(1, V + 1): # εκτός από τον κεντρικό κόμβο έλεγξε αν όλοι οι άλλοι κόμβοι έχουν βαθμό 1. Αν όχι επέστρεψε false
        if (i == centralNode):
           continue

        if (vertexDegree[i] != 1):
           return False
```

```
#αν και οι 3 προ-απαιτούμενες συνθήκες ικανοποιούνται επιστρέφεται true
        return True

def checkStarTopology(adj, V, E): #Συνάρτηση που ελέγχει αν ένας γράφος αντιπροσωπεύει τοπολογία star
        isStar = checkStarTopologyUtil(adj, V, E)
                if (isStar):
                   print("YES")
                else:
                   print("NO" )

# Graph 1
V, E = 5, 4
adj1=[[] for i in range(V + 1)]
addEdge(adj1, 1, 2)
addEdge(adj1, 1, 3)
addEdge(adj1, 1, 4)
addEdge(adj1, 1, 5)
checkStarTopology(adj1, V, E)

# Graph 2
V, E = 4, 4
adj2=[[] for i in range(V + 1)]
addEdge(adj2, 1, 2)
addEdge(adj2, 1, 3)
addEdge(adj2, 3, 4)
addEdge(adj2, 4, 2)
checkStarTopology(adj2, V, E)
```

7 Sockets στην Python

7.1 Περιγραφή Socket

Οι διεπαφές και οι API διεπαφές χρησιμοποιούνται για την αποστολή μηνυμάτων σε ένα δίκτυο. Παρέχουν μια μορφή επικοινωνίας μεταξύ διεργασιών (IPC). Το δίκτυο μπορεί να είναι ένα λογικό ή ένα τοπικό δίκτυο στον υπολογιστή ή ένα δίκτυο που είναι φυσικά συνδεδεμένο σε εξωτερικό δίκτυο, με τις δικές του συνδέσεις με άλλα δίκτυα. Το προφανές παράδειγμα είναι το Διαδίκτυο, στο οποίο συνδεόμαστε μέσω του ISP μας.

7.2 TCP και UDP Sockets

Θα δημιουργήσουμε ένα socket (socket object) χρησιμοποιώντας τη συνάρτηση socket.socket() και θα καθορίσουμε τον τύπο του ως socket.SOCK_STREAM. Όταν γίνει αυτό, το προεπιλεγμένο πρωτόκολλο που χρησιμοποιείται είναι (TCP) που έχει τα ακόλουθα χαρακτηριστικά:

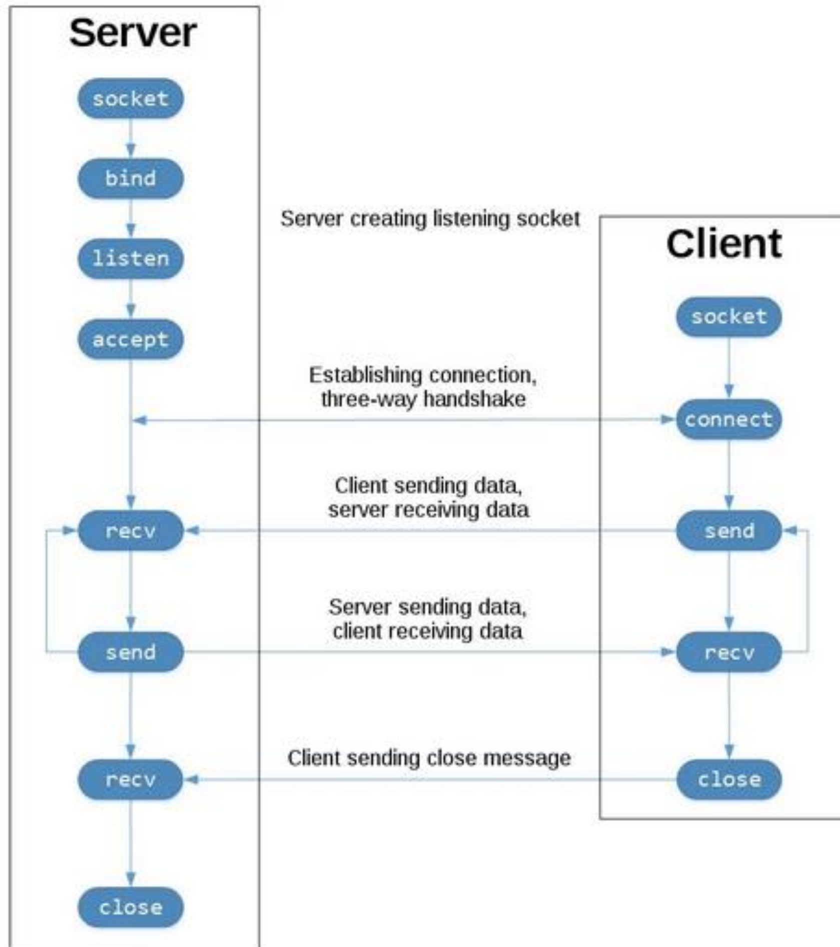
- Είναι αξιόπιστο δηλαδή ο παραλήπτης επιβεβαιώνει κάθε πακέτο το οποίο λαμβάνει. Συνεπώς για κάθε πακέτο που δεν επιβεβαιώνεται από τον παραλήπτη, ο αποστολέας το επαναμεταδίδει

- Τα δεδομένα λαμβάνονται από τον παραλήπτη με τη σειρά που στάλθηκαν από τον αποστολέα

Αντίθετα, τα socket του User Datagram Protocol (UDP) που έχουν δημιουργηθεί με τη συνάρτηση socket.SOCK_DGRAM δεν είναι αξιόπιστα και τα δεδομένα που διαβάζει ο παραλήπτης μπορεί να είναι με διαφορετική σειρά από αυτή που τα έστειλε ο αποστολέας. Τα δίκτυα είναι συστήματα παράδοσης στα οποία δεν υπάρχει εγγύηση ότι τα δεδομένα μας θα φτάσουν στον προορισμό του ή ότι θα λάβουμε ό, τι μας έχει σταλεί.

Οι συσκευές δικτύου (π.χ. δρομολογητές και switches) έχουν διαθέσιμο περιορισμένο εύρος ζώνης και τους δικούς τους περιορισμούς. Διαθέτουν CPU, μνήμη, διαύλους και buffer πακέτων όπως ακριβώς οι πελάτες και οι διακομιστές. Το TCP μας απαλλάσσει από την ανησυχία απώλειας πακέτων είτε για την άφιξη δεδομένων εκτός σειράς και πολλά άλλα που συμβαίνουν όταν επικοινωνούμε μέσω δικτύου.

Στο παρακάτω διάγραμμα παρατηρούμε την ακολουθία κλήσεων για API socket και τη ροή δεδομένων για TCP:



Εικόνα 1-Ροή στο TCP

Στην προηγούμενη εικόνα η αριστερή στήλη αντιπροσωπεύει το server, ενώ η δεξιά στήλη αντιπροσωπεύει τον client.

Ξεκινώντας από την επάνω αριστερή στήλη, παρατηρούμε ότι ο διακομιστής (server) πραγματοποιεί κλήσεις για να ρυθμίσει ένα socket "ακρόασης":

- socket()
- bind()
- listen()
- accept()

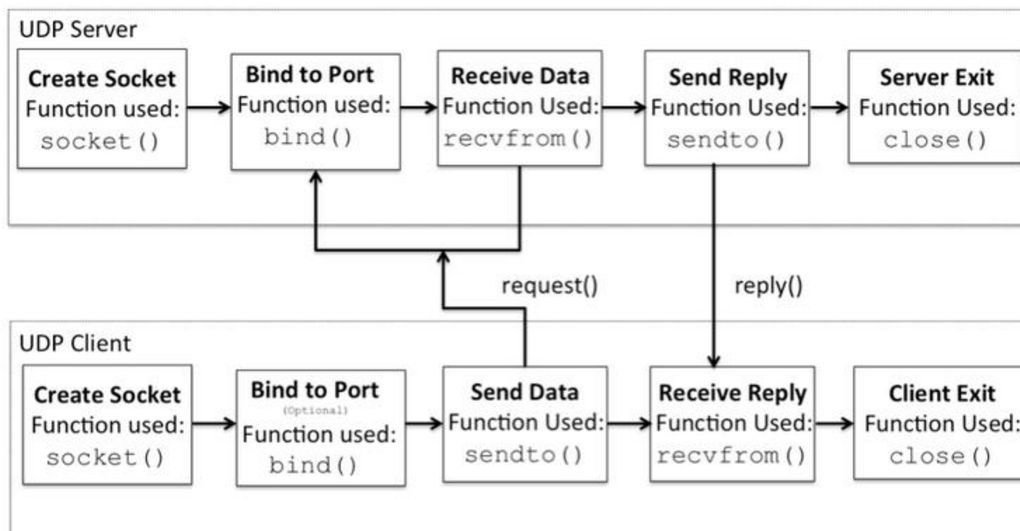
Ένα socket ακρόασης δέχεται συνδέσεις από πελάτες. Όταν ένας πελάτης συνδέεται, ο διακομιστής εκτελεί τη συνάρτηση accept() για να αποδεχτεί ή να ολοκληρώσει τη σύνδεση.

Ο πελάτης καλεί τη συνάρτηση connect() για να δημιουργήσει μια σύνδεση με το διακομιστή και να ξεκινήσει την τριμερή χειραψία (three-way handshake). Το βήμα χειραψίας είναι σημαντικό δεδομένου ότι διασφαλίζει ότι κάθε πλευρά της

σύνδεσης είναι προσβάσιμη στο δίκτυο, με άλλα λόγια ότι ο πελάτης μπορεί να επικοινωνήσει με το διακομιστή και αντίστροφα.

Στο μεσαίο τμήμα της προηγούμενης εικόνας βρίσκεται η ανταλλαγή δεδομένων μεταξύ του πελάτη και του διακομιστή χρησιμοποιώντας κλήσεις send() και recv().

Στο κάτω μέρος, ο πελάτης και ο διακομιστής κλείνουν τα sockets τους με τη συνάρτηση close().



Εικόνα 3-Επικοινωνία Server-Client στο UDP

7.3 Δημιουργία TCP Socket Διακομιστή (Server)

Ο επόμενος κώδικας python παρουσιάζει αναλυτικά τη δημιουργία ενός TCP socket από την πλευρά του διακομιστή (server).

#δημιουργία ενός TCP socket
serversocket = socket.***socket***(***socket.AF_INET, socket.SOCK_STREAM***)

#Λήψη ονόματος τοπικού Η/Υ
host = socket.***gethostname***()

port=9999 *#ορισμός port επικοινωνίας*

#σύνδεση του socket στο host και στο port 9999
serversocket.***bind***(host, port)

#το socket εισέρχεται σε κατάσταση ακρόασης (listening mode) και δέχεται μέχρι 5 αιτήσεις στην ουρά
serversocket.***listen***(5)

#ο server μπαίνει σε ένα άπειρο βρόγχο είτε μέχρι να τον διακόψουμε είτε μέχρι να εμφανιστεί σφάλμα
while True:

#Δημιουργία σύνδεσης με client.
clientsocket, addr = serversocket.***accept***()

#εκτύπωση ληφθείσας διεύθυνσης ως string
print("Got a connection from %s" % str(addr))

#Αποστολή μηνύματος στον client
msg = 'Thank you for connecting\n"

#αποστολή μηνύματος στο client φού κωδικοποιηθεί σε χαρακτήρες
clientsocket.***send***(msg.***encode***('ascii'))

#Κλείσιμο σύνδεσης με client
c.***close***()

Επεξηγήσεις Κώδικα

α) χρησιμοποιήσαμε τη συνάρτηση socket.***gethostname***() ώστε το socket να είναι παντού ορατό. Εάν είχαμε χρησιμοποιήσει τη συνάρτηση s.***bind***(('localhost', 80)) ή τη συνάρτηση s.***bind***(('127.0.0.1', 80)) θα είχαμε ακόμα ένα socket διακομιστή (server socket), αλλά θα ήταν ορατό εντός της ίδιας μηχανής. Το s.***bind*** ((", 80)) καθορίζει ότι το socket θα είναι προσβάσιμο από οπουδήποτε.

β) οι θύρες (Port) χαμηλού αριθμού προορίζονται συνήθως για "γνωστές" υπηρεσίες (HTTP, SNMP κ.λπ.). Συνεπώς θα πρέπει να χρησιμοποιήσουμε υψηλό αριθμό (4 ψηφία) για το port.

γ) Η εντολή serversocket.***listen***(5) λέει στο socket ότι θέλουμε να περιμένουν στην ουρά έως και 5 αιτήσεις σύνδεσης.

Υπάρχουν 2 γενικοί τρόποι με τους οποίους θα μπορούσε να λειτουργήσει ο βρόχος του server:

α) αποστολή ενός νήματος για χειρισμό του socket πελατών

β) δημιουργία μιας νέας διαδικασίας για τον χειρισμό της υποδοχής πελατών.

Αυτό που πρέπει να τονίσουμε είναι ότι το socket «διακομιστή» δεν αποστέλλει δεδομένα, δεν λαμβάνει δεδομένα. Παράγει απλώς sockets «πελάτη». Κάθε socket πελάτη δημιουργείται σε απάντηση σε κάποια άλλο socket "πελάτη" που κάνει connect() με τον κεντρικό υπολογιστή και τη θύρα στην οποία είμαστε συνδεδεμένοι. Μόλις δημιουργήσουμε αυτό το πρόγραμμα-πελάτη, επιστρέφουμε στην ακρόαση για περισσότερες συνδέσεις. Οι δύο «πελάτες» είναι ελεύθεροι να συνομιλήσουν χρησιμοποιώντας κάποια δυναμικά εκχωρημένη θύρα που θα ανακυκλωθεί όταν τελειώσει η συνομιλία.

7.4 Δημιουργία TCP Socket Πελάτη (Client)

Ο επόμενος κώδικας python παρουσιάζει αναλυτικά τη δημιουργία ενός TCP socket από την πλευρά του πελάτη (client).

```
#Εισαγωγή socket module
import socket

#Δημιουργία TCP socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Λήψη ονόματος τοπικού Η/Υ
host = socket.gethostname()

#Ορισμός port σύνδεσης
port = 9999

#Σύνδεση στο server στον τοπικό υπολογιστή στο port 9999
s.connect(host, port)

#Λήψη δεδομένων από server
msg=s.recv(1024)

#Κλείσιμο σύνδεσης
s.close()

#εκτύπωση ληφθέντος μηνύματος αφού γίνει πρώτα αποκωδικοποίηση σε ASCII
print(msg.decode('ascii'))
```

7.5 UDP Server

Ο επόμενος κώδικας python παρουσιάζει αναλυτικά τη δημιουργία ενός UDP socket από την πλευρά του διακομιστή (server).

```
import socket
localIP="127.0.0.1"
localPort=20001
bufferSize=1024

msgFromServer="Hello UDP Client"
bytesToSend= str.encode(msgFromServer)

#Δημιουργία UDP socket από το server. Η σταθερά SOCK_DGRAM δημιουργεί UDP socket
UDPServerSocket=socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM)

#Σύνδεση στην IP 127.0.09.1 και στο port 20001
UDPServerSocket.bind((localIP, localPort))

print("UDP server up and listening")

#O server ακούει εισερχόμενες κλήσεις δηλ. αντιλαμβάνεται εισερχόμενα πακέτα
while(True): #o server μπαίνει σε ένα άπειρο βρόγχο για να δέχεται συνεχώς αιτήσεις
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]
    clientMsg="Message from Client:{}".format(message) #μορφοποίηση ληφθέντος μηνύματος IP από client
    clientIP= "Client IP Address:{}".format(address) #μορφοποίηση ληφθείσας IP από client

    print(clientMsg) #εκτύπωση ληφθέντος μηνύματος από client
    print(clientIP) # εκτύπωση ληφθείσας IP από client

#Αποστολή απάντησης στον client
UDPServerSocket.sendto(bytesToSend, address)
```

<u>Αποτελέσματα Εκτέλεσης</u>

UDP server up and listening

Message from Client: b"Hello UDP Server"

Client IP Address:("127.0.0.1", 51696)

7.6 UDP Client

Ο επόμενος κώδικας python παρουσιάζει αναλυτικά τη δημιουργία ενός UDP socket από την πλευρά του πελάτη (client).

import socket
msgFromClient="Hello UDP Server"
bytesToSend= str.***encode***(msgFromClient)
serverAddressPort= ("127.0.0.1", 20001)
bufferSiz= 1024

#Δημιουργία UDP socket από τον client. Η σταθερά SOCK_DGRAM δημιουργεί UDP socket
UDPClientSocket=socket.socket(family=***socket.AF_INET***, type=socket.***SOCK_DGRAM***)

#Αποστολή στο server χρησιμοποιώντας το UDP socket που δημιουργήθηκε
UDPClientSocket.***sendto***(bytesToSend, serverAddressPort)

msgFromServer = UDPClientSocket.***recvfrom***(bufferSize)
msg = "Message from Server {}".***format***(msgFromServer[0]) *#μορφοποίηση ληφθέντος μηνύματος*

print(msg) #εκτύπωση *ληφθέντος μηνύματος*

<u>Αποτελέσματα Εκτέλεσης</u>

Message from Server b"Hello UDP Client

7.7 Παράδειγμα Echo Client and Server

Τώρα που έχουμε αποκτήσει μια γενική εικόνα του API socket και του τρόπου επικοινωνίας πελάτη και διακομιστή θα δημιουργήσουμε μια απλή εφαρμογή με πελάτη και διακομιστή στην οποία ο διακομιστής θα επαναλάβει απλώς ό, τι λαμβάνει από τον πελάτη.

7.7.1 Υλοποίηση Echo Server

```
import socket
HOST = '127.0.0.1'  #αυτή είναι η διεύθυνση του server (localhost)
PORT = 65432 #Το Port που θα "ακούει" ο server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
        data = conn.recv(1024)
        if not data:
            break
        conn.sendall(data)
```

Επεξήγηση Κώδικα Echo Server

import socket
HOST = '127.0.0.1' *#Η τυπική loopback διεύθυνση (localhost)*
PORT = 65432 *#Το Port επικοινωνίας*

with socket.***socket***(socket.***AF_INET***, socket.***SOCK_STREAM***) ***as*** s:*#Με τη συνάρτηση socket.socket() δημιουργείται ένα αντικείμενο socket που υποστηρίζει τον τύπο context manager άρα μπορεί να χρησιμοποιηθεί σε μια εντολή with . Για ένα τέτοιο socket δεν χρειάζεται στο τέλος να καλέσουμε τη συνάρτηση s.close(). Τα ορίσματα στη συνάρτηση socket() αντιπροσωπεύουν την οικογένεια διευθύνσεων (Internet address family) και τον τύπο του socket. Το AF_INET είναι η Internet address family του IPv4 και το SOCK_STREAM είναι ο τύπος socket για το πρωτόκολλο* TCP *που θα χρησιμοποιηθεί για τη μεταφορά μηνυμάτων στο δίκτυο*

s.***bind***((HOST, PORT)) *#Η συνάρτηση bind() χρησιμοποιείται για να συσχετίσει το socket με ένα συγκεκριμένο network interface και αριθμό θύρας (port number). Οι*

τιμές που δίνονται ως ορίσματα στην bind() εξαρτώνται από την οικογένεια διευθύνσεων (address family) του socket. Στο παράδειγμα αυτό χρησιμοποιούμε την socket.AF_INET (IPv4). Άρα η bind περιμένει μια δυάδα της μορφής (host, port). Εδώ η παράμετρος Host αντιπροσωπεύει τη loopback διεύθυνση 127.0.0.1 και η παράμετρος PORT τον αριθμό θύρας 65432. Η διεύθυνση 127.0.0.1 σημαίνει ότι διεργασίες μόνο σε αυτό τον host θα μπορούν να συνδεθούν με το server. Αν μεταβιβάσουμε μια κενή συμβολοσειρά ο server θα αποδέχεται συνδέσεις από όλες τις διαθέσιμες διεπαφές IPv4

s.***listen***() #*Η εκτέλεση της συνάρτησης αυτής προετοιμάζει το socket για να δέχεται συνδέσεις. Η συνάρτηση αυτή πρέπει να καλείται πριν την κλήση της accept() στο server socket. Η listen() αποδέχεται ένα μέγεθος ουράς μέσω της παραμέτρου backlog. Αυτό το μέγεθος υποδηλώνει το μέγιστο αριθμό συνδέσεων που μπορούν να στέλνουν σε αυτό το socket.*

conn, addr = s.***accept***() #*Η συνάρτηση accept() μπλοκάρει και περιμένει εισερχόμενη σύνδεση. Όταν ένας πελάτης συνδέεται, επιστρέφει ένα νέο αντικείμενο socket που αντιπροσωπεύει τη σύνδεση και μια πλειάδα που κρατά τη διεύθυνση του πελάτη. Η πλειάδα είναι της μορφής (κεντρικός υπολογιστής-host, θύρα-port) για συνδέσεις IPv4. Ένα σημείο που πρέπει να διευκρινιστεί είναι ότι έχουμε τώρα ένα νέο αντικείμενο socket από τη συνάρτηση accept(). Αυτό είναι σημαντικό δεδομένου ότι αυτό είναι το socket που θα χρησιμοποιηθεί για την επικοινωνία με τον πελάτη. Είναι διαφορετικό από το socket ακρόασης που χρησιμοποιεί ο διακομιστής για να δέχεται νέες συνδέσεις*

with conn: #*η εντολή αυτή θα κλείσει αυτόματα το socket στο τέλος*

print('Connected by', addr)

while True: #*ο server μπαίνει σε άπειρο βρόγχο*

data = conn.recv(1024) #*με τη συνάρτηση recv() ο server διαβάζει δεδομένα από ένα πελάτη*

if not data:

break

conn.***sendall***(data) #*με τη συνάρτηση sendall() ο server ξαναστέλνει ότι πήρε ο server πίσω στον πελάτη*

7.7.2 Υλοποίηση Echo Client

```
import socket
HOST = '127.0.0.1' #H IP διεύθυνση του server (server's hostname)
PORT = 65432 #To port που χρησιμοποιείται από το server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

<u>Επεξήγηση Κώδικα Echo Client</u>

Σε σύγκριση με τον διακομιστή, ο πελάτης είναι πολύ απλός. Δημιουργεί ένα αντικείμενο socket, συνδέεται με τον διακομιστή και εκτελεί τη συνάρτηση s.sendall() για να στείλει το μήνυμά του. Τέλος, καλεί τη συνάρτηση s.recv() για να λάβει την απάντηση του διακομιστή και στη συνέχεια να την εκτυπώσει.

7.7.3 Εκτέλεση Echo Server και Client

Αρχικά εκτελούμε το server, τον κώδικα του οποίου έχουμε αποθηκεύσει σε ένα αρχείο με όνομα echo-server.py, με την εντολή:

$ ./echo-server.py

Το τερματικό μας φαίνεται αρχικά να μπλοκάρει επειδή ο server περιμένει να λάβει κλήση. Συγκεκριμένα εκτελεί τη συνάρτηση conn, addr = s.accept() και περιμένει να λάβει κλήση από client. Μετά ανοίγουμε ένα δεύτερο παράθυρο και τρέχουμε τον client με την εντολή:

$ ./echo-client.py
Received b'Hello, world'

Στο παράθυρο του server θα δούμε το ακόλουθο μήνυμα:

$ ./echo-server.py
Connected by ('127.0.0.1', 64623)

Στο μήνυμα αυτό ο server τυπώνει την πλειάδα addr που επιστρέφεται από τη συνάρτηση s.accept(). Αυτή η πλειάδα περιλαμβάνει την IP του client και τον αριθμό θύρας του TCP. Το port 64623 μπορεί να είναι διαφορετικό στον υπολογιστή μας

8 Εργαστηριακή Άσκηση 3 με Κατανεμημένα Ρολόγια

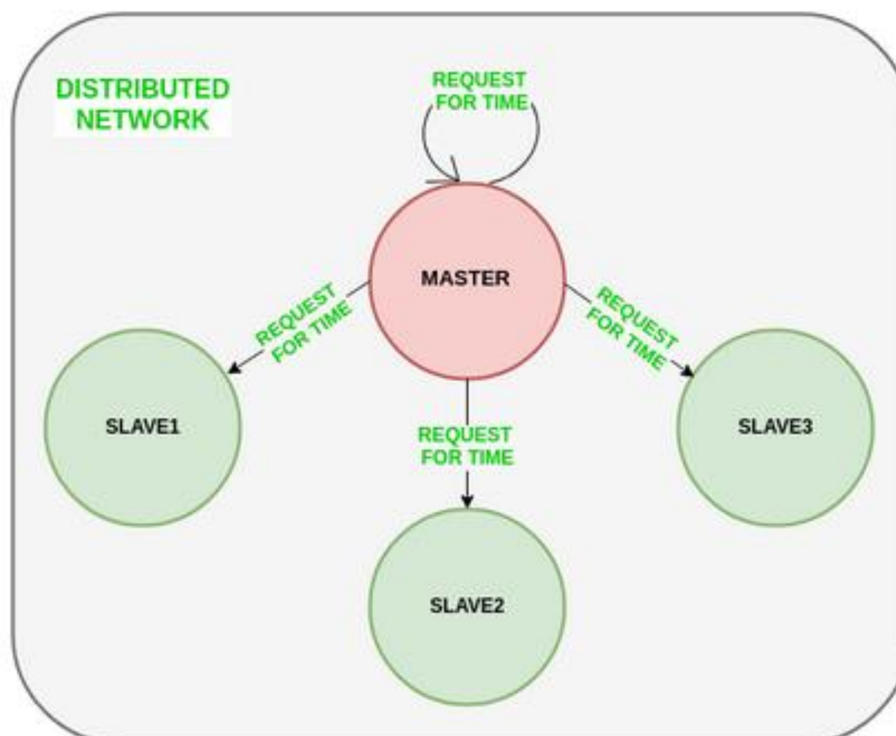
Ο Αλγόριθμος του Berkeley είναι μια τεχνική συγχρονισμού ρολογιού που χρησιμοποιείται σε κατανεμημένα συστήματα. Ο αλγόριθμος υποθέτει ότι κάθε κόμβος στο δίκτυο είτε δεν διαθέτει ακριβή πηγή χρόνου ή δεν διαθέτει διακομιστή UTC. Πριν περιγράψουμε τους αλγορίθμους Berkeley και Cristian για Κατανεμημένα Ρολόγια θα περιγράψουμε τις βασικές αρχές των sockets στην Python.

8.1 Αλγόριθμος Berkeley

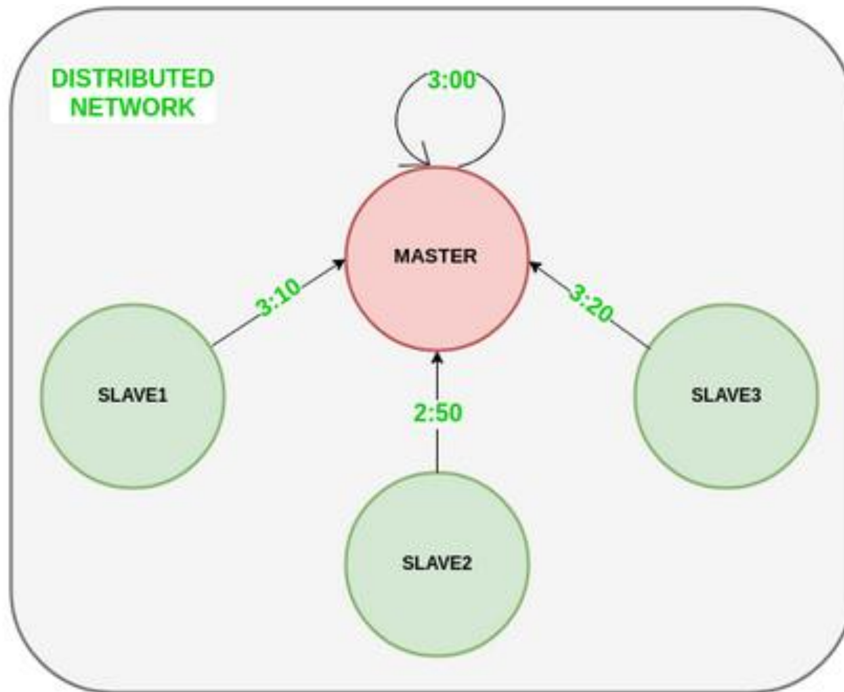
1) Ένας μεμονωμένος κόμβος επιλέγεται ως κύριος κόμβος (αρχηγός) από ένα σύνολο κόμβων στο δίκτυο. Αυτός ο κόμβος είναι ο κύριος κόμβος στο δίκτυο και λειτουργεί ως αρχηγός όπως αναφέραμε ενώ οι υπόλοιποι κόμβοι λειτουργούν ως δευτερεύοντες (slave nodes). Ο αρχηγός επιλέγεται χρησιμοποιώντας ένα αλγόριθμο εκλογικής διαδικασίας εκλογής αρχηγού

2) Ο αρχηγός κάνει ping περιοδικά στους υπόλοιπους κόμβους και λαμβάνει το χρόνο ρολογιού από αυτούς χρησιμοποιώντας τον αλγόριθμο Cristian

Το επόμενο διάγραμμα δείχνει τον τρόπο με τον οποίο ο αρχηγός (master) στέλνει αίτημα στους δευτερεύοντες κόμβους.



Το παρακάτω διάγραμμα δείχνει τον τρόπο με τον οποίο οι δευτερεύοντες κόμβοι (master slaves) στέλνουν πίσω το χρόνο από το ρολόι του δικού τους συστήματος



3) Ο κύριος κόμβος υπολογίζει τη μέση διαφορά χρόνου μεταξύ όλων των ωρών ρολογιού που λαμβάνονται και του χρόνου ρολογιού που δίνεται από το ρολόι συστήματος του κύριου κόμβου (master). Αυτή η μέση χρονική διαφορά προστίθεται στην τρέχουσα ώρα του ρολογιού του κύριου κόμβου και μεταδίδεται μέσω του δικτύου.

8.1.1 Ψευδοκώδικας Τελευταίου Βήματος Αλγορίθμου Berkeley

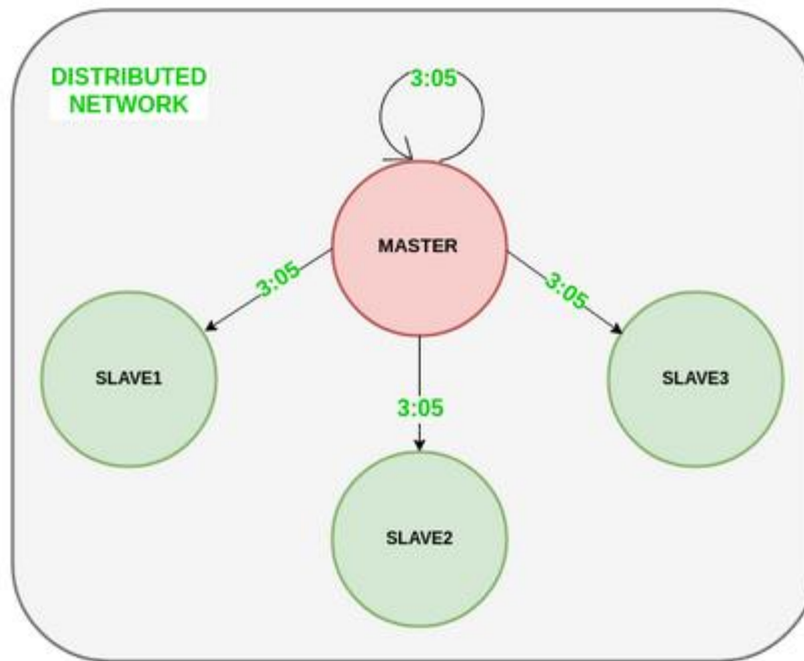
#Λήψη Χρόνου από όλους τους δευτερεύοντες κόμβους (slave nodes)
repeat for all slave nodes
time_at_slave_node = receive_time_at_slave()

#Υπολογισμός χρονικής διαφοράς
time_difference = time_at_master_node - time_at_slave_node

#Υπολογισμός Μέσης Χρονικής Διαφοράς
average_time_difference = sum(all_time_differences)/number_of_slaves

synchronized_time = current_master_time + average_time_difference
#μετάδοση συγχρονισμένης ώρας σε όλο το δίκτυο
broadcast_time_to_all_slaves(synchronized_time)

Το επόμενο διάγραμμα παρουσιάζει το τελευταίο βήμα του αλγορίθμου του Berkeley



Βελτιώσεις του αλγορίθμου Berkeley σε σχέση με τον αλγόριθμο του Cristian

- Βελτίωση της ακρίβειας του αλγόριθμου του Cristian
- Παράβλεψη σημαντικών ακραίων τιμών στον υπολογισμό της μέσης διαφοράς χρόνου
- Σε περίπτωση αποτυχίας του κύριου κόμβου, ένας δευτερεύων κόμβος πρέπει να είναι έτοιμος/προεπιλεγμένος για να τον αντικαταστήσει έτσι ώστε να μειωθεί ο χρόνος καθυστέρησης που προκαλείται λόγω μη διαθεσιμότητας του κύριου κόμβου
- Ο κύριος κόμβος αντί να στέλνει το συγχρονισμένο χρόνο, μεταδίδει τη σχετική αντίστροφη χρονική διαφορά, η οποία οδηγεί σε μείωση της καθυστέρησης που και αυτό γίνεται κατά τη διάρκεια υπολογισμού του χρόνου στον δευτερεύοντα κόμβο

8.1.2 Master Clock server

Ο επόμενος κώδικας αφορά ένα python script που υλοποιεί ένα master clock server:

```
from functools import reduce
from dateutil import parser
import threading
import datetime
import socket
import time

#ορίζουμε ένα λεξικό για την αποθήκευση της διεύθυνσης του client και των δεδομένων ρολογιού
client_data = { } #δήλωση λεξικού

#εμφωλευμένη συνάρτηση που εκτελείται από κάθε νήμα και λαμβάνει το χρόνο ρολογιού από ένα συνδεδεμένο clint
def startRecieveingClockTime(connector, address):
	while True:

	# Λήψη Ώρας ρολογιού
	clock_time_string = connector.recv(1024).decode() #το socket θα λάβει δεδομένα και θα τα τοποθετήσει σε ένα buffer μεγέθους 1024 bytes

	clock_time = parser.parse(clock_time_string) #μετατροπή τρέχουσας ώρας από string σε αντικείμενο ώρας

	clock_time_diff = datetime.datetime.now()-clock_time #υπολογισμός χρονικής διαφοράς μεταξύ τρέχουσας και ληφθείσας ημερομηνίας

	#δημιουργία λεξικού με τα στοιχεία κάθε client
	client_data[address] = {
		"clock_time": clock_time,
		"time_difference": clock_time_diff,
		"connector": connector
		}

	print("Client Data updated with: "+ str(address), end = "\n\n")
	time.sleep(5)
```

#'Συνάρτηση που εκτελείται από το master thread για το άνοιγμα portal για την αποδοχή πελατών στο port αυτό
def startConnecting(master_server):
 # Λήψη ώρας ρολογιού από πελάτες (slaves)

while True: *#εκτέλεση επανάληψης για κάθε πελάτη*
#Λήψη Ώρας από ένα client
master_slave_connector, addr = master_server.***accept***() *#η συνάρτηση accept αποδέχεται μια σύνδεση. Το socket πρέπει να συνδεθεί σε μια διεύθυνση και εκεί να δέχεται συνδέσεις (connections). Το επιστρεφόμενο αποτέλεσμα της accept είναι ένα ζεύγος (conn,address) όπου conn είναι ένα νέο αντικείμενο τύπου socket που χρησιμοποιείται για τη λήψη και αποστολή δεδομένων σε αυτή τη σύνδεση (connection) και address είναι η διεύθυνση που συνδέεται με το socket στο άλλο άκρο της σύνδεσης*

slave_address = str(addr[0]) + ":" + str(addr[1]) *#συνενώνουμε το ζεύγος conn, address με ένα χαρακτήρα : ανάμεσα τους*

print(slave_address + " got connected successfully")

current_thread = threading.***Thread***(target = startRecieveingClockTime, args = (master_slave_connector, slave_address,))

current_thread.start() *#έναρξη master thread*

#Συνάρτηση για τη λήψη μέσης διαφοράς ώρας (average clock difference)
def getAverageClockDiff():
current_client_data = client_data.copy() *#αντιγραφή του λεξικού client_data σε μια τοπική μεταβλητή current_client_data*

time_difference_list=list(client['time_difference'] ***for*** client_addr, client ***in*** client_data.items()) *#δημιουργία λίστας με όλες τις χρονικές διαφορές*

sum_of_clock_difference = sum(time_difference_list, datetime.timedelta(0,0)) *#άθροισμα χρονικών διαφορών*

average_clock_difference = sum_of_clock_difference/len(client_data) *#υπολογισμός μέσης διαφοράς ώρας*

return average_clock_difference *#επιστροφή μέσης διαφοράς ώρας*

```
#Συνάρτηση Συγχρονισμού του master thread για την παραγωγή κύκλων συγχρονισμού
στο δίκτυο
def synchronizeAllClocks():
        while True:
                print("New synchronization cycle started.")

                print("Number of clients to be synchronized: " +str(len(client_data)))
        #υπολογίζεται το πλήθος client  και μετατρέπεται σε string
                if len(client_data) > 0:
                        average_clock_difference = getAverageClockDiff()

                        for client_addr, client in client_data.items(): #για κάθε client
                εκτελείται επανάληψη. Στο πεδίο client καταχωρείται η διεύθυνση του
                                try:
                                    synchronized_time = datetime.datetime.now() + aver-
                                  age_clock_difference #στην τρέχουσα ώρα αθροίζονται όλες οι
                                  χρονικές διαφορές

                                          client['connector'].send(str(synchronized_time).en-
                                  code())#στέλνεται η νέα ώρα σε όλους τους client

                                  except Exception as e: #αν συμβεί σφάλμα
                                          print("Something went wrong while sending synchro-
                                  nized time through " + str(client_addr))

                else:
                        print("No connected clients. Synchronization not applicable.")
                print("\n\n")

                time.sleep(5) #θέτουμε χρονοκαθυστέρηση 5 msec

#Συνάρτηση Αρχικοποίησης του Clock Server (Master Node)
def initiateClockServer(port = 8080):
        master_server = socket.socket()
        master_server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

        print("Socket at Clock Server-Master Node created successfully\n")
        master_server.bind(('', port))

        #O clock server δύναται να "ακούει" αιτήματα
        master_server.listen(10)
```

```
        print("Clock server started...\n")

        #δημιουργία συνδέσεων
        print("Starting to make connections...\n")
        master_thread = threading.Thread(target = startConnecting, args = (master_server, ))
        master_thread.start()

        #Έναρξη Συγχρονισμού
        print("Starting synchronization parallely...\n")
        sync_thread=threading.Thread(target=synchronizeAllClocks, args = ())
        #δημιουργείται ένα νήμα για κάθε client
        sync_thread.start() #εκκίνηση νήματος

#Main
if __name__ == '__main__':

# Έναρξη Clock Server
initiateClockServer(port = 8080) #Καλείται η συνάρτηση initiateClockServer με όρισμα port=8080
```

Αποτελέσματα Εκτέλεσης Κώδικα

```
New synchronization cycle started.
Number of clients to be synchronized: 3
Client Data updated with 127.0.0.1:57284
Client Data updated with: 127.0.0.1:57274
Client Data updated with: 127.0.0.1:57272
```

8.1.3 Clock Client

Ο επόμενος κώδικας αφορά ένα python script που υλοποιεί ένα clock client:

```
from timeit import default_timer as timer
from dateutil import parser
import threading
import datetime
import socket
import time

#Συνάρτηση νημάτων που στέλνει χρόνο από την πλευρά του client
def startSendingTime(slave_client):
        while True:
        #μεταβίβαση στο server του χρόνου ρολογιού κάθε client
        slave_client.send(str(datetime.datetime.now()).encode())

        print("Recent time sent successfully", end = "\n\n")
        time.sleep(5)

#Συνάρτηση Νήματος Πελάτη που χρησιμοποιείται για να λάβει συγχρονισμένο χρόνο
def startReceivingTime(slave_client):
        while True:
        #Λήψη δεδομένων από server
        Synchronized_time = parser.parse(slave_client.recv(1024).decode())#λήψη και
μετατροπή τρέχουσας ώρας από string σε αντικείμενο ώρας από socket που θα λάβει
δεδομένα και θα τα τοποθετήσει σε ένα buffer μεγέθους 1024 bytes

        print("Synchronized time at the client is: "+str(Synchronized_time), end ="\n\n")

#Συνάρτηση συγχρονισμού του χρόνου επεξεργασίας των client
def initiateSlaveClient(port = 8080):
        slave_client = socket.socket() #δημιουργία socket από client

        #σύνδεση στον clock server στον τοπικό υπολογιστή στη διεύθυνση 127.0.0.1
        slave_client.connect(('127.0.0.1', port))

        #έναρξη αποστολής ώρας στο server
        print("Starting sending time to server\n")
        send_time_thread = threading.Thread(target = startSendingTime, args = (slave_client, ))
```

```
        send_time_thread.start() #αρχή νήματος client

        # έναρξη λήψης συγχρονισμένης ώρας από server
        print("Starting receiving synchronized time from server\n")
        receive_time_thread=threading.Thread(target=startReceivingTime,args=(slave_client,))
        receive_time_thread.start()

# Main
if __name__ == '__main__':

# αρχικοποίηση Slave/Client
initiateSlaveClient(port = 8080)
```

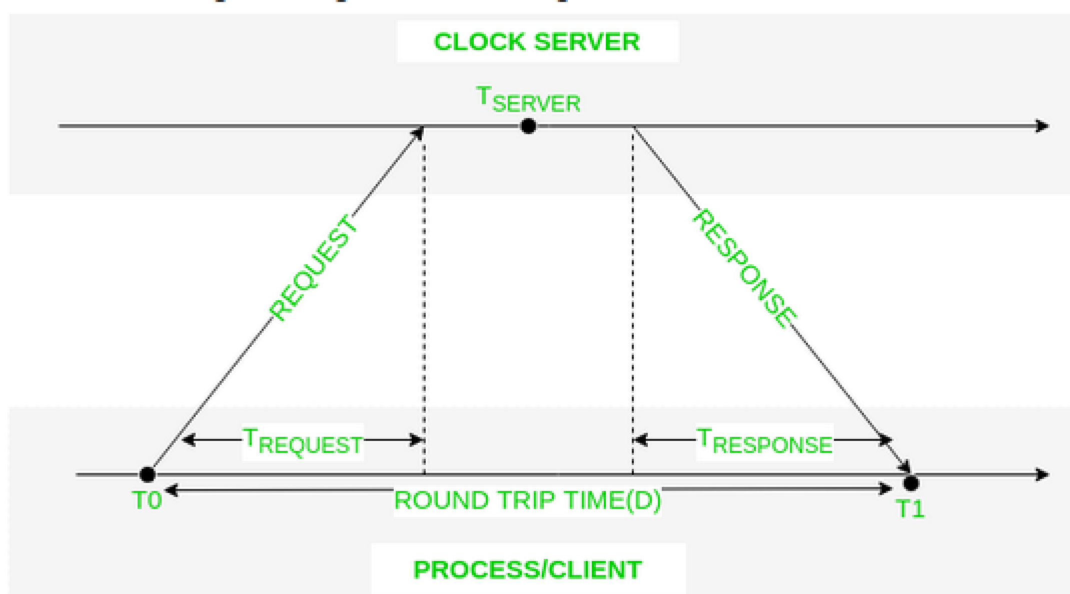
Αποτελέσματα Εκτέλεσης

Recent time sent successfully
Synchronized time at the client is: 2018-11-23 18:49:31.166449

8.2 Αλγόριθμος Cristian

Ο αλγόριθμος Cristian είναι ένας αλγόριθμος συγχρονισμού ρολογιού που χρησιμοποιείται για συγχρονισμό του χρόνου του διακομιστή ώρας με τους χρόνους των πελατών. Αυτός ο αλγόριθμος λειτουργεί καλά με δίκτυα χαμηλού λανθάνοντος χρόνου, όπου ο χρόνος αποστολής μετ 'επιστροφής (Round trip) είναι μικρός σε σύγκριση με την ακρίβεια, ενώ τα κατανεμημένα συστήματα/εφαρμογές επιρρεπείς δεν συμβαδίζουν με αυτόν τον αλγόριθμο. Εδώ το round-trip time αναφέρεται στη διάρκεια του χρόνου μεταξύ της έναρξης ενός αιτήματος και της λήψης της αντίστοιχης απάντησης.

Στην επόμενη Εικόνα παρατίθεται η λειτουργία του αλγορίθμου του Cristian:



8.2.1 Περιγραφή Αλγορίθμου Cristian

1) Η διαδικασία στον πελάτη στέλνει το αίτημα για ανάκτηση ώρας ρολογιού (ώρα διακομιστή) στο διακομιστή ρολογιών τη χρονική στιγμή T_0

2) Ο διακομιστής ρολογιού "ακούει" το αίτημα που υπέβαλε η διαδικασία του πελάτη και επιστρέφει την απάντηση με τη μορφή χρόνου του διακομιστή ρολογιού

3) Η διαδικασία πελάτη λαμβάνει την απάντηση από τον διακομιστή ρολογιών τη χρονική στιγμή T_1 και υπολογίζει το συγχρονισμένο χρόνο ρολογιού πελάτη χρησιμοποιώντας τον παρακάτω τύπο:

$$T_{CLIENT} = T_{SERVER} + (T_1 - T_0)/2$$

όπου:

- Ο χρόνος T_{CLIENT} αναφέρεται στο συγχρονισμένο χρόνο ρολογιού

- Ο χρόνος T_{SERVER} αναφέρεται στην ώρα ρολογιού που επιστρέφει ο διακομιστής
- Ο χρόνος T_0 αναφέρεται στο χρόνο κατά τον οποίο στάλθηκε το αίτημα από τη διαδικασία του πελάτη
- Ο χρόνος T_1 αναφέρεται στο χρόνο κατά τον οποίο ελήφθη η απάντηση από τον πελάτη

8.2.2 Λειτουργία/Αξιοπιστία του προηγούμενου τύπου

Η διαφορά T_1-T_0 αναφέρεται στο συνδυασμένο χρόνο που χρειάζεται το δίκτυο και ο διακομιστής για τη μεταφορά του αιτήματος στο διακομιστή, την επεξεργασία του αιτήματος και την επιστροφή της απόκρισης στη διαδικασία του πελάτη, υποθέτοντας ότι η καθυστέρηση δικτύου T_0 και T_1 είναι περίπου ίση.

Ο χρόνος από την πλευρά του πελάτη διαφέρει από τον πραγματικό χρόνο το πολύ κατά (T_1-T_0)/2 δευτερόλεπτα. Χρησιμοποιώντας την παραπάνω δήλωση μπορούμε να συμπεράνουμε ότι το σφάλμα συγχρονισμού μπορεί να είναι το πολύ (T_1-T_0)/2 δευτερόλεπτα.

Ως εκ τούτου,

$$error \, \epsilon \, [-(T_1 - T_0)/2, \, (T_1 - T_0)/2]$$

Οι παρακάτω κώδικες Python δείχνουν τη λειτουργία του αλγορίθμου Cristian:

8.2.3 Master Clock server

Ο επόμενος κώδικας αφορά ένα python script που υλοποιεί ένα clock server σε ένα τοπικό μηχάνημα (localhost):

```
import socket
import datetime

#Συνάρτηση Αρχικοποίησης ενός clock server
def initiateClockServer():
	s = socket.socket() #δημιουργία socket
	print("Socket successfully created")

	#Ορισμός Server port
	port = 8000

	s.bind(('', port)) #σύνδεση στο port

	#Έναρξη Ακρόασης Αιτήσεων
	s.listen(5)
	print("Socket is listening...")

	#ο Clock Server εκτελείται συνεχώς
	while True:

	#εγκαθίδρυση σύνδεσης με client
	connection, address = s.accept()
	print('Server connected to', address)

	#Ενημέρωση client με το χρόνου του server
	connection.send(str(datetime.datetime.now()).encode()) #η τρέχουσα ώρα στέλνεται σε κάθε client

	#Κλείσιμο της σύνδεσης του client
	connection.close()

#Main
if __name__ == '__main__':
	#Κλήση συνάρτησης server Clock Server
	initiateClockServer()
```

Αποτελέσματα Εκτέλεσης

Socket successfully created
Socket is listening...

8.2.4 Clock Client

Ο επόμενος κώδικας αφορά ένα python script που υλοποιεί ένα clock client σε ένα τοπικό μηχάνημα (localhost):

```
import socket
import datetime
from dateutil import parser
from timeit import default_timer as timer

#Συνάρτηση συγχρονισμού των χρόνων επεξεργασίας των client
def synchronizeTime():
	s = socket.socket()

	# Server port
	port = 8000

	#σύνδεση στο clock server στον τοπικό Η/Υ
	s.connect(('127.0.0.1', port))

	request_time = timer()

	#Λήψη δεδομένων από το server
	server_time = parser.parse(s.recv(1024).decode())
	response_time = timer()
	actual_time = datetime.datetime.now()
	print("Time returned by server: " + str(server_time))

	process_delay_latency = response_time - request_time
	print("Process Delay latency: "+ str(process_delay_latency)+ " seconds")

	print("Actual clock time at client side: "+ str(actual_time))

	#Συγχρονισμός χρόνου ρολογιών (client clock time)
	client_time=server_time+datetime.timedelta(seconds=(process_delay_latency)/2)

	print("Synchronized process client time:+ str(client_time))

	#υπολογισμός σφάλματος συγχρονισμού
	error = actual_time - client_time
	print("Synchronization error : + str(error.total_seconds()) + " seconds")

	s.close()
```

```
#Main
if __name__ == '__main__':

#Κλήση συνάρτησης server για συγχρονισμό ώρας
```

Αποτελέσματα Εκτέλεσης

Time returned by server: 2018-11-07 17:56:43.302379

Process Delay latency: 0.0005150819997652434 seconds

Actual clock time at client side: 2018-11-07 17:56:43.302756

Synchronized process client time: 2018-11-07 17:56:43.302637

Synchronization error : 0.000119 seconds

8.2.5 Βελτίωση στο Συγχρονισμό Ρολογιού

Χρησιμοποιώντας επαναληπτικές δοκιμές μέσω του δικτύου, μπορούμε να ορίσουμε έναν ελάχιστο χρόνο μεταφοράς και χρησιμοποιώντας να διατυπώσουμε έναν βελτιωμένο χρόνο ρολογιού συγχρονισμού (μικρότερο σφάλμα συγχρονισμού).

Εδώ, ορίζοντας έναν ελάχιστο χρόνο μεταφοράς, με υψηλή εμπιστοσύνη, μπορούμε να πούμε ότι θα χρειαστεί ο χρόνος του διακομιστή να δημιουργείται πάντα μετά το $T_0 + T_{min}$ και ο T_{SERVER} θα δημιουργείται πάντα πριν από το T_1-T_{min}, όπου το T_{min} είναι ο ελάχιστος χρόνος μεταφοράς που είναι η ελάχιστη τιμή των $T_{REQUEST}$ και $T_{RESPONSE}$ κατά τη διάρκεια αρκετών επαναληπτικών δοκιμών. Εδώ το σφάλμα συγχρονισμού μπορεί να διατυπωθεί ως εξής:

$$error \epsilon \left[-((T_1 - T_0)/2 - T_{min}),\ ((T_1 - T_0)/2 - T_{min})\right]$$

Ομοίως, εάν τα $T_{REQUEST}$ και $T_{RESPONSE}$ διαφέρουν κατά σημαντικό χρονικό διάστημα, μπορούμε να αντικαταστήσουμε το T_{min} με T_{min1} και T_{min2}, όπου T_{min1} είναι ο ελάχιστος παρατηρούμενος χρόνος αιτήματος και T_{min2} αναφέρεται στον ελάχιστο παρατηρούμενο χρόνο απόκρισης μέσω του δικτύου.

Ο συγχρονισμένος χρόνος ρολογιού σε αυτήν την περίπτωση μπορεί να υπολογιστεί ως εξής:

$$T_{CLIENT} = T_{SERVER} + (T_1 - T_0)/2 + (T_{min2} - T_{min1})/2$$

Έτσι, εισάγοντας απλώς τον χρόνο απόκρισης και το χρόνο αίτησης ως ξεχωριστές χρονικές καθυστερήσεις, μπορούμε να βελτιώσουμε τον συγχρονισμό του χρόνου ρολογιού και συνεπώς να μειώσουμε το συνολικό σφάλμα συγχρονισμού. Ο αριθμός των επαναληπτικών δοκιμών που πρέπει να εκτελεστεί εξαρτάται από τη συνολική μετατόπιση του ρολογιού που παρατηρείται.

9 Βιβλιογραφία

1. Jan Palach Parallel Programming with Python Develop efficient parallel systems using the robust Python environment Copyright © 2014 Packt Publishing BIRMINGHAM – MUMBAI Packt Publishing

2. Giancarlo Zaccone Python Parallel Programming Cookbook Second Edition Copyright © 2019 Packt Publishing

3. Brandon Rhodes John Goerzen Foundations of Python Network Programming The comprehensive guide to building network applications with Python Second Edition Apress

4. Κ. ΜΑΓΚΟΥΤΗΣ, Χ. ΝΙΚΟΛΑΟΥ, ΕΙΣΑΓΩΓΗ ΣΤΟΝ ΑΝΤΙΚΕΙΜΕΝΟΣΤΡΑΦΗ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ ΜΕ PYTHON Μια Προσέγγιση από την Πλευρά των Υπολογιστικών Συστημάτων

5. https://www.python-course.eu/threads.php

6. https://www.geeksforgeeks.org/socket-programming-python/

7. https://www.geeksforgeeks.org/multithreading-in-python-set-2-synchronization/

8. https://docs.python.org/3/library/random.html

9. https://docs.python.org/2/library/random.html

10. https://docs.python.org/2.4/lib/semaphore-examples.html

11. https://www.bogotobogo.com/python/Multithread/python_multithreading_Synchronization_Semaphore_Objects_Thread_Pool.php

12. https://www.python-course.eu/graphs_python.php