Εντολές ελέγχου και επανάληψης

Συμπληρωματικές σημειώσεις

Μιχάλης Δρακόπουλος

Μια εφαρμογή: αντιστροφή ψηφίων ακεραίου

Θα χρησιμοποιήσουμε παρόμοια συλλογιστική με αυτήν που χρησιμοποιήσαμε για να βρούμε το άθροισμα των ψηφίων ενός ακεραίου. Για έναν ακέραιο n:

- Εύρεση τελευταίου ψηφίου: `n%10`, π.χ. `197523%10 -> 3`
- Αποκοπή τελευταίου ψηφίου: `n//10`, π.χ. `197523//10 -> 19752`

```
n = int(input('n = '))
is_negative = n < 0
if is_negative:
    n = -n
reversed_n = 0
while n > 0:
    last_digit = n % 10
    reversed_n = 10*reversed_n + last_digit
    n //= 10
if is_negative:
    reversed_n = -reversed_n
print(reversed_n)
```

Η αντιστροφή μπορεί να γίνει εναλλακτικά και με συμβολοσειρές, χρησιμοποιώντας τον τελεστή συνένωσης `+`.

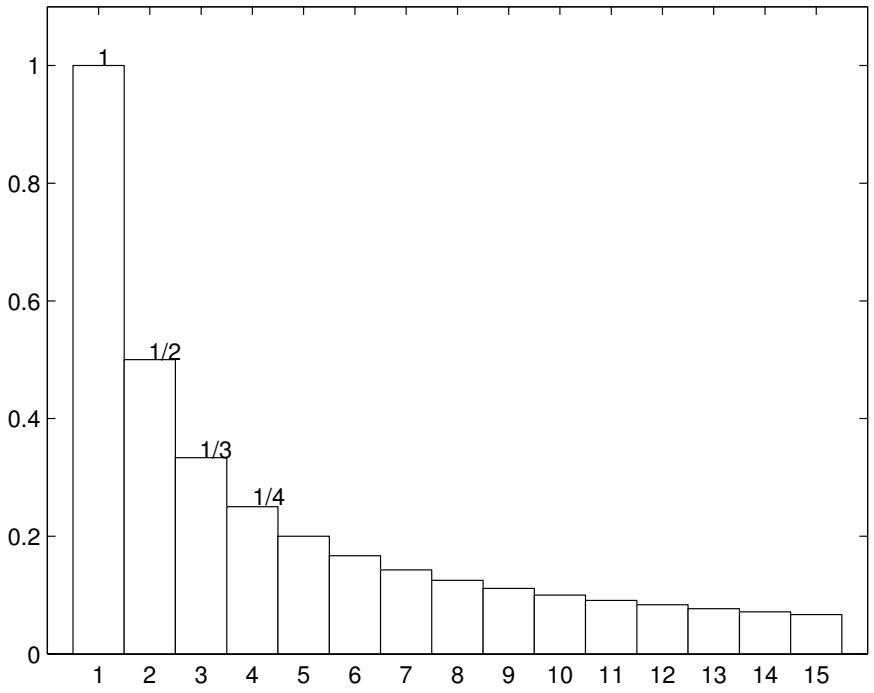
```
n = int(input('n = '))
is_negative = n < 0
if is_negative:
    n = -n
reversed_str = ""
while n > 0:
    last_digit = str(n % 10)
    reversed_str = reversed_str + last_digit
    n //= 10
reversed_n = int(reversed_str)
if is_negative:
    reversed_n = -reversed_n
print(reversed_n)
```

Υπάρχει συντομότερος και κομψότερος τρόπος, αλλά χρειάζεται γνώσεις τεμαχισμού συμβολοσειρών που δεν έχουμε δει ως τώρα.

Αρμονικοί αριθμοί

Ο n-οστός αρμονικός αριθμός:

$$H_n = 1 + 1/2 + 1/3 + 1/4 + \cdots + 1/n$$



Οι αρμονικοί αριθμοί είναι το *διακριτό* ανάλογο του λογαρίθμου:

$$H_n \approx \int_1^n \frac{1}{x}\, dx$$

Η φθίνουσα ακολουθία $H_n - \ln(n)$ συγκλίνει στην σταθερά γ του Euler:

$$\lim_{n\to\infty} H_n - \ln(n) = 0.577215\cdots = \gamma$$

```
import math

n = int(input('n = '))
H_n = 0
```

```
for i in range(1, n+1):
    H_n += 1/i
print(f'H_{n} = {H_n}')
print(f'ln({n}) = {math.log(n)}')
print(f'gamma = {H_n - math.log(n)}')
```

Επεξεργασία αγνώστου πλήθους δεδομένων

Τα δεδομένα εισάγονται ένα-ένα. Αν είναι δυνατή η καταμέτρησή τους, χρησιμοποιούμε παραλλαγές του γενικού αλγορίθμου (τα 4 στάδια της επεξεργασίας εμφανίζονται ως σχόλια):

```
n = int(input('Πλήθος δεδομένων? '))
# Αρχική επεξεργασία
for k in range(n):
    # Διάβασε δεδομένο k
    # Επεξεργασία δεδομένου k
# Τελική επεξεργασία
```

Ας εφαρμόσουμε τον παραπάνω αλγόριθμο σε ένα απλό πρόβλημα, όπως το άθροισμα n πραγματικών αριθμών.

```
n = int(input('Πλήθος δεδομένων? '))
sum = 0                                          # Αρχική επεξεργασία
for k in range(n):
    value = float(input(f'Δεδομένο {k+1}? '))    # Διάβασε δεδομένο k
    sum += value                                 # Επεξεργασία δεδομένου k
print(sum)                                       # Τελική επεξεργασία
```

Όταν το πλήθος των δεδομένων είναι πολύ μεγάλο, και άρα δύσκολο να μετρηθεί, η εφαρμογή του παραπάνω αλγορίθμου είναι πρακτικά απαγορευτική. Στην περίπτωση αυτή χρησιμοποιούμε ένα τέχνασμα:

- Αρχικά επιλέγουμε κάποια *ειδική τιμή τέλους*, η οποία δεν μπορεί να είναι μέρος των επιτρεπτών τιμών που μπορούν να πάρουν τα δεδομένα μας. Αυτό είναι πάντα εφικτό γιατί τα δεδομένα που επεξεργαζόμαστε αντιστοιχούν σε μετρήσεις, υπολογισμούς κλπ και έχουν κάποια φυσική ή άλλη σημασία από τον πραγματικό ή ιδεατό κόσμο. Αν, για παράδειγμα, τα δεδομένα αντιστοιχούν σε αποστάσεις δεν μπορούν να πάρουν αρνητικές τιμές.
- Στη συνέχεια εισάγουμε τα δεδομένα ένα-ένα και ελέγχουμε αν κάποιο από αυτά έχει αυτή την ειδική τιμή που σηματοδοτεί το τέλος τους.

Για τη μέθοδο αυτή ο γενικός αλγόριθμος είναι ο εξής:

```
# Αρχική επεξεργασία
end_of_data = # Ειδική τιμή τέλους
value = # Διάβασε το πρώτο δεδομένο
while value != end_of_data:
    # Επεξεργασία δεδομένου
    value = # Διάβασε το επόμενο δεδομένο
# Τελική επεξεργασία
```

Για το γινόμενο αγνώστου πλήθους *θετικών* αριθμών θα μπορούσαμε να επιλέξουμε ως `end_of_data` οποιαδήποτε τιμή ≤ 0.

```
prod = 1                                        # Αρχική επεξεργασία
end_of_data = 0                                 # Ειδική τιμή τέλους
value = float(input('Πρώτο δεδομένο? '))        # Διάβασε πρώτο δεδομένο
while value != end_of_data:
    prod *= value                               # Επεξεργασία δεδομένου
    value = float(input('Επόμενο δεδομένο? '))  # Διάβασε επόμενο δεδομένο
print(prod)                                     # Τελική επεξεργασία
```

Παρατηρούμε ότι χρειάζεται να διαβάσουμε ξεχωριστά το πρώτο από τα δεδομένα προκειμένου να ξεκινήσει η επαναληπτική διαδικασία. Εναλλακτικά μπορούμε να εντάξουμε την ανάγνωση *όλων* των δεδομένων στην επανάληψη χρησιμοποιώντας την εντολή `break` στην επόμενη αλγοριθμική παραλλαγή:

```
# Αρχική επεξεργασία
end_of_data = # Ειδική τιμή τέλους
while True:
    value = # Διάβασε το τρέχον δεδομένο
    if value == end_of_data:
        break
    # Επεξεργασία δεδομένου
# Τελική επεξεργασία
```

και ο αλγόριθμος για το γινόμενο γίνεται:

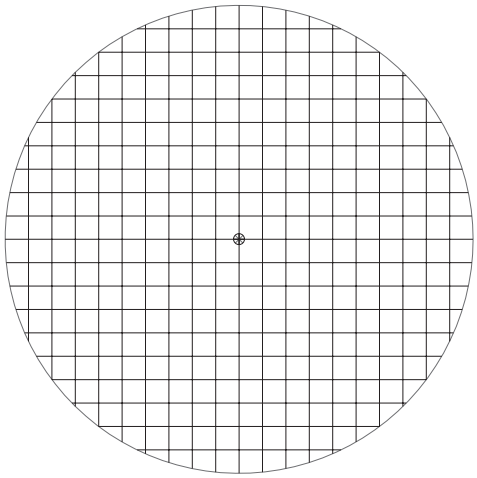
```
prod = 1                                        # Αρχική επεξεργασία
end_of_data = 0                                 # Ειδική τιμή τέλους
while True:
    value = float(input('Επόμενο δεδομένο? '))  # Διάβασε τρέχον δεδομένο
    if value == end_of_data:
        break
    prod *= value                               # Επεξεργασία δεδομένου
print(prod)                                     # Τελική επεξεργασία
```

Σε μεγάλες πρακτικές εφαρμογές, τα δεδομένα είναι συνήθως σύνθετοι τύποι και όχι απλοί αριθμοί και η επεξεργασία (αρχική και κάθε δεδομένου) αντιστοιχεί σε πολλές γραμμές κώδικα.

Υπολογιστική προσέγγιση του π

Κάλυψη δίσκου

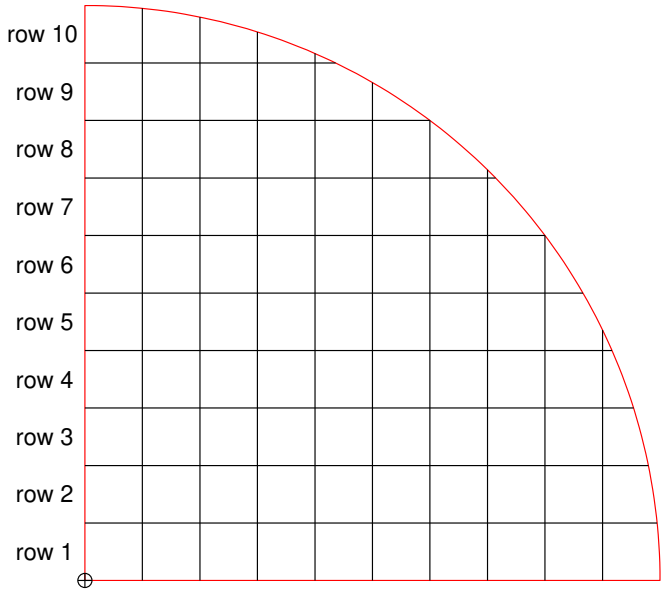
Έστω κύκλος $x^2 + y^2 = n^2$ με ακέραια ακτίνα n και εμβαδόν $E = \pi n^2$. Η προσεγγιστική τιμή του π, υπολογίζεται από κάποια προσέγγιση του E. Μια τέτοια προσέγγιση προκύπτει αν καλύψουμε τον δίσκο με τετράγωνα μοναδιαίου εμβαδού 1×1 και μετρήσουμε πόσα από αυτά είναι ολόκληρα (χωρίς να κόβονται) μέσα στον δίσκο.



Αν ο δίσκος καλύπτεται από N ολόκληρα τετράγωνα τότε:

$$N \approx \pi n^2 \Rightarrow \rho_n = \frac{N}{n^2} \approx \pi$$

Λόγω συμμετρίας, αρκεί να μετρήσουμε τα ολόκληρα τετράγωνα σε ένα τεταρτημόριο, έστω N_1, οπότε $\rho_n = 4N_1/n^2$.



Από το παραπάνω σχήμα παρατηρούμε ότι κατά μήκος της επάνω "πλευράς" της k γραμμής τετραγώνων ισχύει ότι $y = k$, καθώς και ότι στη διασταύρωση του $y = k$ με τον κύκλο $x^2 + y^2 = n^2$ ισχύει ότι

$$x = \sqrt{n^2 - k^2}$$

Επομένως, το πλήθος των ολόκληρων τετραγώνων στη γραμμή k είναι ο μεγαλύτερος ακέραιος που είναι μικρότερος ή ίσος της τιμής αυτής του x, δηλαδή η απόλυτη τιμή $|x|$.

```python
# Προσέγγιση του π μέσω “κάλυψης”

import math

# Διάβασε την ακτίνα του δίσκου...
n = int(input('Δώσε την ακτίνα του κύκλου n: '))

# Κάλυψε το πρώτο τεταρτημόριο και μετά πολλαπλασίασε επί 4...
N1 = 0
for k in range(1, n+1):
    # Πρόσθεσε το πλήθος των ολόκληρων τετραγώνων της γραμμής k...
    m = math.floor(math.sqrt(n**2 - k**2))
    N1 = N1 + m

# Εμφάνισε την προσέγγιση...
rho_n = 4*N1/n**2

print(f'n      = {n}')
print(f'rho_n  = {rho_n:.8f}')
print(f'Σφάλμα = {abs(math.pi - rho_n):.8f}')
```

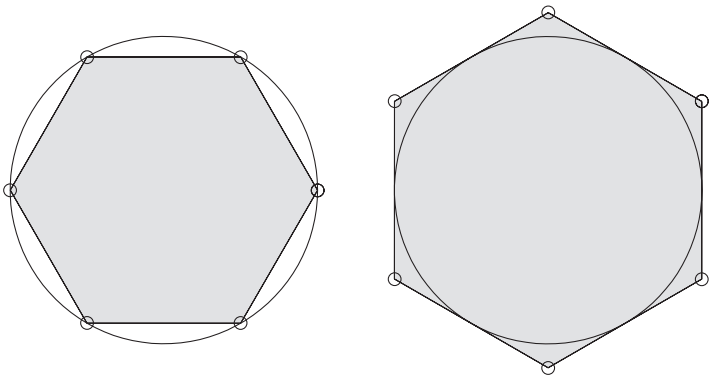
Προφανώς, όσο μεγαλύτερη είναι η ακτίνα n του κύκλου τόσο περισσότερα μοναδιαία τετράγωνα καλύπτουν τον δίσκο και τόσο καλύτερη και η προσέγγιση.

Προσέγγιση με εγγεγραμμένα/περιγεγραμμένα πολύγωνα

Έστω μοναδιαίος κύκλος $x^2 + y^2 = 1$, με εμβαδόν π. Τα εμβαδά του εγγεγραμμένου και του περιγεγραμμένου *κανονικού* n-γώνου είναι

$$A_n = \frac{n}{2}\sin\left(\frac{2\pi}{n}\right), \text{ και } B_n = n\tan\left(\frac{\pi}{n}\right)$$

αντίστοιχα.



Για κάθε n ισχύει:

$$A_n < \pi < B_n$$

και ο μέσος όρος των εμβαδών των πολυγώνων:

$$\rho_n = \frac{A_n + B_n}{2}$$

προσεγγίζει το π (= το εμβαδόν του μοναδιαίου κύκλου), με απόλυτο σφάλμα:

$$|\rho_n - \pi| < B_n - A_n$$

το οποίο συγκλίνει στο μηδέν καθώς $n \to \infty$.

```python
# Προσέγγιση του π μέσω Πολυγώνων

import math

# Διάβασε τις παραμέτρους επανάληψης...
delta = float(input('Δώσε την ανοχή του σφάλματος: '))
n_max = int(input('Δώσε το όριο των επαναλήψεων: '))

# Η περίπτωση τριγώνου...
n = 3                                   # Αριθμός ακμών πολυγώνου
A_n = (n/2)*math.sin(2*math.pi/n)       # Εγγεγραμμένη περιοχή
B_n = n*math.tan(math.pi/n)             # Περιγεγραμμένη περιοχή
error_bound = B_n - A_n                  # Το όριο σφάλματος

# Επανάλαβε όσο το σφάλμα είναι πολύ μεγάλο και το n αρκετά μικρό...
while error_bound > delta and n < n_max:
    n = n+1
    A_n = (n/2)*math.sin(2*math.pi/n)
    B_n = n*math.tan(math.pi/n)
    error_bound = B_n - A_n

# Εμφάνισε την τελική προσέγγιση...
nStar = n
rho_nStar = (A_n + B_n)/2
print()
print(f' delta = {delta}\n nStar = {nStar}\n n_max = {n_max}\n\n')
print(f' rho_nStar = {rho_nStar}\n Pi = {math.pi}\n')
```

for μέσα σε for

Πίνακας πολλαπλασιασμού

Κώδικας που να εμφανίζει τον παρακάτω πίνακα:

```
 1   2   3   4   5   6   7   8   9  10
 2   4   6   8  10  12  14  16  18  20
 3   6   9  12  15  18  21  24  27  30
 4   8  12  16  20  24  28  32  36  40
```

```
 5  10  15  20  25  30  35  40  45  50
 6  12  18  24  30  36  42  48  54  60
 7  14  21  28  35  42  49  56  63  70
 8  16  24  32  40  48  56  64  72  80
 9  18  27  36  45  54  63  72  81  90
10  20  30  40  50  60  70  80  90 100
```

Κάθε στοιχείο του πίνακα βρίσκεται στη γραμμή `i` και στη στήλη `j` και η τιμή του είναι το `i*j`. Για να σχηματίσουμε π.χ. τη γραμμή 3, θα γράφαμε στην Python:

```
i = 3
for j in range(1, 11):
    ij = i*j
    print(f'{ij:4d}', end='')
print()
```

Για να σχηματίσουμε *όλες* τις γραμμές, ο παραπάνω κώδικας θα έπρεπε να εκτελεστεί διαδοχικά για `i = 1, 2, 3, ..., 10`. Δηλαδή:

```
for i in range(1, 11):
    for j in range(1, 11):
        ij = i*j
        print(f'{ij:4d}', end='')
    print()
```

ASCII art

Σχεδιασμός κυκλικού δίσκου ακτίνας `R` χρησιμοποιώντας χαρακτήρες και την `print`. Παράδειγμα για `R = 8` θέλουμε να εμφανιστεί

```
. . . . . . . . * . . . . . . . .
. . . . . * * * * * * * . . . . .
. . . * * * * * * * * * * * . . .
. . * * * * * * * * * * * * * . .
. . * * * * * * * * * * * * * . .
. * * * * * * * * * * * * * * * .
. * * * * * * * * * * * * * * * .
. * * * * * * * * * * * * * * * .
* * * * * * * * * * * * * * * * *
. * * * * * * * * * * * * * * * .
. * * * * * * * * * * * * * * * .
. * * * * * * * * * * * * * * * .
. . * * * * * * * * * * * * * . .
. . * * * * * * * * * * * * * . .
. . . * * * * * * * * * * * . . .
. . . . . * * * * * * * . . . . .
. . . . . . . . * . . . . . . . .
```

Θα προσδιορίσουμε τη θέση κάθε χαρακτήρα με συντεταγμένες `(i, j)`. Το κέντρο του δίσκου είναι το `(0, 0)` και οι 4 γωνίες δεξιόστροφα ξεκινώντας από πάνω αριστερά είναι: `(-R, -R)`, `(-R, +R)`,

(+R, +R), (+R, -R). Υ i αντιστοιχεί στις γραμμές και το j στους χαρακτήρες-σημεία μιας γραμμής.

```
R = int(input('R = '))
for i in range(-R, R+1):
    for j in range(-R, R+1):
        if i*i + j*j <= R*R:
            print('* ', end='')
        else:
            print('. ', end='')
    print()
```

Ρητή προσέγγιση του π

Για δεδομένο M, θέλουμε να βρούμε την καλύτερη ρητή προσέγγιση p/q στο π, με τους ακεραίους p και q να ικανοποιούν τις ανισώσεις $1 \leq p, q \leq M$.

Μια απλή, αλλά όχι αποτελεσματική προσέγγιση θα ήταν για *κάθε* q από το 1 έως και το M να ελέγξουμε *όλα* τα p από το 1 έως και το M και να κρατήσουμε το καλύτερο p/q.

```
for q in range(1, M+1):
    for p in range(1, M+1):
        # έλεγχος αν p/q καλύτερο από προηγούμενα
```

Ο αλγόριθμος αυτός είναι σωστός *αλλά* οι εντολές για τον έλεγχο θα εκτελεστούν M^2 φορές συνολικά. Είναι ένα παράδειγμα *εξαντλητικού* αλγορίθμου που ελέγχει όλες τις δυνατές περιπτώσεις (εδώ όλους τους συνδυασμούς των p και q).

Μπορούμε να επιταχύνουμε δραστικά την εκτέλεση του προγράμματος παρατηρώντας ότι για δεδομένο q, οι τιμές του p που χρειάζεται να ελέγξουμε είναι οι $p_- = \text{floor}(q\pi)$ και $p_+ = \text{ceil}(q\pi)$. Αυτό γιατί οι τιμές του p που είναι μικρότερες του p_- δίνουν λιγότερο ακριβείς προσεγγίσεις από το p_-/q. Παρομοίως τιμές του p μεγαλύτερες από το p_+ είναι επίσης λιγότερο ακριβείς από το p_+/q. Έτσι αντί να ελέγξουμε M πιθανούς αριθμητές ελέγχουμε μόνο δύο. Με αυτό τον τρόπο ο αριθμός των ελέγχων στα πηλίκα μειώνεται σε $2M$. Άλλη μια εξοικονόμηση υπολογιστικού χρόνου προκύπτει αν παρατηρήσουμε ότι οι τιμές του q αρκεί να είναι ανάμεσα στο ένα και στο ceil(M/π) μιας και μεγαλύτερες τιμές αντιστοιχούν σε αριθμητές που είναι μεγαλύτεροι του M . Ενσωματώνοντας αυτές τις παρατηρήσεις καταλήγουμε σε μια πολύ πιο αποτελεσματική υλοποίηση του αλγορίθμου μας

```
# Ρητή προσέγγιση του π
import math

M = int(input('Δώσε την τιμή του M: '))

# Καλύτερος αριθμητής "ως τώρα"...
pBest = 1
# Καλύτερος παρανομαστής "ως τώρα"...
qBest = 1
# Σφάλμα στο τρέχον καλύτερο κλάσμα...
err_pq = abs(pBest/qBest - math.pi)
# Έλεγξε όλους τους πιθανούς παρανομαστές...
```

```
for q in range(1, math.ceil(M/math.pi)+1):
    pMinus = math.floor(math.pi*q);  errMinus = abs(math.pi - pMinus/q)
    pPlus = math.ceil(math.pi*q);  errPlus = abs(math.pi - pPlus/q)
    if errMinus < err_pq:
        pBest = pMinus; qBest = q;  err_pq = errMinus
    if errPlus < err_pq:
        pBest = pPlus;  qBest = q; err_pq = errPlus
MyPi = pBest/qBest

print(f'M = {M}\npBest = {pBest}\nqBest = {qBest}')
print(f'MyPi = {MyPi}\npi = {math.pi}\nerror = {err_pq:.15f}')
```

Εφαρμογή: Ακολουθία Fibonacci

Είναι η ακολουθία $1, 2, 3, 5, 8, 13, \ldots$ που υπολογίζεται *αναδρομικά*:

$$f_n = f_{n-1} + f_{n-2},\ n > 2 \quad \text{με } f_1 = 1, f_2 = 2.$$

Ένα πρώτο πρόγραμμα που υπολογίζει τον n-οστό όρο είναι το εξής:

```
# n-οστός όρος Fibonacci - ver. 1
n = int(input('n = '))
if n == 1:
    f_curr = 1
else:
    f_old = 1; f_curr = 2
    for i in range(3, n+1):
        f_new = f_curr + f_old
        f_old = f_curr
        f_curr = f_new
print(f'f_{n} = {f_curr}')
```

Χρησιμοποιήσαμε 3 μεταβλητές `f_old`, `f_curr` και `f_new` για τον προηγούμενο, τον τρέχοντα, και τον επόμενο όρο αντίστοιχα.

Το πρόγραμμα μπορεί να γραφτεί με 2 μόνο μεταβλητές `f_curr` και `f_old` για τον τρέχοντα και τον προηγούμενο όρο αντίστοιχα.

```
# n-οστός όρος Fibonacci - ver. 2
n = int(input('n = '))
if n == 1:
    f_curr = 1
else:
    f_old = 1; f_curr = 2
    for i in range(3, n+1):
        f_curr = f_curr + f_old
        f_old = f_curr - f_old
print(f'f_{n} = {f_curr}')
```

Σε κάθε επανάληψη, για το νέο όρο της ακολουθίας μπορούμε να χρησιμοποιήσουμε ξανά τη μεταβλητή `f_curr` θέτοντας: `f_curr = f_curr + f_old`. Παρατηρήστε ότι η προηγούμενη τιμή της `f_curr`, που τώρα πρέπει να εκχωρηθεί στη μεταβλητή `f_old`, δεν έχει χαθεί και μπορεί να υπολογιστεί ξανά ως εξής: `f_old = f_curr - f_old`.

Τέλος μια τρίτη παραλλαγή του προγράμματος, εκμεταλλεύεται τη δυνατότητα της Python για ταυτόχρονη εκχώρηση πολλών τιμών σε αντίστοιχες μεταβλητές.

```
# n-οστός όρος Fibonacci - ver. 3
n = int(input('n = '))
if n == 1:
    f_curr = 1
else:
    f_old = 1; f_curr = 2
    for i in range(3, n+1):
        f_curr, f_old = f_curr+f_old, f_curr
print(f'f_{n} = {f_curr}')
```