141. Πληροφορική Ι — Σημειώσεις

Δημήτρης Θηλυκός Ζαφειράκης Ζαφειρακόπουλος

2026-10-26

Πίνακας περιεχομένων

1 Αλγόριθμοι και πρώτα βήματα στην Python **2**
1.1 Τι κάνουμε σε αυτό το μάθημα . . . 2
1.2 Τι είναι αλγόριθμος . . . 2
1.3 Από το πρόβλημα στο πρόγραμμα . . . 2
1.4 Παράδειγμα: γραμμική αναζήτηση . . . 2
1.5 Χαρακτηριστικά της Python . . . 4
1.6 Μεταβλητές και τύποι . . . 4
1.7 Πολλαπλή ανάθεση και ανταλλαγή τιμών . . . 4
1.8 Μεταβλητές ως αναφορές . . . 4
1.9 Είσοδος με `input` . . . 5
1.10 Έξοδος με `print` . . . 5
1.11 Μορφοποίηση και f-strings . . . 5
1.12 Βιβλιοθήκες: τι είναι και πώς τις χρησιμοποιούμε . . . 6
1.13 Γραφική έξοδος με `matplotlib` . . . 7
1.14 Σύνοψη των πρώτων βημάτων . . . 10

2 Ανάθεση μεταβλητών, είσοδος και έξοδος **10**
2.1 Το πρώτο πρόγραμμα: `input`, `int`, `float`, `print` και f-strings . . . 10
2.2 Βιβλιοθήκη `math`, δυνάμεις και προτεραιότητα πράξεων . . . 11
2.3 Μονάδες μέτρησης και ενδιάμεσες μεταβλητές . . . 11
2.4 Πολλές γραμμές εξόδου, εισαγωγικά μέσα σε κείμενο και ένα πρώτο γράφημα . . . 12
2.5 Μέγιστο και ελάχιστο . . . 13
2.6 Αλφαριθμητικά: ένωση και δείκτες . . . 14

3 Συνθήκες: η εντολή `if` **14**
3.1 Σύγκριση: μέγιστο και ελάχιστο χωρίς `if` . . . 14
3.2 `if … else` και εμφωλευμένα `if` . . . 15
3.3 `elif`: πολλές περιπτώσεις στο ίδιο επίπεδο . . . 15
3.4 Ένα πραγματικό πρόβλημα: ελάχιστο του $f(x) = x^2 + bx + c$ στο $[L, R]$ . . . 15
3.5 Ανάλυση κυβικού πολυωνύμου: `if` μέσα σε `else`, `math.sqrt` και γράφημα . . . 17

4 Επανάληψη με `for` **18**
4.1 Παραγοντικό: ο «συσσωρευτής» . . . 19
4.2 Έλεγχος πρώτου αριθμού: σημαίες, `%` και υπό συνθήκη έκφραση . . . 19
4.3 Βελτίωση 1: μόνο περιττοί διαιρέτες, το `step` της `range` . . . 19
4.4 Βελτίωση 2: διαιρέτες μέχρι την τετραγωνική ρίζα . . . 20
4.5 Βρόχος μέσα σε βρόχο σε ένα διάστημα και το `not` . . . 20
4.6 Λίστες, `break`, `math.isqrt` και γράφημα των πρώτων . . . 20

5 Τυχαίοι αριθμοί και προσεγγίσεις του π **21**
5.1 Η βιβλιοθήκη `random` και νέφος σημείων . . . 21
5.2 Η σειρά του Leibniz: εμφωλευμένοι βρόχοι και σύγκλιση . . . 22
5.3 Βελάκια: Monte Carlo με `random.uniform` . . . 22
5.4 Η βελόνα του Buffon: `floor`, `ceil` και πρόσθεση λογικών τιμών . . . 22
5.5 Σημεία πλέγματος σε κύκλο: μέτρηση χωρίς τυχαιότητα . . . 22

6 Επανάληψη με `while` 23
6.1 Επανάληψη μέχρι να δοθεί σωστή είσοδος 23
6.2 Έλεγχος πρώτου με `while` 23
6.3 Έλεγχος πρώτου με `while` που σταματά νωρίς 23
6.4 Λίστα λογικών τιμών και σωρευτικό άθροισμα 24
6.5 Μέτρηση διαιρετών και `while` μέσα σε `for` 24

7 Σύνοψη των στοιχείων της γλώσσας 25

> **i** Πώς να διαβάσετε τις σημειώσεις
>
> Οι σημειώσεις ακολουθούν τα προγράμματα που είδαμε στην τάξη. Τα προγράμματα παρατίθενται **αυτούσια** (το όνομα του αρχείου φαίνεται πάνω από κάθε μπλοκ) και ακολουθεί η εξήγηση των **στοιχείων της γλώσσας** που εμφανίζονται για πρώτη φορά.

1 Αλγόριθμοι και πρώτα βήματα στην Python

Τα παραδείγματα αυτής της ενότητας είναι μικρά: δείχνουν τις έννοιες που χρειάζονται για να διαβάσουμε τα πλήρη προγράμματα των επόμενων ενοτήτων.

1.1 Τι κάνουμε σε αυτό το μάθημα

- Εξοικείωση με την **αλγοριθμική σκέψη**.
- Εισαγωγή στη γλώσσα προγραμματισμού **Python**.
- Πώς κάνουμε μαθηματικά στην τρίτη χιλιετία: υπολογισμοί, πειράματα, γραφήματα.

1.2 Τι είναι αλγόριθμος

Αλγόριθμος είναι μια βήμα προς βήμα διαδικασία επίλυσης ενός προβλήματος, με πεπερασμένο πλήθος βημάτων. Πρέπει:

- να δέχεται **είσοδο** και να παράγει **έξοδο**,
- να είναι **ορισμένος**: κάθε οδηγία να είναι σαφής και, από μόνη της, απλή,
- να καλύπτει **όλες** τις δυνατές περιπτώσεις εισόδου,
- να επιστρέφει πάντα τη **σωστή απάντηση**,
- να **τερματίζει** πάντα, σε πεπερασμένα βήματα.

Τα προγράμματα των επόμενων ενοτήτων ελέγχονται με αυτά τα κριτήρια. Για παράδειγμα, στις συνθήκες θα δούμε πρόγραμμα που δεν καλύπτει την περίπτωση $L > R$ σωστά, και στην επανάληψη `while` έναν βρόχο που αν γραφτεί λάθος δεν τερματίζει ποτέ.

1.3 Από το πρόβλημα στο πρόγραμμα

1. Ανάλυση των δεδομένων.
2. Μαθηματική διατύπωση.
3. Σχεδιασμός ή επιλογή αλγορίθμου.
4. Διατύπωση σε γλώσσα προγραμματισμού.
5. Εκτέλεση.
6. Ερμηνεία των αποτελεσμάτων.

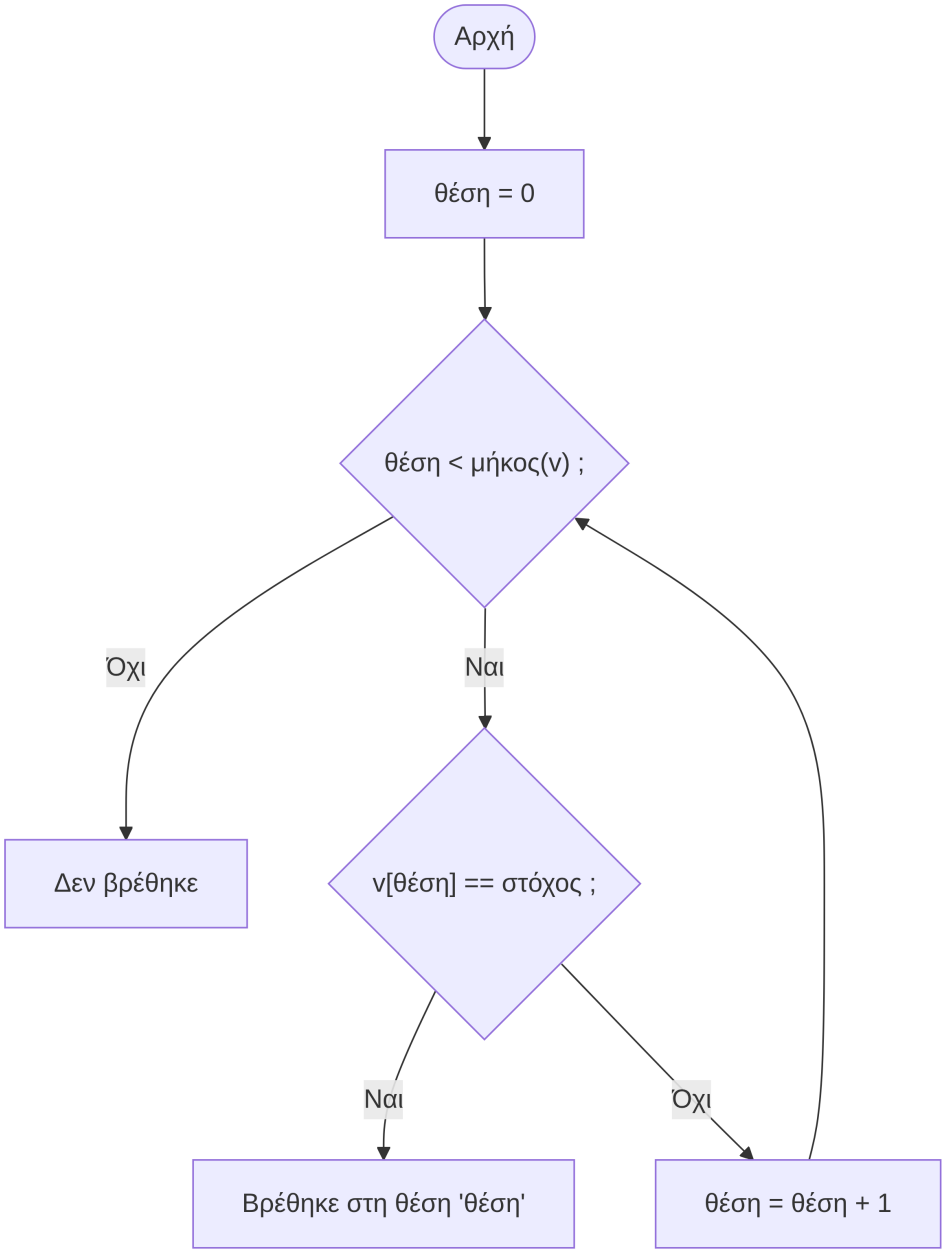
Ένας αλγόριθμος περιγράφεται με **φυσική γλώσσα**, **διάγραμμα ροής**, **ψευδοκώδικα** ή **κώδικα**. Οι τέσσερις μορφές φαίνονται στο παράδειγμα που ακολουθεί.

1.4 Παράδειγμα: γραμμική αναζήτηση

> Δίνεται λίστα `v` και μια τιμή-στόχος. Βρες τη θέση της στη λίστα, ή διαπίστωσε ότι δεν υπάρχει.

Φυσική γλώσσα. Σαρώνουμε τη λίστα από την αρχή, στοιχείο προς στοιχείο, και σταματάμε μόλις βρεθεί ο στόχος.

Διάγραμμα ροής.



Ψευδοκώδικας.

```
ΓΡΑΜΜΙΚΗ_ΑΝΑΖΗΤΗΣΗ(v, στόχος):
    για θέση από 0 έως μήκος(v) - 1:
        αν v[θέση] = στόχος:
            επίστρεψε θέση
    επίστρεψε -1
```

Κώδικας Python. Πρόγευση: τα `for`, `if`, οι λίστες και το `def` εξηγούνται στις επόμενες ενότητες.

```python
def linear_search(v, target):
    for i in range(len(v)):
        if v[i] == target:
            return i
    return -1

data = [15, 8, 42, 4, 16, 23, 100]   # όχι απαραίτητα ταξινομημένη
target = 16   # δοκιμάστε και τιμές που δεν υπάρχουν
```

```
pos = linear_search(data, target)
print(f'Θέση: {pos}' if pos != -1 else 'Δεν βρέθηκε')
```

Συγκρίνετε τον ψευδοκώδικα με τον κώδικα γραμμή προς γραμμή: η αντιστοιχία είναι σχεδόν ένα προς ένα. Αν η λίστα είναι ταξινομημένη υπάρχει πολύ ταχύτερη μέθοδος, η **δυαδική αναζήτηση**, που θα δούμε με τον βρόχο `while`.

1.5 Χαρακτηριστικά της Python

- **Διερμηνευόμενη** γλώσσα: ένα πρόγραμμα (ο διερμηνευτής, interpreter) διαβάζει τον κώδικά μας γραμμή προς γραμμή και τον εκτελεί. Δουλεύουμε είτε **διαδραστικά** (γράφουμε μια εντολή, βλέπουμε αμέσως το αποτέλεσμα) είτε με **script**, δηλαδή αρχείο `.py` που εκτελείται ολόκληρο. Όλα τα προγράμματα αυτών των σημειώσεων είναι scripts.
- **Δυναμικοί τύποι**: δεν δηλώνουμε τον τύπο μιας μεταβλητής, προκύπτει από την τιμή της.
- **Υποχρεωτική εσοχή** (στοίχιση): τα μπλοκ εντολών ορίζονται από την εσοχή (κενά στην αρχή της γραμμής) και όχι από άγκιστρα ή λέξεις όπως `begin`/`end`.
- Μεγάλη **πρότυπη βιβλιοθήκη** και ακόμη μεγαλύτερο οικοσύστημα εξωτερικών βιβλιοθηκών.
- Δημιουργήθηκε από τον Guido van Rossum στις αρχές της δεκαετίας του '90. Χρησιμοποιούμε την Python 3.

1.6 Μεταβλητές και τύποι

Μια μεταβλητή είναι ένα **όνομα** που δείχνει σε ένα **αντικείμενο** στη μνήμη. Ο τύπος του αντικειμένου προσδιορίζεται αυτόματα από την τιμή:

```
room = 506        # int
price = 3.99       # float
name = 'George'    # str

print(type(room))
print(type(price))
print(type(name))
```

Η συνάρτηση `type(x)` επιστρέφει τον τύπο του `x`: `<class 'int'>`, `<class 'float'>`, `<class 'str'>`. Οι τρεις αυτοί τύποι, ακέραιοι, πραγματικοί και αλφαριθμητικά (strings, αλλιώς «συμβολοσειρές»), είναι όλοι όσοι χρειάζονται στις πρώτες ενότητες. Αργότερα προστίθενται οι λογικές τιμές (`bool`) και οι λίστες (`list`).

Κανόνες για τα ονόματα. Ένα όνομα μεταβλητής ξεκινά με γράμμα ή `_`, περιέχει γράμματα, ψηφία και `_`, δεν περιέχει κενά ή τελείες, και δεν μπορεί να είναι δεσμευμένη λέξη της γλώσσας (`if`, `for`, `while`, `else`, `import`, `def` κ.λπ.). Η Python κάνει διάκριση πεζών-κεφαλαίων: `r` και `R` είναι διαφορετικές μεταβλητές (και στο `4_torus.py` παρακάτω είναι και οι δύο ακτίνες του τόρου).

1.7 Πολλαπλή ανάθεση και ανταλλαγή τιμών

```
x, y, z = 1, 2, 3
print(x, y, z)
```

```
a, b = 5, 10
print('Πριν:', a, b)

a, b = b, a        # ανταλλαγή τιμών, χωρίς βοηθητική μεταβλητή
print('Μετά:', a, b)
```

Στην πολλαπλή ανάθεση η Python υπολογίζει **πρώτα όλο το δεξί μέλος** και μετά αναθέτει τις τιμές στα ονόματα του αριστερού. Γι' αυτό το `a, b = b, a` ανταλλάσσει τις τιμές σωστά, ενώ οι δύο ξεχωριστές εντολές `a = b` και `b = a` θα έδιναν στα δύο ονόματα την ίδια τιμή.

1.8 Μεταβλητές ως αναφορές

Ένα όνομα **δείχνει** σε ένα αντικείμενο, δεν είναι «κουτί» που περιέχει μια τιμή:

```
a = 1000
b = a
print(a is b, id(a) == id(b))   # True, True -- ίδιο αντικείμενο

b = b + 1      # ΝΕΟ αντικείμενο· το a δεν επηρεάζεται
print(a, b)
print(a is b)  # False πλέον
```

Η `id(x)` επιστρέφει τη «διεύθυνση» του αντικειμένου και ο τελεστής `is` ελέγχει αν δύο ονόματα δείχνουν στο **ίδιο** αντικείμενο (ενώ το `==` ελέγχει αν έχουν ίση τιμή). Μετά το `b = a` τα δύο ονόματα δείχνουν στο ίδιο αντικείμενο. Το `b = b + 1` όμως δημιουργεί **νέο** αντικείμενο με τιμή 1001 και κάνει το `b` να δείχνει εκεί, ενώ το `a` συνεχίζει να δείχνει στο 1000.

Για **αμετάβλητους** τύπους (αριθμοί, αλφαριθμητικά) η διάκριση περνά απαρατήρητη. Για **μεταβλητούς** τύπους, όπως οι λίστες που θα δούμε στην επανάληψη `for`, έχει ορατές συνέπειες: αν `b = a` και η `a` είναι λίστα, το `b.append(5)` αλλάζει και τη λίστα που βλέπει το `a`, γιατί είναι η ίδια λίστα.

1.9 Είσοδος με `input`

```
age = int(input('Δώσε την ηλικία σου: '))   # input() επιστρέφει πάντα str
```

Η `input` τυπώνει το μήνυμα, περιμένει μια γραμμή από το πληκτρολόγιο και την επιστρέφει **πάντα ως αλφαριθμητικό**. Για να γίνει αριθμός χρειάζεται μετατροπή με `int(...)` ή `float(...)`. Στα προγράμματα που ακολουθούν η γραμμή `x = float(input('...'))` εμφανίζεται σε σχεδόν κάθε αρχείο.

```
weight = float('72.5')   # προσομοιώνει: float(input('Βάρος σε kg: '))
height = float('1.80')   # προσομοιώνει: float(input('Ύψος σε m: '))
bmi = weight / height**2
print(f'Δείκτης μάζας σώματος: {bmi:.2f}')
```

Το παράδειγμα δείχνει και ότι η `float` δέχεται αλφαριθμητικό και επιστρέφει αριθμό: `float('72.5')` είναι ακριβώς αυτό που κάνει το `float(input(...))` όταν ο χρήστης πληκτρολογήσει `72.5`.

1.10 Έξοδος με `print`

```
print(1, 2, 3, sep='***')
print('a', 'b', 'c', sep='-', end=' <- τέλος πρώτης γραμμής\n')
print('δεύτερη γραμμή')
```

Η `print` δέχεται όσα ορίσματα θέλουμε, χωρισμένα με κόμμα, και τα τυπώνει στη σειρά. Δύο προαιρετικά ορίσματα ρυθμίζουν τη μορφή:

- `sep=` είναι ό,τι μπαίνει **ανάμεσα** στα ορίσματα (προεπιλογή: ένα κενό),
- `end=` είναι ό,τι μπαίνει **στο τέλος** (προεπιλογή: αλλαγή γραμμής `'\n'`).

Το `end=' '` θα το ξαναδούμε στο `prime_sqrt.py`, για να τυπωθούν οι πρώτοι αριθμοί στην ίδια γραμμή.

> **!** Το `+` δεν ενώνει αριθμό με κείμενο
>
> Το `'Ηλικία: ' + 20` δίνει `TypeError`: ο τελεστής `+` δουλεύει μόνο ανάμεσα σε τιμές ίδιου τύπου. Χρειάζεται `'Ηλικία: ' + str(20)`, ή κόμμα στην `print`, ή f-string.

1.11 Μορφοποίηση και f-strings

```
product = 'Καφές'
price = 3.5
quantity = 4
total = price * quantity

print(f'{product}: {quantity} x {price:.2f}€ = {total:.2f}€')
```

```
print(f'Έκπτωση 15%: -{total * 0.15:.2f}€')
print(f'Τελική τιμή: {total * 0.85:8.2f}€')
```

Το f-string είναι αλφαριθμητικό με `f` πριν τα εισαγωγικά. Μέσα σε άγκιστρα `{}` μπαίνει οποιαδήποτε έκφραση (ακόμη και `total * 0.85`) και μετά από `:` η προδιαγραφή μορφοποίησης. Το `.2f` σημαίνει δύο δεκαδικά, το `8.2f` δύο δεκαδικά σε συνολικό πλάτος οκτώ θέσεων. Οι προδιαγραφές που θα χρησιμοποιούμε:

Προδιαγραφή	Σημασία	`x = 3.14159` δίνει
`{x:4d}`	ακέραιος σε πλάτος 4 (μόνο για `int`)	`{7:4d}` → `7`
`{x:8.2f}`	δεκαδικός, πλάτος 8, 2 δεκαδικά	`3.14`
`{x:10.3e}`	εκθετική μορφή, πλάτος 10, 3 δεκαδικά	`3.142e+00`
`{x:.4f}`	4 δεκαδικά, όσο πλάτος χρειάζεται	`3.1416`

Το πλάτος είναι *ελάχιστο*: αν ο αριθμός χρειάζεται περισσότερα ψηφία, το πλάτος αγνοείται. Αν το αποτέλεσμα είναι μικρότερο, συμπληρώνεται με κενά από αριστερά, πράγμα που μας επιτρέπει να στοιχίζουμε στήλες.

1.12 Βιβλιοθήκες: τι είναι και πώς τις χρησιμοποιούμε

Ο πυρήνας της γλώσσας περιέχει σχετικά λίγα: αριθμητικές πράξεις, αλφαριθμητικά, `print`, `input`, `abs`, `max`, `min`, `len`, `type`. Ό,τι άλλο χρειαζόμαστε (τριγωνομετρικές συναρτήσεις, τυχαίοι αριθμοί, γραφήματα, πίνακες) βρίσκεται σε **βιβλιοθήκες** (modules). Μια βιβλιοθήκη είναι ένα έτοιμο αρχείο Python με συναρτήσεις και σταθερές, που «φορτώνουμε» στο πρόγραμμά μας με την εντολή `import`.

Η βασική μορφή.

```
import math
print(math.pi)          # η σταθερά π
print(math.sqrt(2))     # τετραγωνική ρίζα
```

Μετά το `import math` κάθε στοιχείο της βιβλιοθήκης είναι προσβάσιμο με το **όνομα της βιβλιοθήκης, τελεία, όνομα του στοιχείου**: `math.pi`, `math.sqrt`, `math.sin`, `math.floor`, `math.ceil`, `math.isqrt`. Η τελεία διαβάζεται «το `pi` της `math`». Αν ξεχάσουμε το `import` ή το πρόθεμα `math.`, παίρνουμε `NameError: name 'pi' is not defined`.

Με συντομογραφία.

```
import numpy as np
import matplotlib.pyplot as plt
```

Το `import … as …` φορτώνει τη βιβλιοθήκη με ένα συντομότερο όνομα που επιλέγουμε εμείς. Γράφουμε `np.linspace`, `plt.plot` αντί για `numpy.linspace`, `matplotlib.pyplot.plot`. Οι συντομογραφίες `np` και `plt` είναι καθιερωμένες, και καλό είναι να τις κρατάμε.

Πολλές βιβλιοθήκες σε μία γραμμή. Το `import math, random` φορτώνει και τις δύο. Συνηθίζεται όμως μία βιβλιοθήκη ανά γραμμή.

Πού γράφουμε τα `import`. Μπορούν να μπουν οπουδήποτε στο αρχείο, αρκεί πριν από την πρώτη χρήση. Η καθιερωμένη πρακτική είναι **όλα μαζί στην αρχή** του αρχείου, ώστε όποιος διαβάζει το πρόγραμμα να βλέπει αμέσως τι χρειάζεται.

Οι βιβλιοθήκες του μαθήματος.

Βιβλιοθήκη	Τι προσφέρει	Πού τη συναντάμε
`math`	`pi`, `sqrt`, `isqrt`, `sin`, `cos`, `floor`, `ceil`	σχεδόν παντού

Βιβλιοθήκη	Τι προσφέρει	Πού τη συναντάμε
`random`	`random`, `uniform`, `normalvariate`, `randint`	προσεγγίσεις του π (Monte Carlo)
`time`	`time()` για χρονομέτρηση	`prime_sqrt.py`
`numpy` (`np`)	πίνακες αριθμών και πράξεις σε όλα τα στοιχεία μαζί	`4_torus.py`, `7_poly.py`
`matplotlib.pyplot` (`plt`)	γραφήματα	σε όλες τις ενότητες

Οι `math`, `random`, `time` ανήκουν στην **πρότυπη βιβλιοθήκη** και υπάρχουν σε κάθε εγκατάσταση Python. Οι `numpy` και `matplotlib` είναι εξωτερικές και εγκαθίστανται χωριστά (στο Anaconda και στο Spyder υπάρχουν ήδη).

1.13 Γραφική έξοδος με `matplotlib`

Το `matplotlib` είναι η βασική βιβλιοθήκη γραφημάτων της Python. Εμείς χρησιμοποιούμε το υποσύστημά της `matplotlib.pyplot`, που προσφέρει απλές εντολές στο στυλ «σχεδίασε αυτό, βάλε τίτλο, δείξε το», και το φορτώνουμε πάντα με την ίδια εντολή:

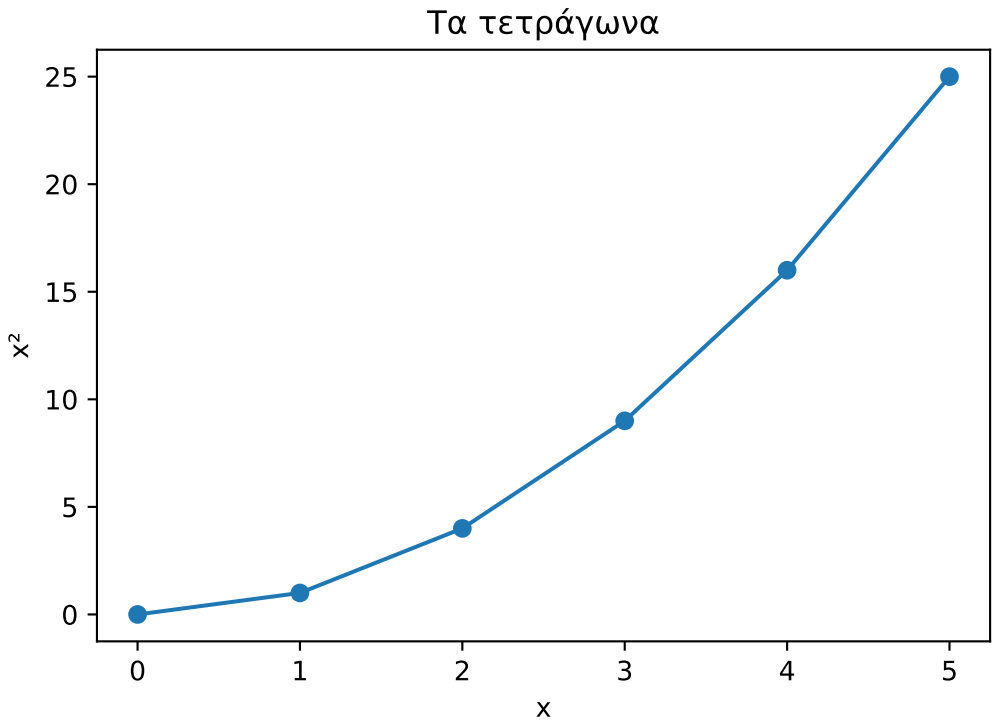
```
import matplotlib.pyplot as plt
```

1.13.1 Το πρώτο γράφημα

```
import matplotlib.pyplot as plt

xs = [0, 1, 2, 3, 4, 5]
ys = [0, 1, 4, 9, 16, 25]

plt.plot(xs, ys, marker='o')
plt.xlabel('x'); plt.ylabel('x²'); plt.title('Τα τετράγωνα')
plt.show()
```



Σχήμα 1: Το πρώτο γράφημα: τα τετράγωνα των 0, 1, ..., 5.

Η βασική ιδέα: ένα γράφημα είναι ένα **σύνολο σημείων** (x_i, y_i). Δίνουμε στην `plt.plot` δύο λίστες ίσου μήκους, τη λίστα των τετμημένων και τη λίστα των τεταγμένων, και αυτή ενώνει τα διαδοχικά σημεία με ευθύγραμμα τμήματα. Αν τα σημεία είναι πυκνά (π.χ. 400 σημεία στο `7_poly.py`), η τεθλασμένη φαίνεται ως ομαλή καμπύλη. Το `marker='o'` σημειώνει επιπλέον κάθε σημείο με έναν κύκλο.

Ο χαρακτήρας `;` (το λατινικό semicolon, που μοιάζει με το ελληνικό ερωτηματικό) επιτρέπει δύο εντολές στην ίδια γραμμή. Γενικά το αποφεύγουμε, εδώ χρησιμεύει για να κρατήσει το παράδειγμα σύντομο.

1.13.2 Πώς δουλεύει: ζωγραφίζουμε σε ένα «φύλλο» και στο τέλος το δείχνουμε

Οι εντολές του `pyplot` **δεν** ανοίγουν αμέσως παράθυρο. Κάθε `plt.plot`, `plt.title` κ.λπ. προσθέτει κάτι σε ένα κρυφό «φύλλο σχεδίασης» (figure). Το παράθυρο ανοίγει μόνο με την `plt.show()`, η οποία και **παγώνει την εκτέλεση του προγράμματος** μέχρι να κλείσουμε το παράθυρο. Αυτό έχει δύο συνέπειες:

- Όλα όσα καλέσουμε πριν την `show` εμφανίζονται **στο ίδιο γράφημα**. Έτσι στο `7_poly.py` η καμπύλη, οι άξονες και τα κόκκινα σημεία των ακροτάτων ζωγραφίζονται με τέσσερις ξεχωριστές εντολές και εμφανίζονται μαζί.
- Η `plt.show()` πρέπει να είναι η **τελευταία** εντολή του γραφήματος. Αν τη βάλουμε πριν από ένα `plt.plot`, το δεύτερο σχέδιο δεν θα φανεί.

Για δεύτερο, χωριστό παράθυρο καλούμε `plt.figure()` πριν αρχίσουμε να σχεδιάζουμε σε αυτό.

1.13.3 Οι εντολές που χρησιμοποιούμε στο μάθημα

Εντολή	Τι κάνει	Πού τη συναντάμε
`plt.plot(x, y)`	γραμμή μέσα από τα σημεία (x_i, y_i)	παντού
`plt.plot(y)`	το ίδιο, με τετμημένες `0, 1, 2, …`	`5_primeall.py`, `6_primeex2.py`
`plt.plot(x, y, '.')`	μόνο τελείες, χωρίς γραμμή	`rand.py`
`plt.plot(x, y, 'ro')`	κόκκινοι κύκλοι, χωρίς γραμμή	`7_poly.py`
`plt.scatter(x, y)`	σημεία (όπως το `'.'`, με περισσότερες επιλογές)	
`plt.bar(x, h)`	ραβδόγραμμα με ύψη `h`	
`plt.xlabel('…')`, `plt.ylabel('…')`	ετικέτες αξόνων	`7_poly.py`
`plt.title('…')`	τίτλος	`7_poly.py`
`plt.legend()`	υπόμνημα από τα `label=` των `plot`	`7_poly.py`
`plt.grid(True)`	πλέγμα στο φόντο	`7_poly.py`
`plt.axhline(0)`, `plt.axvline(0)`	οριζόντια / κατακόρυφη ευθεία (οι άξονες)	`7_poly.py`
`plt.savefig('όνομα.png')`	αποθήκευση σε αρχείο εικόνας (πριν τη `show`)	
`plt.show()`	εμφάνιση του παραθύρου	παντού

1.13.4 Μορφή γραμμής και σημείων

Το τρίτο όρισμα της `plot` είναι ένα σύντομο **αλφαριθμητικό μορφοποίησης** που συνδυάζει χρώμα, σύμβολο σημείου (marker) και είδος γραμμής, με οποιαδήποτε σειρά:

Χρώμα		Σύμβολο σημείου		Γραμμή	
`r`	κόκκινο	`o`	κύκλος	`-`	συνεχής
`b`	μπλε	`.`	τελεία	`--`	διακεκομμένη
`g`	πράσινο	`s`	τετράγωνο	`:`	με τελείες
`k`	μαύρο	`^`	τρίγωνο		

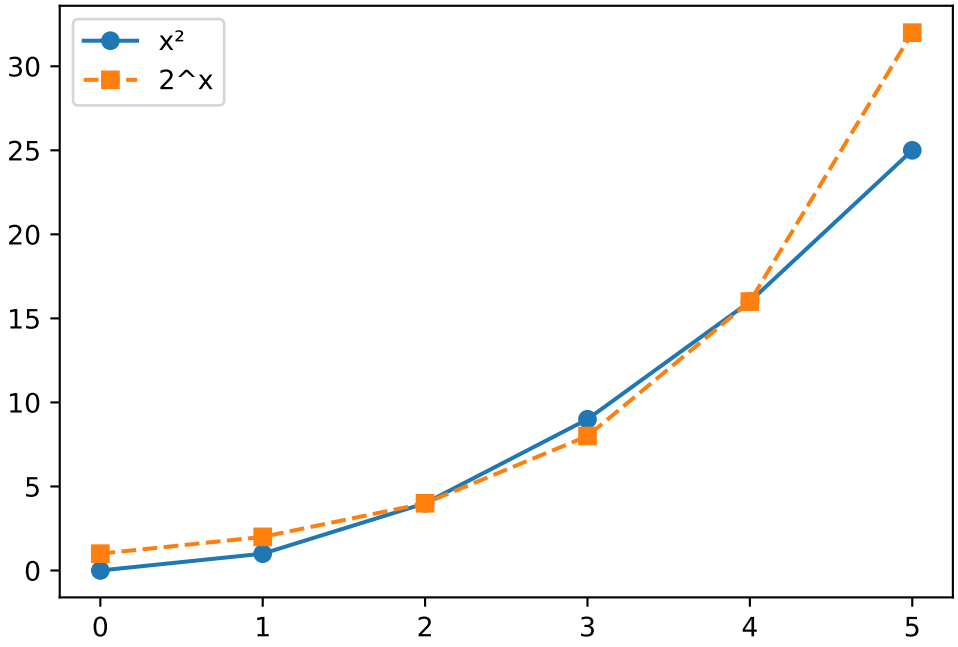
Παραδείγματα: `'ro'` κόκκινοι κύκλοι χωρίς γραμμή, `'b-'` μπλε συνεχής γραμμή, `'g--'` πράσινη διακεκομμένη, `'k.'` μαύρες τελείες. Αν δώσουμε μόνο σύμβολο σημείου (`'o'`, `'.'`) δεν σχεδιάζεται γραμμή. Τα ίδια ρυθμίζονται και με ονομαστικά ορίσματα, όπως `color='black'`, `linewidth=0.8`, `marker='o'`, που είναι πιο ευανάγνωστα. Το `label='q(x)'` δίνει το όνομα που θα εμφανίσει η `legend`.

1.13.5 Πολλές καμπύλες στο ίδιο γράφημα

Κάθε κλήση της `plot` πριν την `show` προσθέτει μία ακόμη καμπύλη, με αυτόματα διαφορετικό χρώμα:

```
import matplotlib.pyplot as plt

xs = [0, 1, 2, 3, 4, 5]
plt.plot(xs, [x**2 for x in xs], 'o-', label='x²')
plt.plot(xs, [2**x for x in xs], 's--', label='2^x')
plt.legend()
plt.show()
```



Σχήμα 2: Δύο καμπύλες στο ίδιο γράφημα, με υπόμνημα.

(Το `[x**2 for x in xs]` φτιάχνει τη λίστα των τετραγώνων σε μία γραμμή. Θα το εξηγήσουμε με τις λίστες στην επανάληψη `for`, προς το παρόν διαβάστε το ως «η λίστα των `x**2` για κάθε `x` στο `xs`».)

1.13.6 Από πού έρχονται τα δεδομένα

Η `plot` θέλει λίστες αριθμών. Στο μάθημα τις φτιάχνουμε με τρεις τρόπους:

- **Με βρόχο και `append`**, ένα σημείο τη φορά: έτσι γίνεται στο `rand.py` (τυχαία σημεία) και στο `prime_vector.py` (οι πρώτοι αριθμοί).
- **Με `list(range(…))`** για τις τετμημένες `1, 2, …, n`, όπως στο `prime_vector.py`.
- **Με το `numpy`**: η `np.linspace(a, b, n)` δίνει `n` ισαπέχοντα σημεία στο $[a, b]$ και οι πράξεις όπως `a*x**3 + b*x**2 + c*x + d` εφαρμόζονται σε όλα τα σημεία μαζί. Έτσι σχεδιάζεται η καμπύλη στο `7_poly.py` με δύο γραμμές κώδικα. Το `4_torus.py` πάει ένα βήμα παραπέρα και σχεδιάζει **τρισδιάστατη** επιφάνεια με `plot_surface`, πάλι από πίνακες του `numpy`.

Το `matplotlib` θα το χρησιμοποιούμε σε όλο το μάθημα: τόρος, κυβικό πολυώνυμο, πρώτοι αριθμοί, νέφος τυχαίων σημείων.

1.14 Σύνοψη των πρώτων βημάτων

- Έννοια αλγορίθμου, τέσσερις μορφές περιγραφής, γραμμική αναζήτηση.
- Μεταβλητές, βασικοί τύποι (`int`, `float`, `str`), δυναμικοί τύποι.
- Μεταβλητές ως **αναφορές** (`id`, `is`), πολλαπλή ανάθεση, ανταλλαγή τιμών (swap).
- `input` και `print`, μετατροπές τύπου, `sep` και `end`.
- Μορφοποίηση εξόδου με f-strings.
- Βιβλιοθήκες: `import`, `import … as …`, το πρόθεμα `βιβλιοθήκη.όνομα`.
- Γραφήματα με `matplotlib`: `plot`, `scatter`, `bar`, αλφαριθμητικά μορφοποίησης (`'ro'`), `xlabel`/`title`/`legend`, `show`.

2 Ανάθεση μεταβλητών, είσοδος και έξοδος

Στόχος αυτής της ενότητας είναι να γράψουμε προγράμματα που:

1. **διαβάζουν** αριθμούς ή κείμενο από το πληκτρολόγιο,
2. τους **αποθηκεύουν** σε μεταβλητές,
3. **υπολογίζουν** κάτι με αριθμητικές πράξεις και συναρτήσεις της βιβλιοθήκης `math`,
4. **τυπώνουν** το αποτέλεσμα μορφοποιημένο.

2.1 Το πρώτο πρόγραμμα: `input`, `int`, `float`, `print` και f-strings

Πρόγραμμα 1 `1_prpr.py`

```
# Τὸ πρῶτο μου πρόγραμμα
# μορφοποιημένοι συμβολισμοί
t = int(input('Δῶστε τὴν θερμοκρασία: '))
# t = 1/t
p = float(input('Δῶστε τὴν Πίεση: '))
# print(f'θερμοκρασία = {t:12.9e}')
# print(f'θερμοκρασία = {t:12.4f} \n Πίεση = {p:10.3e}')
print(f'θερμοκρασία = {t:4d}  Πίεση = {p:8.2f}')
```

Σχόλια. Ό,τι ακολουθεί το `#` μέχρι το τέλος της γραμμής αγνοείται από την Python. Χρησιμοποιούμε σχόλια για να εξηγούμε τι κάνει ο κώδικας, αλλά και για να «απενεργοποιούμε» προσωρινά γραμμές (όπως οι εναλλακτικές `print` παραπάνω) χωρίς να τις σβήσουμε.

Ανάθεση (ή εκχώρηση) τιμής. Η εντολή `t = …` **δεν** είναι εξίσωση. Σημαίνει: υπολόγισε το δεξί μέλος και δώσε στο όνομα `t` την τιμή που προέκυψε. Γι' αυτό και η γραμμή `t = 1/t` (σε σχόλιο) έχει νόημα: υπολογίζει το `1/t` με την *παλιά* τιμή του `t` και την αποθηκεύει ως *νέα* τιμή του.

Είσοδος. Η `input('μήνυμα')` τυπώνει το μήνυμα, περιμένει να πατήσουμε Enter και επιστρέφει ό,τι πληκτρολογήσαμε **ως κείμενο** (τύπος `str`). Για να το χρησιμοποιήσουμε ως αριθμό το μετατρέπουμε:

Συνάρτηση	Τι επιστρέφει	Παράδειγμα
`int(x)`	ακέραιο (`int`)	`int('42')` → `42`, `int(3.9)` → `3`
`float(x)`	πραγματικό (`float`)	`float('3.14')` → `3.14`
`str(x)`	κείμενο (`str`)	`str(42)` → `'42'`

Αν γράψουμε `int(input(...))` και ο χρήστης δώσει `2.5`, το πρόγραμμα σταματά με σφάλμα `ValueError`: το κείμενο `'2.5'` δεν είναι ακέραιος.

Έξοδος με f-strings. Ένα αλφαριθμητικό που αρχίζει με `f` πριν τα εισαγωγικά (`f'...'`) είναι *μορφοποιημένο αλφαριθμητικό* (f-string). Μέσα σε άγκιστρα `{}` βάζουμε οποιαδήποτε έκφραση και η τιμή της αντικαθίσταται στο κείμενο. Μετά από `:` δίνουμε **προδιαγραφή μορφοποίησης**, όπως στον πίνακα της ενότητας «Μορφοποίηση και f-strings». Εδώ χρησιμοποιούνται οι `4d` (ακέραιος σε πλάτος 4) και `8.2f` (δεκαδικός σε πλάτος 8 με 2 δεκαδικά).

Ειδικοί χαρακτήρες. Μέσα σε ένα αλφαριθμητικό το `\n` σημαίνει «αλλαγή γραμμής» και το `\t` «στηλοθέτης» (tab).

2.2 Βιβλιοθήκη `math`, δυνάμεις και προτεραιότητα πράξεων

Πρόγραμμα 2 `2_SurfAreaFirst.py`

```
import math

r = float(input('Δῶσε τὴν ἀκτίνα r: '))
A = math.pi * r**2
print(f'Ὁ κύκλος μὲ ἀκτίνα r = {r:20.12f} ἔχει ἐμβαδὸν A = {A:6.4e}')
```

`import math`. Η σταθερά π δεν είναι ενσωματωμένη στη γλώσσα, βρίσκεται στη βιβλιοθήκη `math` (βλ. την ενότητα για τις βιβλιοθήκες παραπάνω). Μετά το `import math` τη χρησιμοποιούμε ως `math.pi`.

Αριθμητικοί τελεστές.

Τελεστής	Πράξη	Παράδειγμα
`+ - *`	πρόσθεση, αφαίρεση, πολλαπλασιασμός	`2*3` → `6`
`/`	διαίρεση (πάντα `float`)	`7/2` → `3.5`
`//`	ακέραιο πηλίκο	`7//2` → `3`
`%`	υπόλοιπο διαίρεσης	`7%2` → `1`
`**`	δύναμη	`2**10` → `1024`

Η προτεραιότητα είναι η γνωστή από τα μαθηματικά: πρώτα `**`, μετά `* / // %`, μετά `+ -`. Έτσι το `math.pi * r**2` είναι το πr^2 και όχι το $(\pi r)^2$. Όταν έχουμε αμφιβολία, βάζουμε παρενθέσεις.

2.3 Μονάδες μέτρησης και ενδιάμεσες μεταβλητές

Πρόγραμμα 3 `3_surf_inc.py`

```
# Αὔξηση ἐμβαδοῦ ἐπιφάνειας
# Διάβασε καὶ ἐμφάνισε τὰ δεδομένα εἰσόδου...
import math


r = float(input('Παρακαλῶ, δῶστε τὴν ἀκτίνα (χιλιόμετρα):'))
delta_r = float(input('Παρακαλῶ, δῶστε τὴν αὔξηση σὲ (χιλιοστά):'))

print(f'Ἀκτίνα σφαίρας = {r:12.7f} χιλιόμετρα')
print(f'Αὔξηση Ἀκτίνας = {delta_r:12.7f} χιλιοστά\n')
print('Αὔξηση ἐμβαδοῦ ἐπιφάνειας:')


dr = delta_r / 10**6


# Μέθοδος 1
delta_A1 = (4*math.pi*(r + dr)**2 - 4*math.pi*r**2) * 10**6
print(f'\n\tΜέθοδος 1: {delta_A1:15.7f} τετρ/κά μέτρα')
```

Το πρόγραμμα υπολογίζει πόσο αυξάνεται η επιφάνεια $4\pi r^2$ μιας σφαίρας όταν η ακτίνα αυξηθεί κατά `delta_r`. Τα νέα στοιχεία:

- **Ονόματα μεταβλητών.** Επιτρέπονται γράμματα, ψηφία και η κάτω παύλα, αλλά όχι ψηφίο στην αρχή. Το `delta_r` είναι πιο ευανάγνωστο από το `d`. Η Python κάνει διάκριση πεζών-κεφαλαίων: τα `r` και `R` είναι διαφορετικές μεταβλητές.
- **Μετατροπή μονάδων σε ξεχωριστή μεταβλητή.** Το `dr = delta_r / 10**6` μετατρέπει τα χιλιοστά σε χιλιόμετρα. Κρατώντας τη μετατροπή σε δική της μεταβλητή, ο τύπος που ακολουθεί γίνεται πιο καθαρός, και αν κάνουμε λάθος στις μονάδες το εντοπίζουμε ευκολότερα.

- **Καλή πρακτική: τύπωσε ό,τι διάβασες.** Οι δύο `print` αμέσως μετά την είσοδο επιβεβαιώνουν ότι το πρόγραμμα κατάλαβε αυτό που δώσαμε.
- **Αριθμητική ακρίβεια.** Η «Μέθοδος 1» αφαιρεί δύο σχεδόν ίσους μεγάλους αριθμούς, οπότε στην αριθμητική κινητής υποδιαστολής χάνονται σημαντικά ψηφία (το φαινόμενο λέγεται **καταστροφική απαλοιφή**). Η αλγεβρικά ισοδύναμη μορφή $4\pi\,(2r\,dr + dr^2)$ είναι ακριβέστερη. Αυτό είναι το πρώτο παράδειγμα ότι *μαθηματικά ίσοι* τύποι δεν είναι *υπολογιστικά* ίσοι.

2.4 Πολλές γραμμές εξόδου, εισαγωγικά μέσα σε κείμενο και ένα πρώτο γράφημα

Πρόγραμμα 4 `4_torus.py`

```
# Ὑπολογισμὸς ἐμβαδοῦ καὶ ὄγκου τόρου!
import math
import numpy as np
import matplotlib.pyplot as plt

R = int(input('Παρακαλῶ, δῶστε τὴν μεγάλη ἀκτίνα (ἀκέραιος): '))
print(f'R = {R}')
r = float(input('Παρακαλῶ, δῶστε τὴν μικρὴ ἀκτίνα: '))
print(f'r = {r}')

A = 4*math.pi**2*R*r
print(f'A = {A}')
V = 2*math.pi**2*R*r**2
print(f'V = {V}')

z = 'θεώρημα τοῦ Πάππου'
print(f'{z}:\n Ὁ τόρος μὲ "μεγάλη ἀκτίνα" R= {R:12d} καὶ \'μικρὴ ἀκτίνα\' r = {r:10.4e}\n'
      f'ἔχει ἐπιφάνεια A = {A:20.8f} καὶ ὄγκο V = {V:20.7f}')

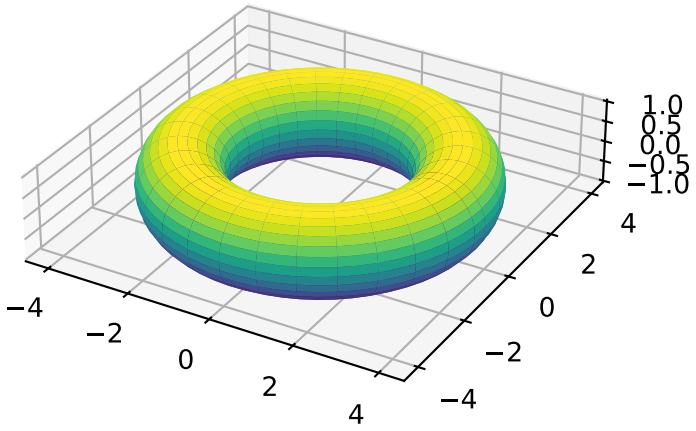






# Σχεδίαση τοῦ τόρου!
u, v = np.meshgrid(np.linspace(0, 2*np.pi, 60), np.linspace(0, 2*np.pi, 30))
X = (R + r*np.cos(v))*np.cos(u)
Y = (R + r*np.cos(v))*np.sin(u)
Z = r*np.sin(v)

ax = plt.figure().add_subplot(projection='3d')
ax.plot_surface(X, Y, Z, cmap='viridis')
ax.set_aspect('equal')
#ax.set_zticks([-r, 0, r])
#ax.set_title(f'Τόρος: R = {R}, r = {r}')
plt.show()
```



Σχήμα 3: Ο τόρος για R = 3 και r = 1.

Το θεώρημα του Πάππου δίνει για τον τόρο με ακτίνες R και r: επιφάνεια $A = 4\pi^2 Rr$ και όγκο $V = 2\pi^2 Rr^2$.

- **Μεταβλητές κειμένου.** Το `z = 'Θεώρημα τοῦ Πάππου'` αποθηκεύει ένα αλφαριθμητικό. Μέσα στο f-string το `{z}` αντικαθίσταται από το κείμενο.
- **Εισαγωγικά μέσα σε εισαγωγικά.** Ένα αλφαριθμητικό ορίζεται είτε με `'...'` είτε με `"..."`. Μέσα σε `'...'` μπορούμε να βάλουμε ελεύθερα `"`, ενώ για ένα `'` πρέπει να βάλουμε μπροστά του τον **χαρακτήρα διαφυγής** `\`, δηλαδή να γράψουμε `\'`.
- **Μία εντολή σε πολλές γραμμές.** Δύο αλφαριθμητικά γραμμένα το ένα δίπλα στο άλλο (εδώ, σε διαδοχικές γραμμές μέσα στην παρένθεση του `print`) ενώνονται αυτόματα σε ένα. Η Python επιτρέπει να χωρίζουμε μια εντολή σε περισσότερες γραμμές, αρκεί να βρισκόμαστε μέσα σε ανοιχτή παρένθεση.
- **Δύο νέες βιβλιοθήκες.** Τα `import numpy as np` και `import matplotlib.pyplot as plt` φορτώνουν το `numpy` και το `matplotlib` με τις καθιερωμένες συντομογραφίες τους (βλ. την ενότητα για τις βιβλιοθήκες).
- **Γράφημα.** Το τελευταίο μέρος σχεδιάζει τον τόρο ως παραμετρική επιφάνεια $(u, v) \mapsto ((R + r\cos v)\cos u,\ (R + r\cos v)\sin u,\ r\sin v)$. Η `np.linspace(0, 2*np.pi, 60)` δίνει 60 ισαπέχοντα σημεία στο $[0, 2\pi]$, η `np.meshgrid` φτιάχνει το πλέγμα των ζευγών (u, v), και οι `np.cos`, `np.sin` εφαρμόζονται σε ολόκληρο τον πίνακα με μία κλήση. Τα πλήρη εργαλεία του `numpy` θα τα δούμε αργότερα, εδώ αρκεί να δούμε ότι το γράφημα είναι λίγες γραμμές κώδικα.

2.5 Μέγιστο και ελάχιστο

Πρόγραμμα 5 `5_minmax.py`

```
a = float(input('Παρακαλῶ, δῶστε τὸ a: '))
print(f'a = {a}')
b = float(input('Παρακαλῶ, δῶστε τὸ b: '))
print(f'b = {b}')

max = ((a + b) + abs(a - b)) / 2
min = ((a + b) - abs(a - b)) / 2
print(f'τὸ μικρότερο ἀπὸ τὰ a καὶ b εἶναι τό:{int(min):3d}\n'
      f'καὶ τὸ μεγαλύτερο ἀπὸ τὰ a καὶ b εἶναι τό:{int(max):3d}')
```

Οι ταυτότητες

$$\max(a,b) = \frac{(a+b)+|a-b|}{2}, \qquad \min(a,b) = \frac{(a+b)-|a-b|}{2}$$

υπολογίζουν μέγιστο και ελάχιστο **χωρίς διακλάδωση**. Η `abs(x)` είναι η απόλυτη τιμή και είναι ενσωματωμένη συνάρτηση (δεν χρειάζεται `import`).

> ⚠ Μην χρησιμοποιείτε `max`, `min` ως ονόματα μεταβλητών
>
> Η Python έχει ήδη ενσωματωμένες συναρτήσεις `max(...)` και `min(...)`. Γράφοντας `max = ...` **επισκιάζουμε** (shadowing) τη συνάρτηση: το όνομα `max` δείχνει πλέον σε έναν αριθμό. Από εκείνο το σημείο και μετά, μια κλήση `max(x, y)` στο ίδιο πρόγραμμα θα δώσει σφάλμα `TypeError: 'float' object is not callable`. Προτιμήστε ονόματα όπως `mx`, `mn`, `largest`.

`int()` και μορφοποίηση `d`. Η προδιαγραφή `:3d` δουλεύει μόνο για ακεραίους, γι' αυτό πρώτα γίνεται `int(min)`. Προσοχή: η `int` **αποκόπτει** το δεκαδικό μέρος (`int(-2.7)` → `-2`), δεν στρογγυλοποιεί. Για στρογγυλοποίηση υπάρχει η `round`.

2.6 Αλφαριθμητικά: ένωση και δείκτες

Πρόγραμμα 6 `6_string.py`

```
name = str(input('Παρακαλῶ, δῶστε τὸ ὄνομά σας: '))
surname = str(input('Παρακαλῶ, δῶστε τὸ ἐπώνυμό σας: '))
full_name = name+" "+surname
print(full_name[4])
```

- Το `str(input(...))` είναι περιττό (η `input` επιστρέφει ήδη `str`), αλλά δείχνει τη συμμετρία με τα `int(...)`, `float(...)`.
- **Συνένωση.** Ο τελεστής `+` ανάμεσα σε δύο αλφαριθμητικά τα **ενώνει**. Το `" "` είναι ένα κενό. Άρα το `name + " " + surname` είναι το πλήρες όνομα με ένα κενό ανάμεσα.
- **Δείκτες.** Το `full_name[4]` είναι ο **πέμπτος** χαρακτήρας, γιατί η αρίθμηση αρχίζει από το `0`. Αν το αλφαριθμητικό έχει λιγότερους από 5 χαρακτήρες παίρνουμε `IndexError`. Το ίδιο σύστημα δεικτών θα το ξαναδούμε στις λίστες.

3 Συνθήκες: η εντολή `if`

Ως τώρα κάθε εντολή εκτελούνταν ακριβώς μία φορά, με τη σειρά. Η εντολή `if` επιτρέπει στο πρόγραμμα να **επιλέγει** ποιες εντολές θα εκτελέσει, ανάλογα με το αν μια συνθήκη είναι αληθής.

3.1 Σύγκριση: μέγιστο και ελάχιστο χωρίς `if`

Πρόγραμμα 7 `1_minmax.py`

```
a = float(input('Παρακαλῶ, δῶστε τὸ a: '))
print(f'a = {a}')
b = float(input('Παρακαλῶ, δῶστε τὸ b: '))
print(f'b = {b}')

max = ((a + b) + abs(a - b)) / 2
min = ((a + b) - abs(a - b)) / 2
print(f'τὸ μικρότερο ἀπὸ τὰ a καὶ b εἶναι τό:{int(min):3d}\n'
      f' καὶ τὸ μεγαλύτερο ἀπὸ τὰ a καὶ b εἶναι τό:{int(max):3d}')
```

Το πρόγραμμα αυτό είναι ίδιο με το `5_minmax.py` και χρησιμεύει ως σημείο σύγκρισης: εδώ το «ποιος είναι μεγαλύτερος» κρύβεται στην `abs`. Στα επόμενα προγράμματα η απόφαση γίνεται ρητή.

Πρόγραμμα 8 `2_first.py`

```
a = float(input('Παρακαλῶ δῶστε τὸ a: '))
b = float(input('Εὐχαριστῶ! Παρακαλῶ δῶστε τὸ b: '))
if a > b:
    print(f'Τὸ a={a:6.2f} εἶναι μεγαλύτερο τοῦ b={b:6.2f}')
else:
    if a < b:
        print(f'Τὸ a={a:6.2f} εἶναι μικρότερο τοῦ b={b:6.2f}')
    else:
        print(f'Τὸ a={a:6.2f} εἶναι ἴσο τοῦ b={b:6.2f}')
```

3.2 `if … else` και εμφωλευμένα `if`

Σύνταξη.

```
if συνθήκη:
    εντολές που εκτελούνται αν η συνθήκη είναι αληθής
else:
    εντολές που εκτελούνται αν είναι ψευδής
```

Τρία πράγματα είναι υποχρεωτικά: η **άνω-κάτω τελεία** στο τέλος της γραμμής `if`/`else`, η **εσοχή** (4 κενά) των εντολών που ανήκουν στο μπλοκ, και το ότι το `else` στοιχίζεται ακριβώς κάτω από το `if` του. Στην Python η εσοχή **δεν είναι διακοσμητική**: αυτή ορίζει ποιες εντολές ανήκουν σε ποιο μπλοκ.

Τελεστές σύγκρισης. Μια συνθήκη είναι μια έκφραση με τιμή `True` ή `False` (τύπος `bool`).

Τελεστής	Σημασία
`<`, `<=`, `>`, `>=`	μικρότερο, μικρότερο ή ίσο, κ.λπ.
`==`	ίσο (διπλό ίσον: το απλό `=` είναι ανάθεση)
`!=`	διάφορο

Εμφώλευση. Μέσα στο μπλοκ του `else` μπορεί να υπάρχει νέο `if … else`, με μία επιπλέον εσοχή. Έτσι διακρίνουμε τρεις περιπτώσεις ($a > b$, $a < b$, $a = b$). Η εμφώλευση όμως βαθαίνει γρήγορα και γίνεται δυσανάγνωστη, γι᾽ αυτό υπάρχει το `elif`.

3.3 `elif`: πολλές περιπτώσεις στο ίδιο επίπεδο

Πρόγραμμα 9 `3_comp.py`

```
# Ἡ ἐντολὴ if
a = float(input('Παρακαλῶ δῶστε τὸ a: '))
b = float(input('Εὐχαριστῶ! Παρακαλῶ δῶστε τὸ b: '))
if a > b:
    print(f'Τὸ a={a:6.2f} εἶναι μεγαλύτερο τοῦ b={b:6.2f}')
elif a < b:
    print(f'Τὸ a={a:6.2f} εἶναι μικρότερο τοῦ b={b:6.2f}')
else:
    print(f'Τὸ a={a:6.2f} εἶναι ἴσο τοῦ b={b:6.2f}')
```

Το `elif` («else if») ελέγχει μια νέα συνθήκη **μόνο αν** όλες οι προηγούμενες ήταν ψευδείς. Σε μια αλυσίδα `if / elif / … / else` εκτελείται **ακριβώς ένα** μπλοκ: το πρώτο του οποίου η συνθήκη είναι αληθής, ή το `else` αν καμία δεν είναι. Το `else` είναι προαιρετικό. Το πρόγραμμα είναι λογικά ίδιο με το προηγούμενο, αλλά «επίπεδο» και ευανάγνωστο.

3.4 Ένα πραγματικό πρόβλημα: ελάχιστο του $f(x) = x^2 + bx + c$ στο $[L, R]$

Το ελάχιστο της παραβολής στο διάστημα $[L, R]$ βρίσκεται στην κορυφή $x_c = -b/2$ αν αυτή ανήκει στο διάστημα, αλλιώς στο άκρο που είναι πλησιέστερα στην κορυφή. Πρώτα πρέπει να ελέγξουμε ότι η είσοδος είναι έγκυρη ($L \le R$).

3.4.1 Εκδοχή 1: εμφωλευμένα `if`

Λογικοί τελεστές. Η συνθήκη `L <= xc and xc <= R` είναι αληθής μόνο αν **και οι δύο** συγκρίσεις είναι αληθείς.

Πρόγραμμα 10 `4_ifthen.py`

```
b = float(input('Δῶσε τὸ b: '))
c = float(input('Δῶσε τὸ c: '))
L = float(input('Δῶσε τὸ L: '))
R = float(input('Δῶσε τὸ R: '))
if L > R:
    print('Ἡ τιμὴ τοῦ L πρέπει νὰ μὴν ὑπερβαίνει τὴν τιμὴ τοῦ R')
else:
    xc = -b / 2
    if L <= xc and xc <= R:
        fc = c - (b/2)**2
        print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {fc:6.3f}')
    else:
        if xc > R:
            fr = R**2 + b*R + c
            print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {fr:6.3f}')
        else:
            fl = L**2 + b*L + c
            print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {fl:6.3f}')
```

Η Python επιτρέπει και τη μαθηματική γραφή `L <= xc <= R`, που σημαίνει ακριβώς το ίδιο.

Έλεγχος εγκυρότητας εισόδου. Το πρώτο `if` αντιμετωπίζει την περίπτωση λάθους και τυπώνει μήνυμα. Όλος ο υπόλοιπος υπολογισμός μπαίνει στο `else`, ώστε να μην εκτελεστεί με άκυρα δεδομένα.

3.4.2 Εκδοχή 2: αλυσίδα `if / elif / else`

Πρόγραμμα 11 `5_ifthen_cases.py`

```
# Πολυώνυμο f(x)=x^2+bx+c ὁρισμένο στὸ διάστημα [L,R]
# Εὕρεση ἐλάχιστης τιμῆς τῆς συνάρτησης f(x)
b = float(input('Παρακαλῶ, δῶστε τὸ b: '))
c = float(input('Παρακαλῶ, δῶστε τὸ c: '))
L = float(input('Παρακαλῶ, δῶστε τὸ L: '))
R = float(input('Παρακαλῶ, δῶστε τὸ R: '))
xc = -b / 2
if L > R:
    print('Ἡ τιμὴ τοῦ L πρέπει νὰ μὴν ὑπερβαίνει τὴν τιμὴ τοῦ R')
elif L <= xc and xc <= R:
    fc = c - (b/2)**2
    print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {fc:6.3f}')
elif xc > R:
    fr = R**2 + b*R + c
    print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {fr:6.3f}')
else:
    fl = L**2 + b*L + c
    print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {fl:6.3f}')
```

Οι τέσσερις περιπτώσεις (άκυρη είσοδος, κορυφή μέσα, κορυφή δεξιά, κορυφή αριστερά) είναι τώρα τέσσερα μπλοκ στο ίδιο επίπεδο. Παρατηρήστε ότι η **σειρά** των ελέγχων έχει σημασία: ο έλεγχος `xc > R` γίνεται μόνο αφού αποκλειστεί το `L <= xc <= R`, οπότε στο τελικό `else` ξέρουμε με βεβαιότητα ότι `xc < L`.

3.4.3 Εκδοχή 3: ένα μόνο `print` και αριθμητική με `bool`

Δύο ιδέες:

- **Μία μεταβλητή για το αποτέλεσμα, ένα `print` στο τέλος.** Αντί να επαναλαμβάνουμε το ίδιο μήνυμα σε τρία σημεία, κάθε κλάδος απλώς δίνει τιμή στο `t`. Ο κώδικας είναι μικρότερος και αν

Πρόγραμμα 12 `6_max1.py`

```
# Πολυώνυμο f(x)=x^2+bx+c όρισμένο στὸ διάστημα [L,R]
# Εὕρεση ἐλάχιστης τιμῆς τῆς συνάρτησης f(x)
b = float(input('Παρακαλῶ, δῶστε τὸ b: '))
c = float(input('Παρακαλῶ, δῶστε τὸ c: '))
L = float(input('Παρακαλῶ, δῶστε τὸ L: '))
R = float(input('Παρακαλῶ, δῶστε τὸ R: '))
x0 = -b / 2

if L > R:
    print('Ἡ τιμὴ τοῦ L πρέπει νὰ μὴν ὑπερβαίνει τὴν τιμὴ τοῦ R')
elif L <= x0 and x0 <= R:
    t = c - (b/2)**2
else:
    t = (x0 > R)*(R**2 + b*R + c) + (x0 < L)*(L**2 + b*L + c)

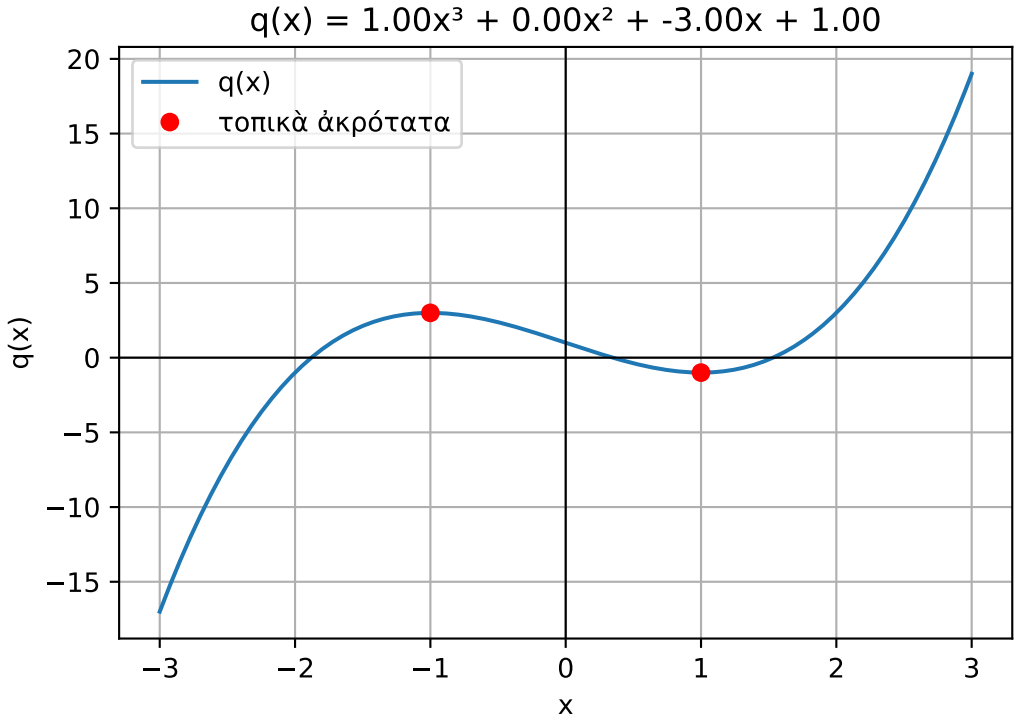
print(f'Ἡ μικρότερη τιμὴ τῆς συνάρτησης f στὸ διάστημα [L,R] εἶναι ἡ: {t:6.3f}')
```

θέλουμε να αλλάξουμε το μήνυμα το αλλάζουμε σε ένα σημείο.

- **Οι λογικές τιμές είναι αριθμοί.** Στην Python το `True` συμπεριφέρεται ως `1` και το `False` ως `0` μέσα σε αριθμητικές πράξεις. Στην έκφραση `(x0 > R)*(…) + (x0 < L)*(…)` ακριβώς μία από τις δύο συγκρίσεις είναι αληθής, οπότε ο ένας όρος διατηρείται και ο άλλος πολλαπλασιάζεται με `0`. Είναι ένας τρόπος να γράψουμε δύο περιπτώσεις σε μία γραμμή. Θα τον ξαναδούμε στη μέτρηση επιτυχιών με τυχαίους αριθμούς.

> Σφάλμα που αξίζει να δείτε
>
> Αν δώσετε `L > R`, το πρόγραμμα τυπώνει το μήνυμα λάθους αλλά η μεταβλητή `t` δεν παίρνει ποτέ τιμή. Το τελικό `print` τότε σταματά με `NameError: name 't' is not defined`. Είναι τυπικό παράδειγμα μεταβλητής που ορίζεται μόνο σε κάποιους κλάδους ενός `if`. Πώς θα το διορθώνατε;

3.5 Ανάλυση κυβικού πολυωνύμου: `if` μέσα σε `else`, `math.sqrt` και γράφημα



Σχήμα 4: Η γραφική παράσταση για a = 1, b = 0, c = -3, d = 1: δύο τοπικά ακρότατα με ετερόσημες τιμές, άρα το πολυώνυμο είναι απλό.

Το μαθηματικό σκεπτικό. Το $q(x) = ax^3 + bx^2 + cx + d$ έχει παράγωγο $q'(x) = 3ax^2 + 2bx + c$. Αν η διακρίνουσα της q' είναι ≤ 0, η q' δεν αλλάζει πρόσημο και το q είναι μονότονο (άρα έχει μία μόνο πραγματική ρίζα). Αλλιώς το q έχει δύο τοπικά ακρότατα στις ρίζες r_1, r_2 της q', και έχει τρεις διακριτές πραγματικές ρίζες ακριβώς όταν οι τιμές $q(r_1)$, $q(r_2)$ είναι ετερόσημες.

Στοιχεία της γλώσσας.

- **Συνθήκη που προστατεύει έναν υπολογισμό.** Η `math.sqrt` σταματά με σφάλμα αν της δώσουμε αρνητικό αριθμό. Το `if discriminant <= 0` εξασφαλίζει ότι η ρίζα υπολογίζεται μόνο όταν έχει νόημα. Γενικά: *πρώτα ελέγχουμε, μετά υπολογίζουμε.*
- **Εμφωλευμένο `if` μέσα σε `else`.** Ο έλεγχος «είναι απλό;» έχει νόημα μόνο στον κλάδο όπου υπάρχουν δύο ακρότατα, γι' αυτό βρίσκεται μέσα στο `else`.
- **Εισαγωγικά.** Το μήνυμα για την παράγωγο περιέχει τον χαρακτήρα `'` (στο `q'(x)`), οπότε το αλφαριθμητικό ορίζεται με διπλά εισαγωγικά `"..."`.
- **`import` στη μέση του αρχείου.** Όπως είπαμε στην ενότητα για τις βιβλιοθήκες, το `import` μπορεί να γίνει οπουδήποτε πριν την πρώτη χρήση. Εδώ τα `numpy`/`matplotlib` φορτώνονται πριν τη σχεδίαση για να διαχωριστεί οπτικά ο «υπολογισμός» από τη «γραφική παράσταση».
- **Η ενσωματωμένη `max`** χρησιμοποιείται σωστά εδώ (`max(2, 1.5*abs(r1 - r2))`), σε αντίθεση με το `5_minmax.py` όπου την επισκιάσαμε.
- **Σχεδίαση με `matplotlib`.** Η `plt.plot(x, y)` ενώνει τα σημεία (x_i, y_i) με γραμμή. Το `'ro'` σημαίνει κόκκινους (`r`) κύκλους (`o`) χωρίς γραμμή. Οι `axhline`/`axvline` σχεδιάζουν τους άξονες, η `legend` εμφανίζει τις ετικέτες των `label=` και η `show` ανοίγει το παράθυρο. Το παράθυρο σχεδίασης προσαρμόζεται γύρω από το σημείο καμπής $x_0 = -b/(3a)$, με πλάτος που εξαρτάται από την απόσταση των ακροτάτων: ακόμη και η «εμφάνιση» ενός γραφήματος χρειάζεται `if`.

4 Επανάληψη με `for`

Για να εκτελέσουμε μια ομάδα εντολών πολλές φορές χρησιμοποιούμε **βρόχο** (loop, βρόχο επανάληψης). Ο βρόχος `for` επαναλαμβάνει ένα μπλοκ για κάθε στοιχείο μιας ακολουθίας. Τη συνηθέστερη ακολουθία, τους διαδοχικούς ακεραίους, τη δίνει η `range`.

4.1 Παραγοντικό: ο «συσσωρευτής»

Σύνταξη του `for`.

```
for μεταβλητή in ακολουθία:
    μπλοκ εντολών (με εσοχή)
```

Η μεταβλητή παίρνει διαδοχικά κάθε τιμή της ακολουθίας και για καθεμία εκτελείται το μπλοκ.

Η `range`.

Κλήση	Παράγει
`range(n)`	`0, 1, …, n-1`
`range(a, b)`	`a, a+1, …, b-1`
`range(a, b, step)`	`a, a+step, a+2·step, …` όσο `< b`

Το **άνω άκρο δεν περιλαμβάνεται**. Γι' αυτό για να πάρουμε `1, …, n` γράφουμε `range(1, n + 1)`. Αν `a >= b` (με θετικό βήμα) η `range` είναι κενή και το μπλοκ δεν εκτελείται καμία φορά. Έτσι για `n = 0` το `f` μένει `1`, που είναι σωστό.

Ο συσσωρευτής (accumulator). Το σχήμα «αρχικοποίηση πριν τον βρόχο, ενημέρωση μέσα στον βρόχο» είναι το πιο συχνό μοτίβο προγραμματισμού:

- για **γινόμενο** ξεκινάμε από `1` και πολλαπλασιάζουμε,
- για **άθροισμα** ξεκινάμε από `0` και προσθέτουμε,
- για **μέτρηση** ξεκινάμε από `0` και προσθέτουμε `1`.

Σύνθετη ανάθεση. Το `f *= i` είναι συντομογραφία του `f = f * i`. Ομοίως `+=`, `-=`, `/=`, `//=`, `%=`, `**=`.

`print` με πολλά ορίσματα. Το `print(a, b, c)` τυπώνει τις τιμές χωρισμένες με κενό. Είναι η απλή εναλλακτική των f-strings όταν δεν μας νοιάζει η μορφοποίηση.

Αναλλοίωτη του βρόχου (loop invariant). Το σχόλιο «μετά το βήμα `i` ισχύει `f = i!`» είναι μια πρόταση που είναι αληθής σε κάθε επανάληψη. Τέτοιες προτάσεις είναι ο τρόπος να *αποδείξουμε* ότι ένας βρόχος κάνει αυτό που θέλουμε.

4.2 Έλεγχος πρώτου αριθμού: σημαίες, `%` και υπό συνθήκη έκφραση

- **Υπόλοιπο `%`.** Το `z % i` είναι το υπόλοιπο της διαίρεσης του `z` με το `i`. «Το `i` διαιρεί το `z`» γράφεται `z % i == 0`. Είναι ο βασικός έλεγχος σε όλα τα προγράμματα με πρώτους.
- **Σημαία (flag, σημαιοφόρος).** Η `isprime` είναι μια λογική μεταβλητή που «θυμάται» αν συνέβη κάτι μέσα στον βρόχο. Την αρχικοποιούμε στην «αισιόδοξη» τιμή `True` και την αλλάζουμε σε `False` μόλις βρεθεί διαιρέτης. Αφού γίνει `False`, δεν ξαναγίνεται `True`.
- **`if` μέσα σε `for`.** Το μπλοκ του `if` έχει διπλή εσοχή: ανήκει στο `if`, το οποίο ανήκει στο `for`.
- **Υπό συνθήκη έκφραση.** Το `x = '' if isprime else 'δεν'` είναι ένα `if … else` σε μία γραμμή που **επιστρέφει τιμή**: το `x` γίνεται `''` (κενό) αν `isprime`, αλλιώς `'δεν'`. Έτσι το ίδιο `print` τυπώνει «είναι πρώτος» ή «δεν είναι πρώτος».
- **Αδυναμίες.** Ο βρόχος συνεχίζει να δοκιμάζει διαιρέτες ακόμη και αφού βρει έναν, και δοκιμάζει πολύ περισσότερους από όσους χρειάζεται. Επίσης το `z = 1` θα δηλωθεί «πρώτος». Τα επόμενα προγράμματα βελτιώνουν σταδιακά και τα δύο.

4.3 Βελτίωση 1: μόνο περιττοί διαιρέτες, το `step` της `range`

- **`range(3, z, 2)`** παράγει `3, 5, 7, …`: το τρίτο όρισμα είναι το **βήμα**. Έτσι οι δοκιμές μειώνονται στο μισό.
- **Ειδικές περιπτώσεις πριν τη γενική.** Το 2 (ο μόνος άρτιος πρώτος) και οι άρτιοι αντιμετωπίζονται με `if`, και ο βρόχος τρέχει μόνο για περιττούς. Είναι συνηθισμένη δομή: *πρώτα οι ειδικές περιπτώσεις, μετά ο βρόχος*.
- **Εκτύπωση λογικής τιμής.** Το `print(isprime)` τυπώνει `True` ή `False`.

4.4 Βελτίωση 2: διαιρέτες μέχρι την τετραγωνική ρίζα

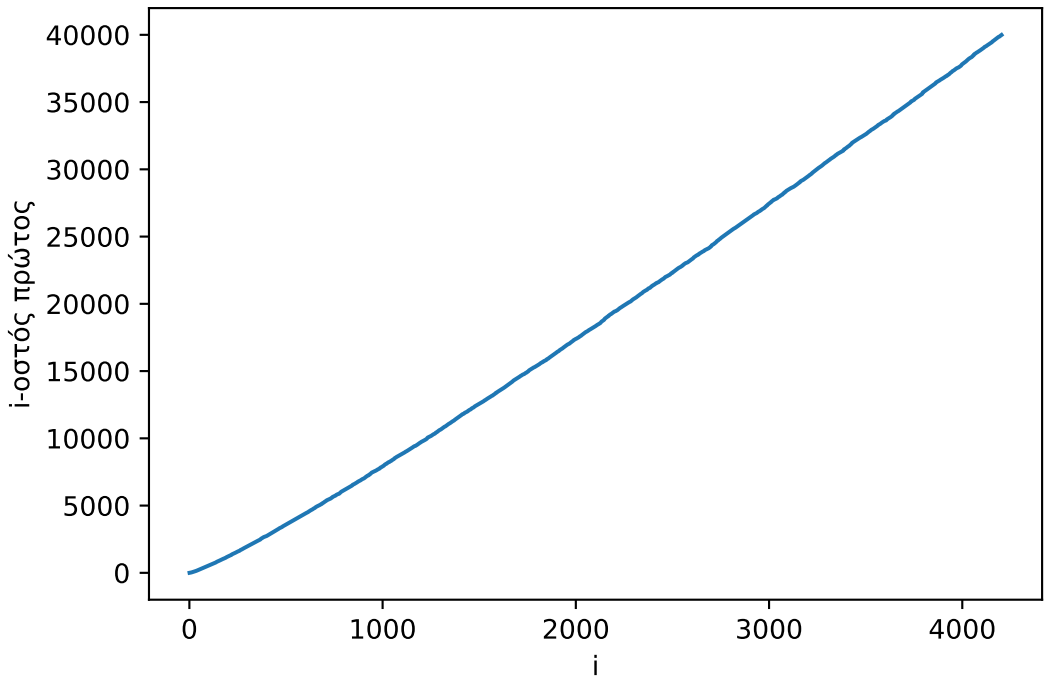
Η μαθηματική παρατήρηση. Αν ο n είναι σύνθετος, $n = p \cdot q$ με $p \le q$, τότε $p \le \sqrt{n}$. Άρα αρκεί να δοκιμάσουμε διαιρέτες μέχρι το $\lceil \sqrt{n} \rceil$. Για $n = 10^6$ αυτό σημαίνει περίπου 500 δοκιμές αντί για 500.000.

- **Εμφωλευμένοι βρόχοι.** Ο εξωτερικός `for n` διατρέχει τους υποψήφιους, ο εσωτερικός `for k` τους διαιρέτες του τρέχοντος `n`. Η σημαία `a` πρέπει να **επαναρχικοποιείται** σε `True` σε κάθε επανάληψη του εξωτερικού βρόχου, γι' αυτό βρίσκεται μέσα στον `for n`.
- **`math.ceil`, `math.sqrt`.** Η `math.sqrt(n)` δίνει `float`, ενώ `n` `range` θέλει ακεραίους, γι' αυτό `n` `math.ceil` (στρογγυλοποίηση προς τα πάνω). Η Python έχει και την `math.isqrt(n)`, την ακέραια τετραγωνική ρίζα, που είναι και ακριβέστερη (θα τη δούμε παρακάτω).
- **`print(..., end=' ')`.** Κανονικά η `print` αλλάζει γραμμή στο τέλος. Με το προαιρετικό όρισμα `end=' '` βάζει κενό αντί για αλλαγή γραμμής, ώστε οι πρώτοι να τυπωθούν στην ίδια σειρά.
- **`import math, time`.** Μπορούμε να φορτώσουμε δύο βιβλιοθήκες σε μία γραμμή. Η `time` φορτώνεται για χρονομέτρηση (π.χ. `t0 = time.time()` πριν και `time.time() - t0` μετά), αλλά στην τελική εκδοχή δεν χρησιμοποιήθηκε.
- **Προσοχή στο όριο.** Το σχόλιο λέει «έως το 10000» αλλά η `range` φτάνει ως 10^8: το πρόγραμμα όπως είναι θα τρέξει πολλή ώρα. Αλλάξτε το όριο σε `10001` για να το δοκιμάσετε. Είναι καλή υπενθύμιση ότι τα **σχόλια πρέπει να ενημερώνονται** μαζί με τον κώδικα.

4.5 Βρόχος μέσα σε βρόχο σε ένα διάστημα και το `not`

- **Κάτω παύλα σε αριθμούς.** Το `1_000_000` είναι ο αριθμός 1000000. Η Python αγνοεί τις κάτω παύλες μέσα σε αριθμητικές σταθερές, εμείς όμως τις διαβάζουμε ευκολότερα.
- **`not`.** Το `not isprime` αντιστρέφει τη λογική τιμή. Εδώ χρησιμοποιείται απλώς για να τυπωθεί `False` (θα μπορούσαμε να γράψουμε `print(z, False)`).
- **Ακέραιο πηλίκο `//`.** Το `z // 2` είναι το ακέραιο μέρος του `z/2` και χρησιμοποιείται ως άνω όριο: κανένας γνήσιος διαιρέτης δεν ξεπερνά το μισό του αριθμού. Είναι καλύτερο από το `z` αλλά πολύ χειρότερο από το $\sqrt{z}$ του προηγούμενου προγράμματος.
- **`if … else` με βρόχο μέσα στο `else`.** Μπλοκ μπορούν να περιέχουν οποιαδήποτε εντολή, άρα και βρόχους. Η εσοχή μετρά το «βάθος».

4.6 Λίστες, `break`, `math.isqrt` και γράφημα των πρώτων



Σχήμα 5: Ο i-οστός πρώτος ως συνάρτηση του i, για τους πρώτους έως 40000.

Τέσσερα νέα στοιχεία:

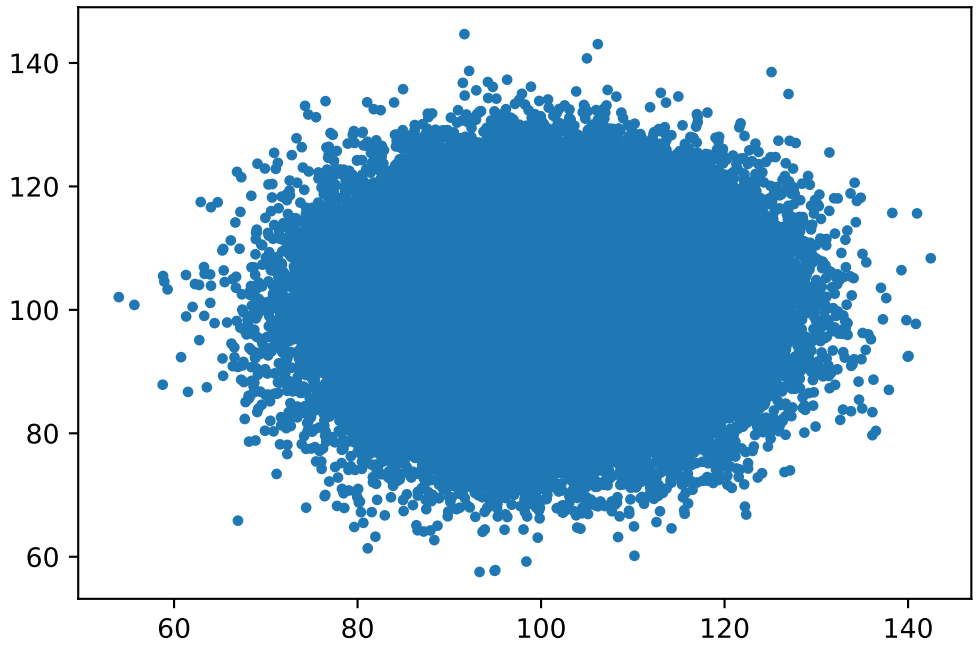
- **Λίστες.** Το `y = [2]` δημιουργεί μια **λίστα** με ένα στοιχείο. Η λίστα είναι μια διατεταγμένη συλλογή τιμών που μπορεί να μεγαλώνει. Το `y.append(c)` προσθέτει το `c` στο τέλος, το `len(y)` δίνει το πλήθος των στοιχείων και το `y[i]` το στοιχείο στη θέση `i` (αρίθμηση από `0`, όπως στα αλφαριθμητικά). Το `list(range(1, l + 1))` μετατρέπει μια `range` σε κανονική λίστα `[1, 2, …, l]`.
- **`break`.** Μόλις βρεθεί διαιρέτης, η `break` **τερματίζει αμέσως** τον εσωτερικό βρόχο. Δεν έχει νόημα να ψάχνουμε κι άλλους διαιρέτες. Η `break` τερματίζει μόνο τον **εσωτερικότερο** βρόχο στον οποίο βρίσκεται. Η εκτέλεση συνεχίζει με το `if flag`.
- **`math.isqrt`.** Η ακέραια τετραγωνική ρίζα $\lfloor \sqrt{c} \rfloor$, υπολογισμένη ακριβώς με ακέραια αριθμητική. Προτιμάται από το `math.ceil(math.sqrt(c))` για μεγάλους αριθμούς.
- **Σημαία ως `0`/`1`.** Η `flag` είναι ακέραιος αντί για `bool`, και το `if flag:` είναι αληθές όταν `flag != 0`. Γενικά η Python θεωρεί «αληθή» κάθε μη μηδενική τιμή και «ψευδή» το `0`, το `0.0`, το κενό αλφαριθμητικό και την κενή λίστα.

Το γράφημα δείχνει τον i-οστό πρώτο ως συνάρτηση του i. Η καμπύλη είναι περίπου $i \ln i$ (θεώρημα των πρώτων αριθμών).

5 Τυχαίοι αριθμοί και προσεγγίσεις του π

Η ενότητα αυτή συνεχίζει με το `for` και εστιάζει σε υπολογισμούς με **πολλές επαναλήψεις**: σειρές, μετρήσεις σημείων και προσομοιώσεις με τυχαίους αριθμούς (μέθοδος Monte Carlo).

5.1 Η βιβλιοθήκη `random` και νέφος σημείων



Σχήμα 6: Νέφος 100000 σημείων με κανονική κατανομή στις δύο συντεταγμένες.

- **`random`.** Η βιβλιοθήκη παράγει ψευδοτυχαίους αριθμούς:

Κλήση	Επιστρέφει
`random.random()`	ομοιόμορφα στο $[0, 1)$
`random.uniform(a, b)`	ομοιόμορφα στο $[a, b]$
`random.normalvariate(m, s)`	κανονική κατανομή με μέση τιμή `m` και τυπική απόκλιση `s`
`random.randint(a, b)`	ακέραιο στο $\{a, \ldots, b\}$

- **Κενή λίστα και `append` μέσα σε βρόχο.** Το `x = []` είναι μια κενή λίστα. Σε κάθε επανάληψη προσθέτουμε ένα στοιχείο. Μετά τον βρόχο έχουμε 100000 σημεία.
- **Μεταβλητή βρόχου που δεν χρησιμοποιείται.** Το `i` δεν εμφανίζεται μέσα στο μπλοκ: ο βρόχος απλώς επαναλαμβάνει «κάτι» 100000 φορές. Πολλοί προγραμματιστές γράφουν τότε `for _ in range(100000)`.
- **Σχεδίαση σημείων.** Το τρίτο όρισμα `"."` στην `plt.plot` σημαίνει «μόνο τελείες, χωρίς γραμμές». Το νέφος είναι κυκλικό γύρω από το $(100, 100)$ με «ακτίνα» περίπου 3 τυπικές αποκλίσεις, δηλαδή 30. (Τα σχόλια στο αρχείο αναφέρουν παλαιότερες τιμές: με `normalvariate(1, .1)` επί 100 η τυπική απόκλιση είναι 10, όχι 100, και τα σημεία είναι 100000, όχι 10000.)
- **Προσοχή στο `print` μιας λίστας με 100000 στοιχεία:** γεμίζει την οθόνη. Τυπώστε καλύτερα μόνο `len(x)` ή τα πρώτα στοιχεία `x[:5]`.

5.2 Η σειρά του Leibniz: εμφωλευμένοι βρόχοι και σύγκλιση

$$\frac{\pi}{4} = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

- **Συσσωρευτής αθροίσματος.** Το `e` αρχικοποιείται στο `0` και προστίθεται ο k-οστός όρος σε κάθε βήμα. Το `e = e + y` είναι το ίδιο με `e += y`.
- **Εναλλασσόμενο πρόσημο.** Το `(-1)**(k+1)` είναι `+1` για περιττό `k` και `-1` για άρτιο. Εναλλακτικά θα μπορούσαμε να κρατάμε μια μεταβλητή `s = 1` και να γράφουμε `s = -s` σε κάθε βήμα, που είναι και ταχύτερο.
- **Εξωτερικός βρόχος για πείραμα.** Ο `for i in range(1, 7)` τρέχει τον ίδιο υπολογισμό για $n = 10, 10^2, \ldots, 10^6$ όρους, ώστε να δούμε πώς μειώνεται το σφάλμα. Το `e = 0` βρίσκεται μέσα στον εξωτερικό βρόχο: κάθε πείραμα ξεκινά από την αρχή.
- **Ερμηνεία.** Το σφάλμα είναι περίπου $1/n$: δεκαπλασιάζοντας τους όρους κερδίζουμε ένα δεκαδικό ψηφίο. Η σειρά συγκλίνει πολύ αργά.

5.3 Βελάκια: Monte Carlo με `random.uniform`

- **Μετρητής.** Το `hits` είναι συσσωρευτής μέτρησης: αρχίζει από `0` και αυξάνεται κατά `1` (`hits += 1`) **μόνο όταν** ισχύει η συνθήκη.
- **Αποφυγή περιττών πράξεων.** Το «απόσταση ≤ 1» ελέγχεται ως $x^2 + y^2 \le 1$, χωρίς τετραγωνική ρίζα: η ρίζα είναι αύξουσα συνάρτηση, οπότε η σύγκριση είναι ισοδύναμη και γλιτώνουμε μία πράξη σε κάθε επανάληψη.
- **Τι περιμένουμε.** Η εκτίμηση `my_pi` αλλάζει σε κάθε εκτέλεση, γιατί οι τυχαίοι αριθμοί αλλάζουν. Το σφάλμα μειώνεται σαν $1/\sqrt{n}$: για 100 φορές περισσότερες ρίψεις κερδίζουμε ένα δεκαδικό ψηφίο.

5.4 Η βελόνα του Buffon: `floor`, `ceil` και πρόσθεση λογικών τιμών

- **Μέτρηση χωρίς `if`.** Η έκφραση `(math.floor(...) == math.ceil(...))` είναι `True` ή `False`. Προσθέτοντάς την στο `success` προσθέτουμε `1` ή `0`. Είναι η ίδια ιδέα με το `6_max1.py`: οι λογικές τιμές είναι αριθμοί. Το `hits += 1` μέσα σε `if` και το `success += (συνθήκη)` κάνουν ακριβώς το ίδιο.
- **`math.floor`, `math.ceil`.** Ακέραιο μέρος προς τα κάτω και προς τα πάνω. Υπάρχει ακέραιος στο $[u, v]$ ακριβώς όταν $\lceil u \rceil \le \lfloor v \rfloor$, και όταν το διάστημα έχει μήκος το πολύ 1 αυτό σημαίνει ισότητα.
- **Οι ενσωματωμένες `max` και `min`** χρησιμοποιούνται εδώ ως συναρτήσεις, πράγμα που θα ήταν αδύνατο αν είχαμε νωρίτερα γράψει `max = …`.
- **Κλίμακα τυχαίου αριθμού.** Το `random.random() * 2 * math.pi` μετατρέπει έναν ομοιόμορφο αριθμό στο $[0, 1)$ σε ομοιόμορφο στο $[0, 2\pi)$.

5.5 Σημεία πλέγματος σε κύκλο: μέτρηση χωρίς τυχαιότητα

Σε αντίθεση με τις δύο προηγούμενες μεθόδους, εδώ **δεν υπάρχει τυχαιότητα**: για την ίδια ακτίνα `w` παίρνουμε πάντα το ίδιο αποτέλεσμα.

- **Αντικατάσταση εσωτερικού βρόχου με τύπο.** Θα μπορούσαμε να διατρέξουμε όλα τα ζεύγη `(k, j)` με δύο εμφωλευμένους βρόχους και να μετράμε όσα ικανοποιούν $k^2 + j^2 \le w^2$. Αντί γι' αυτό, για κάθε `k` **υπολογίζουμε** πόσα `j` ταιριάζουν: $\lfloor\sqrt{w^2 - k^2}\rfloor$. Ο υπολογισμός γίνεται $O(w)$ αντί $O(w^2)$.
- **Μορφοποίηση σε στήλες.** Οι τρεις `print` με σταθερά πλάτη (`8d`, `18.12f`) δίνουν στοιχισμένη έξοδο. Συγκρίνετε με το απλό `print(my_pi)` του προγράμματος με τα βελάκια.
- **Σύγκριση μεθόδων.** Το σφάλμα εδώ είναι περίπου $1/w$, όπως στη σειρά Leibniz, αλλά με πολύ λιγότερες πράξεις: w επαναλήψεις για ακρίβεια $\sim 1/w$.

6 Επανάληψη με `while`

Ο βρόχος `for` είναι κατάλληλος όταν **ξέρουμε από πριν** πόσες επαναλήψεις θέλουμε. Όταν ο αριθμός των επαναλήψεων εξαρτάται από αυτό που συμβαίνει *κατά* την εκτέλεση («μέχρι να δώσει σωστή τιμή ο χρήστης», «μέχρι να βρεθεί διαιρέτης»), χρησιμοποιούμε `while`.

6.1 Επανάληψη μέχρι να δοθεί σωστή είσοδος

Σύνταξη.

```
while συνθήκη:
    μπλοκ εντολών
```

Η συνθήκη ελέγχεται **πριν** από κάθε επανάληψη. Αν είναι αληθής εκτελείται το μπλοκ και ξαναελέγχεται. Αν είναι ψευδής από την αρχή, το μπλοκ δεν εκτελείται καθόλου.

Το μοτίβο «διάβασε, και όσο είναι λάθος ξαναδιάβασε». Η πρώτη `input` βρίσκεται πριν τον βρόχο, ώστε η `x` να έχει τιμή όταν ελεγχθεί η συνθήκη. Μέσα στον βρόχο η `x` ξαναδιαβάζεται: αυτό είναι ουσιώδες. Αν ξεχάσουμε να **αλλάξουμε μέσα στον βρόχο** κάτι που επηρεάζει τη συνθήκη, ο βρόχος δεν τελειώνει ποτέ (ατέρμων βρόχος, τον σταματάμε με Ctrl+C).

6.2 Έλεγχος πρώτου με `while`

Ο βρόχος `for i in range(2, c + 1)` γράφεται με `while` ως εξής: **αρχικοποίηση** `i = 2` πριν τον βρόχο, **συνθήκη** `i <= c` στο `while`, **αύξηση** `i += 1` στο τέλος του μπλοκ. Τα τρία αυτά βήματα, που η `range` κάνει αυτόματα, εδώ είναι δική μας ευθύνη. Το `i += 1` στο τέλος είναι η γραμμή που ξεχνιέται συχνότερα.

Το `and r != 2` εξαιρεί την περίπτωση $r = 2$, όπου το `i = 2` διαιρεί το `r` ενώ το 2 είναι πρώτος. (Για $r = 2$ ο βρόχος τρέχει μία φορά αφού $\lceil\sqrt{2}\rceil = 2$.)

6.3 Έλεγχος πρώτου με `while` που σταματά νωρίς

Εδώ το `while` δείχνει το πραγματικό του πλεονέκτημα: η συνθήκη `r % i != 0 and i <= c` λέει «συνέχισε **όσο** δεν έχει βρεθεί διαιρέτης **και** δεν έχουμε ξεπεράσει τη ρίζα». Ο βρόχος σταματά για έναν από τους δύο λόγους, και **ποιος** από τους δύο ήταν το μαθαίνουμε μετά τον βρόχο:

- αν `i > c`, εξαντλήσαμε τους διαιρέτες χωρίς να βρούμε κανέναν, άρα πρώτος,
- αλλιώς σταματήσαμε γιατί `r % i == 0`, άρα σύνθετος.

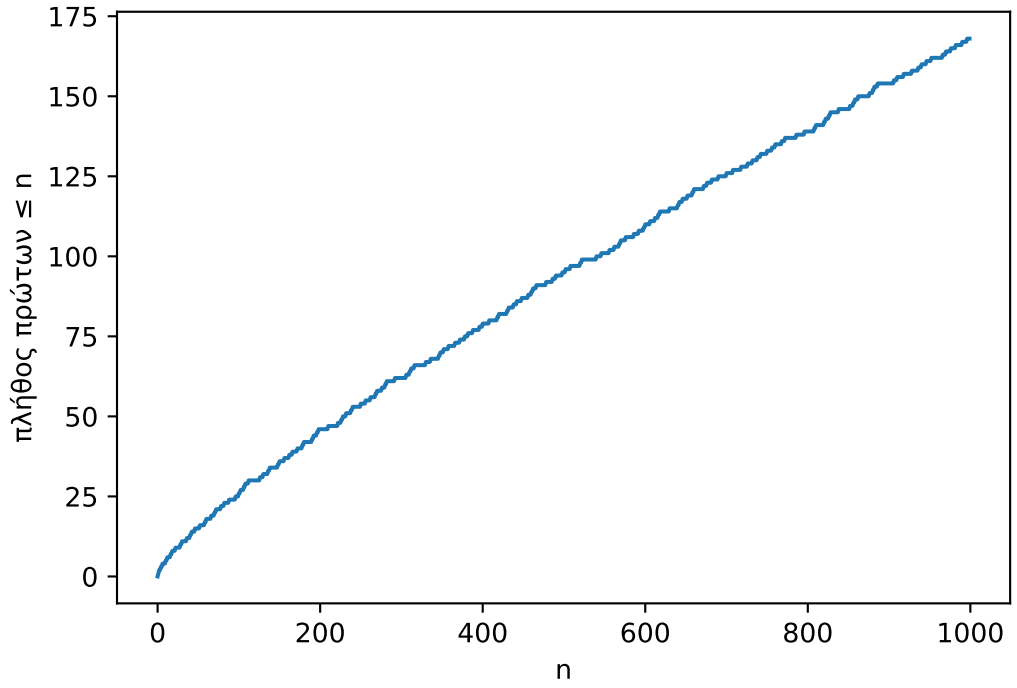
Η ανάθεση `isprime = i > c` αποθηκεύει κατευθείαν το αποτέλεσμα της σύγκρισης στη λογική μεταβλητή, χωρίς `if`. Η συνθήκη τερματισμού του `while` παίζει εδώ τον ρόλο που έπαιζε η `break` στον βρόχο `for`.

Άλλα στοιχεία:

- **Έλεγχος εισόδου με `while`** (όπως στο `1_simple_while.py`) πριν τον κυρίως υπολογισμό.
- **Ειδικές περιπτώσεις με `if / elif / else`.** Ο βρόχος τρέχει μόνο για περιττούς $r > 4$. Στο `else` καταλήγουν το 2, το 3 και, λανθασμένα, το 1. Το `r = 1` δηλώνεται πρώτος, ενώ δεν είναι. Πώς θα το διορθώνατε;
- **Σειρά στη συνθήκη `and`.** Η Python αποτιμά το `and` με **βραχυκύκλωση** (short-circuit evaluation): αν το αριστερό μέλος είναι `False`, το δεξί δεν υπολογίζεται καν. Εδώ και τα δύο μέλη είναι

ασφαλή με οποιαδήποτε σειρά, αλλά γενικά βάζουμε πρώτα τον έλεγχο που «προστατεύει» τον δεύτερο (π.χ. `i < len(a) and a[i] > 0`).

6.4 Λίστα λογικών τιμών και σωρευτικό άθροισμα



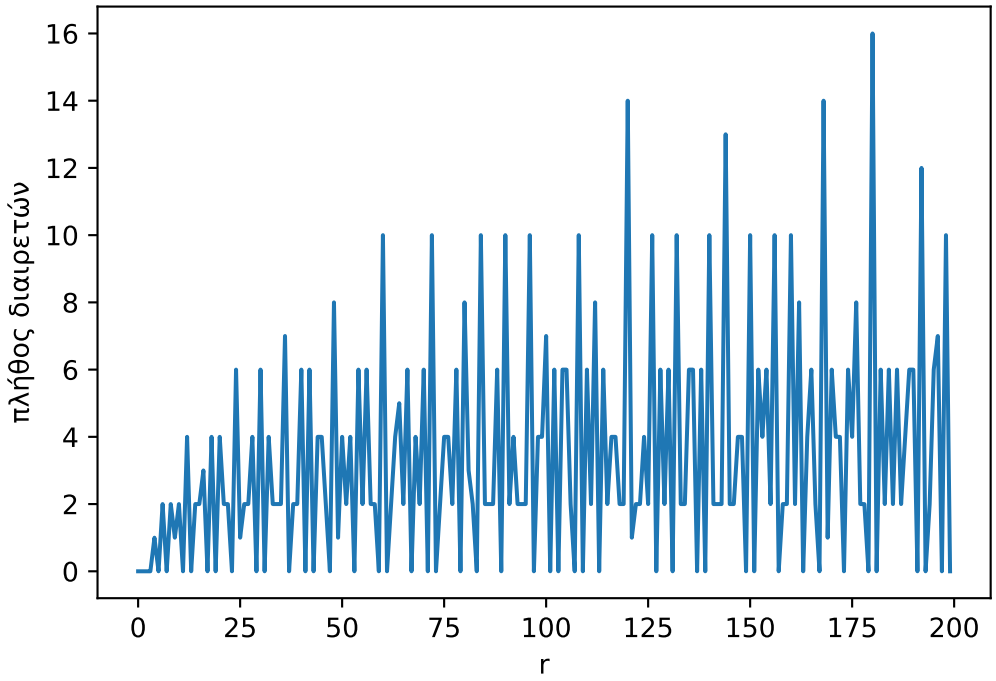
Σχήμα 7: Το γράφημα των δύο τελευταίων (σχολιασμένων) γραμμών: το πλήθος των πρώτων έως κάθε αριθμό, δηλαδή n συνάρτηση π(n) για n ≤ 1000.

Το πρόγραμμα έχει τρεις φάσεις, καθεμία με δικό της βρόχο:

1. **Κατασκευή λίστας λογικών τιμών.** Η `a` αρχίζει με τις απαντήσεις για τους 1, 2, 3, 4 (`[False, True, True, False]`) και για κάθε $r = 5, \ldots, 1000$ προσθέτουμε το αποτέλεσμα `i > c and r % 2 == 1`. Προσέξτε τη **μετατόπιση δεικτών**: επειδή οι λίστες αριθμούνται από το 0, το `a[r-1]` αφορά τον αριθμό `r`. Το `while` μέσα στο `for` είναι ο ίδιος έλεγχος με το προηγούμενο πρόγραμμα. Ο όρος `r % 2 == 1` απορρίπτει τους άρτιους: ο βρόχος δοκιμάζει μόνο περιττούς διαιρέτες, άρα δεν βρίσκει ποτέ τον διαιρέτη 2, και χωρίς αυτόν τον όρο θα θεωρούσε τους άρτιους πρώτους.
2. **Σωρευτικό άθροισμα.** Η `y[i]` είναι το πλήθος των πρώτων μέχρι τον αριθμό `i+1`, δηλαδή η συνάρτηση $\pi(n)$. Κάθε νέο στοιχείο είναι το προηγούμενο (`y[i-1]`) συν 1 αν ο αριθμός είναι πρώτος. Αυτό είναι το **μοτίβο του σωρευτικού αθροίσματος** και δείχνει πώς ένας βρόχος **διαβάζει** μια λίστα (`a[i]`) και **χτίζει** μια άλλη.
3. **Μετατροπή σε λίστα πρώτων.** Ο τρίτος βρόχος διατρέχει τις **θέσεις** `i` της `a` (`range(len(a))`) και όπου `a[i]` είναι `True` προσθέτει τον αντίστοιχο αριθμό `i + 1`.

Οι δύο τελευταίες γραμμές (σε σχόλια) σχεδιάζουν την $\pi(n)$: `plt.plot(y)` με ένα μόνο όρισμα χρησιμοποιεί ως τετμημένες τις θέσεις `0, 1, 2, …`.

6.5 Μέτρηση διαιρετών και `while` μέσα σε `for`



Σχήμα 8: Το πλήθος των γνήσιων διαιρετών κάθε αριθμού, για z = 200. Τα μηδενικά είναι οι πρώτοι.

- **Πολλές μεταβλητές που επαναρχικοποιούνται.** Για κάθε `r` ξεκινάμε από την αρχή τα `i`, `isprime`, `d`: βρίσκονται μέσα στον `for` αλλά **πριν** τον `while`.
- **Μετρητής `d`.** Αυξάνεται κάθε φορά που βρίσκεται διαιρέτης, οπότε στο τέλος είναι το πλήθος των διαιρετών του `r` στο $\{2, \dots, r-1\}$, δηλαδή των γνήσιων διαιρετών του εκτός από το 1. Για πρώτο `r` το `d` είναι `0`.
- **`count += isprime`.** Ξανά πρόσθεση λογικής τιμής: το `count` αυξάνεται κατά 1 σε κάθε βήμα του `while` στο οποίο η `isprime` είναι ακόμη `True`. Το νόημα αυτής της ποσότητας είναι λιγότερο προφανές (μετρά «πόσα `i` δοκιμάστηκαν πριν βρεθεί ο πρώτος διαιρέτης», αθροιστικά για όλα τα `r`) και η μεταβλητή δεν τυπώνεται. Καλή άσκηση: τι ακριβώς μετρά το `count`, και πώς θα το χρησιμοποιούσατε για να μετρήσετε το κόστος του αλγορίθμου;
- **Το γράφημα** `plt.plot(a)` δείχνει το πλήθος διαιρετών κάθε αριθμού: τα μηδενικά είναι οι πρώτοι, οι κορυφές είναι αριθμοί με πολλούς διαιρέτες (π.χ. το 120, το 168, το 180).
- **Κόστος.** Ο εσωτερικός βρόχος κάνει $r-2$ βήματα για κάθε `r`, άρα συνολικά περίπου $z^2/2$ διαιρέσεις. Για `z = 10000` είναι 50 εκατομμύρια πράξεις. Συγκρίνετε με το προηγούμενο πρόγραμμα, που σταματά στη ρίζα.

7 Σύνοψη των στοιχείων της γλώσσας

Στοιχείο	Πρώτη εμφάνιση	Παρατηρήσεις
Αλγόριθμος, διάγραμμα ροής, ψευδοκώδικας	πρώτα βήματα	γραμμική αναζήτηση
`type`, `int`/`float`/`str` ως τύποι	πρώτα βήματα	δυναμικοί τύποι
Πολλαπλή ανάθεση, ανταλλαγή τιμών `a, b = b, a`	πρώτα βήματα	πρώτα το δεξί μέλος
Αναφορές, `is`, `id`	πρώτα βήματα	ορατό στις λίστες
`print(…, sep=, end=)`	πρώτα βήματα	
Βιβλιοθήκες, `import`, `import … as …`	πρώτα βήματα	`math`, `random`, `numpy`, `matplotlib`
Σχόλια `#`	`1_prpr.py`	και για προσωρινή απενεργοποίηση κώδικα
Ανάθεση `=`	`1_prpr.py`	«υπολόγισε δεξιά, αποθήκευσε αριστερά»

Στοιχείο	Πρώτη εμφάνιση	Παρατηρήσεις
`input`, `int`, `float`, `str`	`1_prpr.py`	η `input` επιστρέφει πάντα κείμενο
f-strings, `{x:8.2f}`, `{n:4d}`, `{x:.3e}`	`1_prpr.py`	πλάτος, δεκαδικά, εκθετική μορφή
`\n`, `\t`, `\'`	`1_prpr.py`, `4_torus.py`	ειδικοί χαρακτήρες
`math.pi`, `math.sqrt`	`2_SurfAreaFirst.py`	`floor`, `ceil`, `isqrt`, `sin` αργότερα
`**`, `/`, `//`, `%`	`2_SurfAreaFirst.py`, `prime_simple.py`	`%` είναι ο έλεγχος διαιρετότητας
`numpy`, 3D γράφημα	`4_torus.py`	`np.linspace`, `np.meshgrid`, `plot_surface`
`abs`, `max`, `min`, `round`, `len`	`5_minmax.py`	μην επισκιάζετε τα ονόματά τους
Αλφαριθμητικά: `+`, `s[i]`	`6_string.py`	αρίθμηση από το 0
`if / elif / else`, εσοχή	`2_first.py`, `3_comp.py`	ακριβώς ένας κλάδος εκτελείται
`< <= > >= == !=`	`2_first.py`	το `==` είναι σύγκριση, το `=` ανάθεση
`and`, `or`, `not`	`4_ifthen.py`, `prime_interval.py`	αποτίμηση με βραχυκύκλωση
`True`/`False` ως 1/0	`6_max1.py`, `π_Buffon.py`	`count += συνθήκη`
`for … in range(a, b, step)`	`fact.py`, `primetest_odd.py`	το `b` δεν περιλαμβάνεται
Συσσωρευτής, `*=`, `+=`	`fact.py`, `π_Leibniz.py`	αρχικοποίηση **πριν** τον βρόχο
Σημαία `isprime = True`	`prime_simple.py`	επαναρχικοποίηση σε κάθε εξωτερική επανάληψη
Υπό συνθήκη έκφραση `a if c else b`	`prime_simple.py`	`if` που επιστρέφει τιμή
`print(…, end=' ')`, `print(a, b)`	`prime_sqrt.py`, `fact.py`	
`1_000_000`	`prime_interval.py`	κάτω παύλες σε αριθμούς
Λίστες: `[]`, `append`, `len`, `a[i]`, `list(range(…))`	`prime_vector.py`	
`break`	`prime_vector.py`	τερματίζει τον εσωτερικότερο βρόχο
`random.random`, `uniform`, `normalvariate`	`rand.py`, `π_βελάκια.py`	Monte Carlo
`while συνθήκη:`	`1_simple_while.py`	αρχικοποίηση, συνθήκη, ενημέρωση
`isprime = i > c`	`3_primewhile.py`	αποθήκευση αποτελέσματος σύγκρισης
Σωρευτικό άθροισμα σε λίστα	`5_primeall.py`	`y.append(y[i-1] + …)`

Πρόγραμμα 13 `7_poly.py`

```
# Ἀνάλυση πολυωνύμου q(x)=ax^3+bx^2+cx+d, a≠0
# Ἕνα πολυώνυμο εἶναι ἁπλὸ ὅταν ὅλες οἱ ρίζες εἶναι διακριτὲς καὶ πραγματικές
# Ἕνα πολυώνυμο εἶναι μονότονο ὅταν εἶναι φθίνουσα ἢ αὔξουσα συνάρτηση
import math

a = float(input('Δῶσε τὸ a: '))
b = float(input('Δῶσε τὸ b: '))
c = float(input('Δῶσε τὸ c: '))
d = float(input('Δῶσε τὸ d: '))

print(f'Τὸ πολυώνυμο εἶναι τὸ {a:6.2f}x^3+{b:6.2f}x^2+{c:6.2f}x+{d:6.2f}')

a1 = a * 3
b1 = b * 2
c1 = c

print(f"Ἡ παράγωγος τοῦ πολυωνύμου εἶναι τὸ q'(x)={a1:6.2f}x^2+{b1:6.2f}x+{c1:6.2f}")

discriminant = b1**2 - 4*a1*c1

print(f"Ἡ διακρίνουσα τῆς παραγώγου τοῦ q'(x) εἶναι {discriminant:6.2f}")

if discriminant <= 0:
    print('Τὸ πολυώνυμο q(x) εἶναι μονότονο καὶ ὄχι ἁπλό')
else:
    print('Τὸ πολυώνυμο q(x) δὲν εἶναι μονότονο')
    r1 = (-b1 + math.sqrt(discriminant)) / (2*a1)
    r2 = (-b1 - math.sqrt(discriminant)) / (2*a1)
    qr1 = a*r1**3 + b*r1**2 + c*r1 + d
    qr2 = a*r2**3 + b*r2**2 + c*r2 + d
    print(f'r1 = {r1:6.2f}, qr1 = {qr1:6.2f},r2 = {r2:6.2f}, qr2 = {qr2:6.2f}  ')
    if qr1*qr2 < 0:
        print('Τὸ πολυώνυμο q(x) εἶναι ἁπλό')
    else:
        print('Τὸ πολυώνυμο q(x) δὲν εἶναι ἁπλό')




# Σχεδίαση τῆς γραφικῆς παράστασης τοῦ q(x)
import numpy as np
import matplotlib.pyplot as plt

x0 = -b / (3*a)          # σημεῖο καμπῆς, ἐκεῖ ποὺ q''(x)=0
if discriminant > 0:
    w = max(2, 1.5 * abs(r1 - r2))
else:
    w = 3
x = np.linspace(x0 - w, x0 + w, 400)
y = a*x**3 + b*x**2 + c*x + d

plt.plot(x, y, label='q(x)')
plt.axhline(0, color='black', linewidth=0.8)
plt.axvline(0, color='black', linewidth=0.8)
if discriminant > 0:
    plt.plot([r1, r2], [qr1, qr2], 'ro', label='τοπικὰ ἀκρότατα')
plt.title(f'q(x) = {a:.2f}x³ + {b:.2f}x² + {c:.2f}x + {d:.2f}')
plt.xlabel('x')
plt.ylabel('q(x)')
plt.grid(True)
plt.legend()
plt.show()
```

Πρόγραμμα 14 `fact.py`

```
# fact.py -- παραγοντικό με βρόχο for
# Υπολογίζει το n! = 1 · 2 · 3 · ... · n.

# Διαβάζουμε τον ακέραιο n από τον χρήστη.
n = int(input('Δώσε έναν ακέραιο: '))

# Το f κρατά το γινόμενο που έχει υπολογιστεί ως τώρα.
# Αρχική τιμή 1, το ουδέτερο στοιχείο του πολλαπλασιασμού,
# οπότε για n = 0 προκύπτει σωστά 0! = 1.
f = 1

# Πολλαπλασιάζουμε το f διαδοχικά με 1, 2, ..., n.
# Μετά το βήμα i ισχύει f = i!.
for i in range(1, n + 1):
    f *= i          # συντομογραφία του f = f * i

# Εκτύπωση του αποτελέσματος.
print('Το παραγοντικό του', n, 'είναι το', f)
```

Πρόγραμμα 15 `prime_simple.py`

```
# Έλεγχος αν ένας ακέραιος είναι πρώτος, με δοκιμαστικές διαιρέσεις.

# Διαβάζουμε από τον χρήστη έναν αριθμό (ως κείμενο) και τον
# μετατρέπουμε σε ακέραιο.
z = int(input("Δῶσε μου ἕναν ἀριθμό: "))

# Υποθέτουμε αρχικά ότι ο z είναι πρώτος, μέχρι να βρεθεί διαιρέτης.
isprime = True

# Δοκιμάζουμε κάθε πιθανό διαιρέτη i από το 2 έως το z-1.
for i in range(2, z):
    # Αν το i διαιρεί ακριβώς τον z (μηδενικό υπόλοιπο),
    # τότε ο z έχει γνήσιο διαιρέτη και δεν είναι πρώτος.
    if z % i == 0:
        isprime = False

# Επιλέγουμε τη λέξη άρνησης ανάλογα με το αποτέλεσμα
# (υπό συνθήκη έκφραση: τιμή_αν_αληθές if συνθήκη else τιμή_αν_ψευδές).
x = '' if isprime else 'δεν'

# Εκτύπωση του αποτελέσματος.
print('Ὁ ἀριθμὸς', z, x, 'εἶναι πρώτος')
```

Πρόγραμμα 16 primetest_odd.py

```
# Έλεγχος αν ένας ακέραιος είναι πρώτος. Βελτίωση της προηγούμενης
# εκδοχής: χειριζόμαστε χωριστά το 2 και τους άρτιους, και
# δοκιμάζουμε μόνο περιττούς διαιρέτες.

# Διαβάζουμε τον αριθμό από τον χρήστη και τον μετατρέπουμε σε ακέραιο.
z = int(input("Δῶσε μου ἕναν ἀριθμό: "))

# Υποθέτουμε αρχικά ότι ο z είναι πρώτος.
isprime = True

# Το 2 είναι πρώτος, άρα δεν χρειάζεται κανένας έλεγχος.
if z != 2:
    if z % 2 != 0:
        # Ο z είναι περιττός, άρα δεν έχει άρτιους διαιρέτες.
        # Δοκιμάζουμε μόνο τους περιττούς i = 3, 5, 7, ..., < z.
        for i in range(3, z, 2):
            if z % i == 0:
                # Βρέθηκε γνήσιος διαιρέτης: ο z δεν είναι πρώτος.
                isprime = False
    else:
        # Ο z είναι άρτιος και διάφορος του 2,
        # άρα διαιρείται από το 2 και δεν είναι πρώτος.
        isprime = False

# Εκτύπωση του αποτελέσματος (True ή False).
print(isprime)
```

Πρόγραμμα 17 prime_sqrt.py

```
import math, time

# Εκτύπωση όλων των πρώτων αριθμών έως το 10000 με δοκιμαστικές
# διαιρέσεις έως την τετραγωνική ρίζα, και μέτρηση του χρόνου.


# Εξετάζουμε μόνο τους περιττούς n = 3, 5, 7, ..., 9999.
for n in range(3, 100000001, 2):
    a = True                # υποθέτουμε ότι ο n είναι πρώτος
    for k in range(3, math.ceil(math.sqrt(n)) + 1, 2):
        if n % k == 0:
            a = False               # βρέθηκε γνήσιος διαιρέτης
    if a :
        print(n, end=' ')
```

Πρόγραμμα 18 `prime_interval.py`

```python
# Έλεγχος αν κάθε αριθμός σε ένα διάστημα είναι πρώτος.
# Εξετάζουμε τους ακεραίους από 1.000.000 έως 1.099.999
# (το range δεν περιλαμβάνει το άνω άκρο).

for z in range(1_000_000, 1_100_000):
    # Για κάθε z υποθέτουμε αρχικά ότι είναι πρώτος.
    isprime = True

    if z % 2 == 0:
        # Ο z είναι άρτιος και μεγαλύτερος του 2, άρα δεν είναι πρώτος.
        # Τυπώνουμε not isprime, δηλαδή False.
        print(z, not isprime)
    else:
        # Ο z είναι περιττός: δοκιμάζουμε μόνο περιττούς διαιρέτες
        # i = 3, 5, 7, ..., έως z // 2. Κανένας γνήσιος διαιρέτης
        # δεν υπερβαίνει το z/2.
        for i in range(3, z // 2 + 1, 2):
            if z % i == 0:
                # Βρέθηκε διαιρέτης: ο z δεν είναι πρώτος.
                isprime = False
        # Τυπώνουμε τον αριθμό και το αποτέλεσμα (True ή False).
        print(z, isprime)
```

Πρόγραμμα 19 `prime_vector.py`

```python
import math
import matplotlib.pyplot as plt

# Βρίσκουμε τους πρώτους έως n και σχεδιάζουμε
# τη γραφική παράσταση «α/α πρώτου → πρώτος».

n = 40000
y = [2]                         # το 2 είναι ο μόνος άρτιος πρώτος

# Εξετάζουμε τους περιττούς c = 3, 5, ..., n.
for c in range(3, n + 1, 2):
    flag = 1                    # σημαία: 1 = «πρώτος», 0 = «σύνθετος»

    # Δοκιμαστικές διαιρέσεις μόνο με περιττούς k = 3, 5, ..., ⌊√c⌋,
    # αφού ο c είναι περιττός.
    for k in range(3, math.isqrt(c) + 1, 2):
        if c % k == 0:
            flag = 0            # βρέθηκε διαιρέτης
            break               # δεν χρειάζεται να συνεχίσουμε

    if flag:
        y.append(c)             # ο c είναι πρώτος· τον κρατάμε

# Οι τετμημένες: ο αύξων αριθμός κάθε πρώτου (1, 2, ..., l),
# ώστε y[i-1] να είναι ο i-οστός πρώτος.
l = len(y)
x = list(range(1, l + 1))
print(f'Πλήθος πρώτων ≤ {n}: {l}')

# Γραφική παράσταση: στο σημείο i ο i-οστός πρώτος.
plt.plot(x, y)
plt.show()
```

Πρόγραμμα 20 `rand.py`

```python
import random

import matplotlib.pyplot as plt

# Νέφος 10000 σημείων με κανονική κατανομή στις δύο συντεταγμένες.
# Οι συντεταγμένες x και y είναι ανεξάρτητες, άρα το νέφος είναι
# κυκλικά συμμετρικό γύρω από το κέντρο του.

x = []      # οι τετμημένες των σημείων
y = []      # οι τεταγμένες των σημείων
for i in range(100000):
    # random.normalvariate(1, 1): κανονική κατανομή με μέση τιμή 1
    # και τυπική απόκλιση 1. Ο πολλαπλασιασμός επί 100 δίνει
    # κανονική κατανομή με μέση τιμή 100 και τυπική απόκλιση 100.
    x.append(100 * random.normalvariate(1, .1))
    y.append(100 * random.normalvariate(1, .1))

# Εκτύπωση των δύο λιστών και του πλήθους των σημείων
# (προσοχή: τυπώνονται 20000 αριθμοί)
print(x, y, len(x))

# Σχεδίαση: κάθε σημείο (x[i], y[i]) ως τελεία, χωρίς γραμμές
plt.plot(x, y, ".")
plt.show()
```

Πρόγραμμα 21 `π_Leibniz.py`

```python
import math

# Υπολογισμός του π με τη σειρά Leibniz:
#   π/4 = 1 - 1/3 + 1/5 - 1/7 + ... = Σ_{k≥1} (-1)^(k+1) / (2k-1).
# Για n = 10, 100, ..., 10^6 υπολογίζουμε το μερικό άθροισμα
# των n πρώτων όρων και συγκρίνουμε την προσέγγιση με το π.

for i in range(1, 7):
    n = 10**i                       # πλήθος όρων της σειράς
    print(f'n = {n}')

    e = 0                           # μερικό άθροισμα (προσέγγιση του π/4)
    for k in range(1, n + 1):
        # Ο k-οστός όρος: εναλλασσόμενο πρόσημο (+, -, +, ...)
        # και παρονομαστής ο k-οστός περιττός αριθμός 2k-1.
        y = (-1)**(k+1) / (2*k - 1)
        e = e + y

    appi = 4 * e                    # προσέγγιση του π

    # Εκτύπωση με 10 δεκαδικά ψηφία: προσέγγιση, π, σφάλμα.
    print(f'{appi:13.10f}, {math.pi:13.10f}, {math.pi - appi:13.10f} ')
```

Πρόγραμμα 22 π_βελάκια.py

```python
import random

# Προσέγγιση του π με τη μέθοδο Monte Carlo:
# ρίχνουμε n βελάκια στο τετράγωνο [-1, 1] x [-1, 1] και μετράμε
# πόσα πέφτουν μέσα στον εγγεγραμμένο κύκλο ακτίνας 1.
# Ο λόγος των εμβαδών κύκλου και τετραγώνου είναι π·1² / 2² = π/4,
# άρα η πιθανότητα ένα βελάκι να πέσει στον κύκλο είναι π/4.

n = 1000000      # πλήθος ρίψεων
hits = 0         # πλήθος επιτυχιών (βελάκια μέσα στον κύκλο)
for i in range(n):
    # ρίχνω το βελάκι i: τυχαίο σημείο (x, y), ομοιόμορφα στο τετράγωνο
    x = random.uniform(-1, 1)
    y = random.uniform(-1, 1)
    # ελέγχω αν είναι επιτυχία: το σημείο απέχει από το (0, 0)
    # απόσταση το πολύ 1 (συγκρίνουμε τετράγωνα, χωρίς ρίζα)
    if x**2 + y**2 <= 1:
        hits += 1

# Η σχετική συχνότητα hits/n προσεγγίζει το π/4,
# οπότε το 4·hits/n προσεγγίζει το π.
my_pi = 4 * hits / n
print(my_pi)
```

Πρόγραμμα 23 π_Buffon.py

```python
import math, random

# Προσομοίωση του προβλήματος της βελόνας του Buffon:
# ρίχνουμε τυχαία μια βελόνα μήκους 1 σε επίπεδο με παράλληλες
# οριζόντιες ευθείες σε απόσταση 1 (τις ευθείες y = ακέραιος).
# Η πιθανότητα η βελόνα να τέμνει κάποια ευθεία είναι 2/π,
# οπότε από τη σχετική συχνότητα τομών εκτιμούμε το π.

drops = 1000000
# φορὲς ῥίψης
success = 0      # πλήθος ρίψεων στις οποίες η βελόνα τέμνει ευθεία

for i in range(1, drops + 1):
    # Το ύψος του ενός άκρου της βελόνας, ομοιόμορφα στο [0, 1).
    # Λόγω περιοδικότητας αρκεί να εξετάσουμε μία μόνο λωρίδα.
    y = random.random()

    # Η γωνία της βελόνας ως προς τις ευθείες, ομοιόμορφα στο [0, 2π).
    a = random.random() * 2 * math.pi

    # Το ύψος του άλλου άκρου (η βελόνα έχει μήκος 1).
    # Μας ενδιαφέρει μόνο η κατακόρυφη συνιστώσα sin(a).
    y1 = y + math.sin(a)

    # Η βελόνα τέμνει ευθεία ακριβώς όταν υπάρχει ακέραιος στο
    # διάστημα [min(y, y1), max(y, y1)]. Αφού το διάστημα έχει μήκος
    # το πολύ 1, αυτό ισοδυναμεί με floor(max) == ceil(min).
    # Η Python μετατρέπει το True/False σε 1/0 κατά την πρόσθεση.
    success = success + (math.floor(max(y, y1)) == math.ceil(min(y, y1)))

# Από success/drops ≈ 2/π προκύπτει π ≈ 2·drops/success.
mypi = 2*drops / success

print(f'mypi = {mypi}')        # η εκτίμηση του π
print(f'pi = {math.pi}')       # η πραγματική τιμή
print(f'{math.pi - mypi}')     # το σφάλμα της εκτίμησης
```

Πρόγραμμα 24 π_πλέγμα.py

```
import math

# Προσέγγιση του π με μέτρηση σημείων πλέγματος (ακέραιων σημείων)
# μέσα σε κύκλο ακτίνας w με κέντρο την αρχή των αξόνων.
# Μετράμε τα σημεία (k, j) με k, j ≥ 1 και k² + j² ≤ w²,
# δηλαδή τα σημεία του πρώτου τεταρτημορίου εκτός των αξόνων.

w = int(input('Ἀκτίνα κύκλου: '))

n1 = 0                          # πλήθος σημείων στο πρώτο τεταρτημόριο
for k in range(1, w + 1):
    # Για σταθερή τετμημένη k, οι επιτρεπτές τεταγμένες j ≥ 1 είναι
    # αυτές με j ≤ √(w² - k²), δηλαδή m = ⌊√(w² - k²)⌋ το πλήθος.
    m = math.floor(math.sqrt(w**2 - k**2))
    n1 = n1 + m

# Κάθε σημείο (k, j) είναι η πάνω δεξιά κορυφή του μοναδιαίου
# τετραγώνου [k-1, k] × [j-1, j], που βρίσκεται εξ ολοκλήρου
# μέσα στον κύκλο. Ἄρα 4·n1 τετράγωνα (και στα τέσσερα τεταρτημόρια)
# καλύπτουν σχεδόν όλο το εμβαδόν π·w², και 4·n1 / w² ≈ π.
rho_w = 4*n1 / w**2

# Εκτύπωση της ακτίνας, της προσέγγισης και του απόλυτου σφάλματος.
print(f'w      ={w:8d}')
print(f'rho_w  ={rho_w:18.12f}')
print(f'Σφάλμα ={abs(math.pi - rho_w):18.12f}')
```

Πρόγραμμα 25 1_simple_while.py

```
# Επανάληψη ερώτησης μέχρι ο χρήστης να δώσει θετικό αριθμό

x = int(input('Δῶστε θετικό αριθμό: '))
while x <= 0:
    x = int(input('Ὁ ἀριθμὸς ποὺ δώσατε δὲν εἶναι θετικός!\nΠαρακαλώ δώστε θετικό αριθμό: '))
print(x)
```

Πρόγραμμα 26 2_mypr.py

```
import math

# Έλεγχος αν ένας αριθμός είναι πρώτος, με βρόχο while μέχρι τη ρίζα του

r = int(input('Please give me a number: '))
isprime = True
i = 2
while i <= math.ceil(math.sqrt(r)):
    if r % i == 0 and r != 2:
        isprime = False
    i += 1
print(isprime)
```

Πρόγραμμα 27 3_primewhile.py

```
import math

# Έλεγχος αν ένας θετικός αριθμός είναι πρώτος, με χωριστή αντιμετώπιση
# των αρτίων και του 2.

r = int(input('Παρακαλῶ δῶστε μου ἕναν ἀριθμό: '))
while r < 1:
    r = int(input('Ὁ ἀριθμὸς ποὺ δώσατε δὲν εἶναι μεγαλύτερος τοῦ 1!\nΠαρακαλῶ δῶστε σωστὸ ἀριθμό: '))

if r > 4 and r % 2 == 1:
    i = 3
    c = math.ceil(math.sqrt(r))
    while r % i != 0 and i <= c:
        i += 2
    isprime = i > c
elif r != 2 and r % 2 == 0:
    isprime = False
else:
    isprime = True

print(isprime)
```

Πρόγραμμα 28 5_primeall.py

```
import math

import matplotlib.pyplot as plt

# a[r] = True αν ο r+1 είναι πρώτος (το 1 δὲν εἶναι πρῶτος).
# Στη συνέχεια το y κρατά το πλήθος των πρώτων μέχρι κάθε θέση (σωρευτικά).

a = [False, True, True, False]
for r in range(5, 1001):
    i = 3
    c = math.ceil(math.sqrt(r))
    while r % i != 0 and i <= c:
        i += 2
    a.append(i > c and r % 2 == 1)

y = [0, 1]
for i in range(2, len(a)):
    w = y[i - 1]
    if a[i]:
        y.append(w + 1)
    else:
        y.append(w)

# Ἡ λίστα τῶν πρώτων ἀριθμῶν, ξεκινῶντας ἀπὸ τὸ 2.
p = []
for i in range(len(a)):
    if a[i]:
        p.append(i + 1)
print(p)

# plt.plot(y)
# plt.show()
```

Πρόγραμμα 29 6_primeex2.py

```
import matplotlib.pyplot as plt

# Για κάθε r στο [3, z) μετράμε πόσους διαιρέτες d έχει (πέρα από το 1)
# ελέγχοντας όλα τα i από 2 έως r-1.

z = int(input('Please give me a number: '))
count = 0
a = [0, 0, 0]
for r in range(3, z):
    i = 2
    isprime = True
    d = 0
    while i <= r - 1:
        if r % i == 0:
            isprime = False
            d += 1
        i += 1
        count += isprime
    a.append(d)

print("The vector is:", a)

plt.plot(a)
plt.show()
```