Python: Σύντομη Εισαγωγή

Επιμέλεια: Α.Δρακόπουλος

Περιεχόμενα

1. Είσοδος Δεδομένων από το Πληκτρολόγιο (Input)

Στην Python, η είσοδος δεδομένων από τον χρήστη (πληκτρολόγιο) γίνεται με τη χρήση της ενσωματωμένης συνάρτησης **input()**.

Ο Τρόπος Χρήσης

Η συνάρτηση input():

1. **Σταματά** την εκτέλεση του προγράμματος.
2. **Προβάλλει** στην οθόνη ένα προαιρετικό μήνυμα (prompt) που δίνουμε ως όρισμα.
3. **Περιμένει** τον χρήστη να πληκτρολογήσει δεδομένα και να πατήσει το πλήκτρο Enter.
4. **Επιστρέφει** τα δεδομένα που πληκτρολογήθηκαν πάντα ως **συμβολοσειρά** (str).

Σημαντικό: Ο Τύπος Δεδομένων

Ακόμα κι αν ο χρήστης πληκτρολογήσει έναν αριθμό, η input() τον επιστρέφει ως str. Για να τον χρησιμοποιήσεις ως αριθμό (π.χ., για πράξεις), πρέπει να γίνει **μετατροπή τύπου (type casting)**, συνήθως με τις συναρτήσεις int() ή float().

Παράδειγμα Χρήσης & Κώδικας

```
# Παράδειγμα 1: Απλή είσοδος με μήνυμα (prompt)
name = input("Ποιο είναι το όνομά σου; ")
print(f"Χαίρομαι που σε γνωρίζω, {name}!")

# Παράδειγμα 2: Είσοδος αριθμού και μετατροπή τύπου
age_str = input("Πόσο χρονών είσαι; ")
# Μετατροπή της συμβολοσειράς σε ακέραιο (int)
try:
    age = int(age_str)
    next_age = age + 1
    print(f"Του χρόνου θα είσαι {next_age} ετών.")
except ValueError:
    print("Σφάλμα: Παρακαλώ εισάγετε έναν ακέραιο αριθμό για την ηλικία.")
```

2. Έξοδος Δεδομένων στην Οθόνη (Output)

Η έξοδος δεδομένων (π.χ., εμφάνιση αποτελεσμάτων, μηνυμάτων, τιμών μεταβλητών) στην οθόνη γίνεται με τη χρήση της ενσωματωμένης συνάρτησης **print()**.

Ο Τρόπος Χρήσης

Η συνάρτηση print():

1. Μπορεί να δεχτεί **ένα ή περισσότερα ορίσματα** (τιμές ή μεταβλητές).
2. Προβάλλει τα ορίσματα στην τυπική έξοδο (συνήθως η οθόνη της κονσόλας).
3. **Αυτόματα** εισάγει ένα **διάστημα** (' ') ανάμεσα στα ορίσματα (με το προαιρετικό όρισμα sep).
4. **Αυτόματα** προσθέτει μια αλλαγή γραμμής ('\n') στο τέλος (με το προαιρετικό όρισμα end).

Προαιρετικά Ορίσματα της print()

Όρισμα	Προεπιλεγμένη Τιμή	Περιγραφή
sep	' ' **(ένα κενό)**	**Το διαχωριστικό ανάμεσα στα πολλαπλά στοιχεία που εκτυπώνονται.**
end	'\n' **(αλλαγή γραμμής)**	**Ο χαρακτήρας που προστίθεται στο τέλος της εξόδου.**

Παράδειγμα Χρήσης & Κώδικας

```
# Παράδειγμα 1: Απλή εκτύπωση
message = "Η Python είναι εύκολη."
result = 10 * 5
print(message)
print(result)

# Παράδειγμα 2: Εκτύπωση πολλαπλών ορισμάτων
num1 = 10
num2 = 20
print("Οι αριθμοί είναι:", num1, num2) # Αυτόματα βάζει κενό ανάμεσα στα ορίσματα

# Παράδειγμα 3: Χρήση του 'sep' και 'end'
```

```
print(num1, num2, 30, sep=' | ', end=' *** ') # Χρησιμοποιεί '|' ως διαχωριστικό
print("Τέλος.") # Εμφανίζεται στην ίδια γραμμή λόγω του end=' *** '
```

Χρήση F-strings (Format Strings)

Ο πιο σύγχρονος και ευανάγνωστος τρόπος για να συνδυάσεις κείμενο και μεταβλητές στην έξοδο είναι τα **F-strings** (formatted string literals):

```
name = "Maria"
age = 28
weight = 65.5

# Χρήση F-string
print(f"Το όνομα είναι {name}, η ηλικία είναι {age} και το βάρος είναι {weight:.1f} κιλά.")
# Σημείωση: Το :.1f μορφοποιεί τον float με 1 δεκαδικό ψηφίο.
```

3.Εισαγωγή στους Τύπους Δεδομένων της Python

Οι **Τύποι Δεδομένων (Data Types)** στην Python καθορίζουν τον τύπο των τιμών που μπορούν να αποθηκευτούν σε μια μεταβλητή. Η Python είναι μια γλώσσα με **δυναμική τυποποίηση (dynamically typed)**, που σημαίνει ότι δεν χρειάζεται να δηλώσεις τον τύπο μιας μεταβλητής κατά τη δημιουργία της – ο τύπος καθορίζεται αυτόματα από την τιμή που της εκχωρείται.

Οι βασικότεροι ενσωματωμένοι τύποι δεδομένων είναι:

1. **Αριθμητικοί (Numeric Types):** int, float, complex
2. **Ακολουθίες (Sequence Types):** str, list, tuple, range
3. **Χαρτογραφήσεις (Mapping Type):** dict
4. **Σύνολα (Set Types):** set, frozenset
5. **Λογικοί (Boolean Type):** bool
6. **None Type:** NoneType

Αναλυτική Περιγραφή Βασικών Τύπων Δεδομένων

Ακολουθεί μια αναλυτική περιγραφή των πιο συχνά χρησιμοποιούμενων τύπων, με **όρια**, βασικές **μεθόδους** και **παραδείγματα χρήσης**.

1. Ακέραιοι Αριθμοί (int)

Χαρακτηριστικό	**Περιγραφή**
Τρόπος Χρήσης	**Χρησιμοποιείται για την αναπαράσταση ακέραιων αριθμών (θετικών, αρνητικών, μηδέν).**
Όρια	**Θεωρητικά απεριόριστο (περιορίζεται μόνο από τη διαθέσιμη μνήμη του συστήματος).**
Βασικές Μέθοδοι	int.from_bytes(), int.to_bytes(), int.bit_length()
Αμετάβλητος	**Ναι (Immutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
age = 30
big_number = 123456789012345678901234567890
print(f"Ηλικία: {age}, Τύπος: {type(age)}")
print(f"Bit Length: {big_number.bit_length()}") # Επιστρέφει τα bits που χρειάζονται
```

2. Δεκαδικοί Αριθμοί Κινητής Υποδιαστολής (float)

Χαρακτηριστικό	**Περιγραφή**
Τρόπος Χρήσης	**Χρησιμοποιείται για την αναπαράσταση πραγματικών αριθμών (με δεκαδικό μέρος).**
Όρια	**Συνήθως διπλής ακρίβειας (64-bit), με ακρίβεια περίπου 15-17 δεκαδικών ψηφίων και εύρος 10^{-308} έως 10^{308}.**
Βασικές Μέθοδοι	float.as_integer_ratio(), float.is_integer(), float.hex()
Αμετάβλητος	**Ναι (Immutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
pi_value = 3.14159
percentage = 0.75
print(f"Τιμή Pi: {pi_value}, Τύπος: {type(pi_value)}")
print(f"Είναι ακέραιος: {pi_value.is_integer()}") # Επιστρέφει False
```

3. Συμβολοσειρές (str)

Χαρακτηριστικό	**Περιγραφή**
Τρόπος Χρήσης	**Χρησιμοποιείται για την αναπαράσταση κειμένου, μια ακολουθία χαρακτήρων σε εισαγωγικά (' ' ή " ").**
Όρια	**Περιορίζεται από τη διαθέσιμη μνήμη.**
Βασικές Μέθοδοι	str.lower(), str.upper(), str.strip(), str.split(), str.replace(), str.find(), str.join()
Αμετάβλητος	**Ναι (Immutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
greeting = "Hello Python World!"
length = len(greeting) # Μήκος συμβολοσειράς
upper_case = greeting.upper()
print(f"Αρχική: {greeting}, Μήκος: {length}")
print(f"Κεφαλαία: {upper_case}")
```

4. Λίστες (list)

Χαρακτηριστικό	**Περιγραφή**
Τρόπος Χρήσης	**Μια διατεταγμένη και μεταβλητή ακολουθία στοιχείων, τα οποία μπορούν να είναι οποιουδήποτε τύπου. Δηλώνονται με τετράγωνες αγκύλες** [ ]**.**
Όρια	**Περιορίζεται από τη διαθέσιμη μνήμη.**
Βασικές Μέθοδοι	list.append(), list.insert(), list.remove(), list.pop(), list.sort(), list.reverse(), list.extend()
Αμετάβλητος	**Όχι (Μεταβλητός - Mutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange") # Προσθήκη στο τέλος
first_fruit = fruits[0] # Πρόσβαση στοιχείου
print(f"Λίστα: {fruits}")
print(f"Πρώτο στοιχείο: {first_fruit}")
```

5. Πλειάδες (tuple)

Χαρακτηριστικό	**Περιγραφή**
Τρόπος Χρήσης	**Μια διατεταγμένη ακολουθία στοιχείων, όπως οι λίστες, αλλά είναι αμετάβλητη. Δηλώνονται με παρενθέσεις** ( )**.**
Όρια	**Περιορίζεται από τη διαθέσιμη μνήμη.**
Βασικές Μέθοδοι	tuple.count(), tuple.index()
Αμετάβλητος	**Ναι (Immutable)**

Χαρακτηριστικό	Περιγραφή

Παράδειγμα Χρήσης & Κώδικας:

```
coordinates = (10.0, 20.0)
x_coord = coordinates[0]
# coordinates.append(30.0) # Αυτό θα προκαλούσε σφάλμα (TypeError)
print(f"Πλειάδα: {coordinates}")
print(f"Συντεταγμένη x: {x_coord}")
```

6. Λεξικά (dict)

Χαρακτηριστικό	Περιγραφή
Τρόπος Χρήσης	**Μια άτακτη (μέχρι την Python 3.7+ όπου είναι διατεταγμένη) συλλογή ζευγών κλειδιού-τιμής (key-value pairs). Δηλώνονται με άγκιστρα** { }**. Τα κλειδιά πρέπει να είναι αμετάβλητα (π.χ.** str**,** int**,** tuple**).**
Όρια	**Περιορίζεται από τη διαθέσιμη μνήμη.**
Βασικές Μέθοδοι	dict.keys()**,** dict.values()**,** dict.items()**,** dict.get()**,** dict.pop()**,** dict.update()
Αμετάβλητος	**Όχι (Μεταβλητός - Mutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
person = {"name": "Alice", "age": 25, "city": "London"}
person["age"] = 26 # Αλλαγή τιμής
city = person.get("city")
print(f"Λεξικό: {person}")
print(f"Πόλη: {city}")
```

7. Σύνολα (set)

Χαρακτηριστικό	Περιγραφή
Τρόπος Χρήσης	**Μια άτακτη συλλογή μοναδικών και αμετάβλητων στοιχείων. Χρησιμοποιείται για γρήγορους ελέγχους ύπαρξης και μαθηματικές πράξεις συνόλων. Δηλώνονται με άγκιστρα** { } **ή τη συνάρτηση** set()**.**
Όρια	**Περιορίζεται από τη διαθέσιμη μνήμη.**
Βασικές Μέθοδοι	set.add()**,** set.remove()**,** set.union()**,** set.intersection()**,** set.difference()**,** set.issubset()
Αμετάβλητος	**Όχι (Μεταβλητός - Mutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
numbers = {1, 2, 3, 3, 4} # Το 3 θα αποθηκευτεί μόνο μία φορά
numbers.add(5)
is_member = 2 in numbers
print(f"Σύνολο: {numbers}")
print(f"Το 2 είναι μέλος: {is_member}")
```

8. Λογικός Τύπος (bool)

Χαρακτηριστικό	Περιγραφή
Τρόπος Χρήσης	**Αναπαριστά λογικές τιμές Αληθές (True) ή Ψευδές (False).**
Όρια	**Δύο μόνο τιμές:** True **και** False**.**
Βασικές Μέθοδοι	**Δεν έχει ειδικές μεθόδους. Χρησιμοποιείται σε λογικές εκφράσεις και δομές ελέγχου.**
Αμετάβλητος	**Ναι (Immutable)**

Παράδειγμα Χρήσης & Κώδικας:

```
is_adult = True
```

```
is_raining = (5 > 10) # is_raining θα γίνει False
if is_adult:
        print("Είναι ενήλικος.")
print(f"Τύπος is_adult: {type(is_adult)}")
```

4.Δομές Επιλογής στην Python

Οι **δομές επιλογής (Selection Structures)** επιτρέπουν στο πρόγραμμα να εκτελεί διαφορετικά τμήματα κώδικα ανάλογα με το αν μια δεδομένη συνθήκη είναι **Αληθής (`True`)** ή **Ψευδής (`False`)**.

Οι βασικές δομές επιλογής στην Python είναι:

1. **`if` (Απλή Επιλογή)**
2. **`if-else` (Διπλή Επιλογή)**
3. **`if-elif-else` (Πολλαπλή Επιλογή)**
4. **Ενσωματωμένος Τελεστής Ternary** (Σύντομη μορφή `if-else`)

1. Η Δομή `if` (Απλή Επιλογή)

Η δομή `if` είναι η πιο βασική δομή επιλογής. Εκτελεί ένα μπλοκ κώδικα **μόνο** αν η συνθήκη που δίνεται είναι **Αληθής**.

Χαρακτηριστικό	Περιγραφή
Σύνταξη	`if συνθήκη:`
Περίπτωση Χρήσης	Όταν θέλεις να εκτελέσεις ένα **μοναδικό** σετ εντολών βάσει μιας συνθήκης, χωρίς να υπάρχει εναλλακτική ενέργεια αν η συνθήκη είναι ψευδής.

Παράδειγμα Χρήσης & Κώδικας:

Έλεγχος αν ένας αριθμός είναι θετικός.

```
score = 95

if score > 90:
    # Αυτό το μπλοκ κώδικα εκτελείται ΜΟΝΟ αν το score είναι μεγαλύτερο από 90
    print("Συγχαρητήρια! Πήρες Άριστα.")
    print("Η επίδοσή σου είναι εξαιρετική.")

print("Ο έλεγχος ολοκληρώθηκε.")
```

2. Η Δομή `if-else` (Διπλή Επιλογή)

Η δομή `if-else` προσφέρει δύο μονοπάτια εκτέλεσης. Εκτελεί το μπλοκ `if` αν η συνθήκη είναι **Αληθής** και το μπλοκ `else` αν η συνθήκη είναι **Ψευδής**.

Χαρακτηριστικό	Περιγραφή
Σύνταξη	`if συνθήκη: ... else: ...`
Περίπτωση Χρήσης	Όταν πρέπει να επιλέξεις ανάμεσα σε **δύο αμοιβαία αποκλειόμενες** ενέργειες (ή «αυτό» ή «το άλλο»).

Παράδειγμα Χρήσης & Κώδικας:

Έλεγχος αν ένας αριθμός είναι άρτιος ή περιττός.

```
number = 15

if number % 2 == 0:
    # Εκτελείται αν το υπόλοιπο της διαίρεσης με το 2 είναι 0 (Άρτιος)
    print(f"Ο αριθμός {number} είναι άρτιος.")
else:
    # Εκτελείται αν η συνθήκη του if είναι False (Περιττός)
    print(f"Ο αριθμός {number} είναι περιττός.")
```

3. Η Δομή `if-elif-else` (Πολλαπλή Επιλογή)

Η δομή `if-elif-else` χρησιμοποιείται για τον έλεγχο **πολλαπλών** συνθηκών διαδοχικά. Η Python ελέγχει τις συνθήκες με τη σειρά: μόλις βρει την **πρώτη** συνθήκη που είναι **Αληθής**, εκτελεί το αντίστοιχο μπλοκ κώδικα και **παραλείπει** όλες τις υπόλοιπες `elif` και το τελικό `else`.

Χαρακτηριστικό	Περιγραφή
Σύνταξη	`if συνθήκη1: ... elif συνθήκη2: ... elif συνθήκηΝ: ... else: ...`
Περίπτωση Χρήσης	Όταν πρέπει να επιλέξεις ανάμεσα σε **τρία ή περισσότερα** μονοπάτια εκτέλεσης, όπου μόνο **ένα** μπορεί να είναι σωστό.

Παράδειγμα Χρήσης & Κώδικας:

Καθορισμός βαθμού με βάση τη βαθμολογία.

```
mark = 78

if mark >= 90:
    grade = "A"
elif mark >= 80:
    grade = "B"
elif mark >= 70:
    grade = "C"
elif mark >= 60:
    grade = "D"
else:
    # Εάν καμία από τις παραπάνω συνθήκες δεν είναι True
    grade = "F"

print(f"Για βαθμολογία {mark}, ο βαθμός είναι: {grade}")
```

4. Ενσωματωμένος Τελεστής Ternary (Ternary Conditional Operator)

Αυτή η δομή επιτρέπει τη **συμπυκνωμένη** έκφραση μιας απλής `if-else` σε **μία γραμμή**.
Χρησιμοποιείται κυρίως για την άμεση εκχώρηση μιας τιμής σε μια μεταβλητή βάσει μιας συνθήκης.

Χαρακτηριστικό	Περιγραφή
Σύνταξη	`τιμή_αν_true if συνθήκη else τιμή_αν_false`
Περίπτωση Χρήσης	Όταν χρειάζεται να εκχωρήσεις μια τιμή σε μια μεταβλητή με βάση μια απλή δυαδική συνθήκη (`True`/`False`), κάνοντας τον κώδικα πιο σύντομο και ευανάγνωστο (για απλές περιπτώσεις).

Παράδειγμα Χρήσης & Κώδικας:

Αναγνώριση κατάστασης με βάση την ισορροπία.

```
balance = 1500

# Αν η balance > 0, η κατάσταση είναι "Θετική", αλλιώς "Αρνητική"
status = "Θετική" if balance > 0 else "Αρνητική"

print(f"Η ισορροπία είναι {balance}€, άρα η κατάσταση είναι: {status}")
```

5.Δομές Επανάληψης στην Python

Οι **Δομές Επανάληψης (Iteration/Loop Structures)** επιτρέπουν την εκτέλεση ενός μπλοκ κώδικα **πολλές φορές** μέχρι να ικανοποιηθεί μια συγκεκριμένη συνθήκη.

Οι δύο βασικές δομές επανάληψης στην Python είναι:

1. **for loop (Βρόχος μετρητή ή βρόχος διαδοχής)**
2. **while loop (Βρόχος συνθήκης)**

1. Η Δομή for (Βρόχος Διαδοχής)

Ο βρόχος for χρησιμοποιείται για την επανάληψη σε μια **ακολουθία** (όπως μια λίστα, πλειάδα, συμβολοσειρά, ή εύρος αριθμών) ή οποιοδήποτε **επαναληπτικό αντικείμενο (iterable)**.

Χαρακτηριστικό	**Περιγραφή**
Σύνταξη	for μεταβλητή_στοιχείου in επαναληπτικό_αντικείμενο:
Περίπτωση Χρήσης	**Όταν γνωρίζεις εκ των προτέρων ή μπορείς να καθορίσεις τον αριθμό των επαναλήψεων (π.χ., επανάληψη για κάθε στοιχείο μιας λίστας, ή για ένα συγκεκριμένο εύρος τιμών).**

Παράδειγμα 1: Επανάληψη σε Λίστα

Χρήση: Εμφάνιση κάθε στοιχείου μιας λίστας.

```
fruits = ["μήλο", "μπανάνα", "κεράσι"]

print("Λίστα Φρούτων:")
for fruit in fruits:
    print(f"- {fruit}")
```

Παράδειγμα 2: Επανάληψη σε Εύρος Αριθμών (range())

Η συνάρτηση range(start, stop, step) χρησιμοποιείται για την επανάληψη έναν **συγκεκριμένο αριθμό φορών**.

Χρήση: Εκτύπωση των αριθμών από 0 έως 4.

```
# Το range(5) παράγει τους αριθμούς 0, 1, 2, 3, 4
print("\nΑριθμοί από 0 έως 4:")
for i in range(5):
    print(i)
```

Χρήση: Εκτύπωση των άρτιων αριθμών από 2 έως 10.

```
# range(2, 11, 2) παράγει 2, 4, 6, 8, 10
print("\nΆρτιοι αριθμοί από 2 έως 10:")
for num in range(2, 11, 2):
    print(num)
```

2. Η Δομή while (Βρόχος Συνθήκης)

Ο βρόχος while εκτελεί ένα μπλοκ κώδικα **εφόσον** μια δεδομένη **λογική συνθήκη** παραμένει **Αληθής (True)**.

Χαρακτηριστικό	**Περιγραφή**
Σύνταξη	while συνθήκη:
Περίπτωση Χρήσης	**Όταν δεν γνωρίζεις εκ των προτέρων πόσες φορές πρέπει να γίνει η επανάληψη. Χρησιμοποιείται για να επαναλαμβάνεται μια διαδικασία μέχρι να ικανοποιηθεί μια εξωτερική συνθήκη (π.χ., ο χρήστης να εισάγει έγκυρα δεδομένα, ή να μηδενιστεί ένας μετρητής).**
Προσοχή	**Η συνθήκη πρέπει τελικά να γίνει** False **για να αποφευχθεί ένας ατέρμων βρόχος (infinite loop).**

Παράδειγμα 1: Επανάληψη με Μετρητή

Χρήση: Καταμέτρηση από το 1 έως το 3.

```
count = 1

print("Μετρητής:")
while count <= 3:
    print(f"Τρέχων μετρητής: {count}")
    count += 1 # ΣΗΜΑΝΤΙΚΟ: Αύξηση του μετρητή για να τερματίσει ο βρόχος

print("Η καταμέτρηση τελείωσε.")
```

Παράδειγμα 2: Επανάληψη για Έγκυρη Είσοδο Χρήστη

Χρήση: Ζητάει από τον χρήστη μια θετική βαθμολογία μέχρι να την εισάγει σωστά.

```
score = -1 # Αρχική τιμή που κάνει τη συνθήκη True

while score < 0 or score > 100:
    # Ζητάμε είσοδο μέσα στον βρόχο
    score_str = input("Εισάγετε βαθμολογία (0-100): ")
    try:
        score = int(score_str)
        if score < 0 or score > 100:
            print("Λάθος εύρος. Δοκιμάστε ξανά.")
    except ValueError:
        print("Μη έγκυρη εισαγωγή. Παρακαλώ εισάγετε αριθμό.")
        score = -1 # Επαναφορά για να συνεχιστεί ο βρόχος

print(f"Εισήχθη έγκυρη βαθμολογία: {score}")
```

Εντολές Ελέγχου Ροής Βρόχων

Μέσα στις δομές επανάληψης, μπορούμε να χρησιμοποιήσουμε ειδικές εντολές για να τροποποιήσουμε τη ροή εκτέλεσης:

1. break

Χαρακτηριστικό	**Περιγραφή**
Χρήση	**Τερματίζει αμέσως τον τρέχοντα βρόχο (**for **ή** while**) και η εκτέλεση συνεχίζεται στην πρώτη εντολή μετά τον βρόχο.**

Χαρακτηριστικό	Περιγραφή
Περίπτωση Χρήσης	**Όταν έχει βρεθεί το ζητούμενο στοιχείο ή έχει συμβεί μια κρίσιμη συνθήκη και δεν χρειάζεται να συνεχιστεί η επανάληψη.**

Παράδειγμα:

```
# Ψάχνουμε τον αριθμό 7
numbers = [1, 4, 7, 10, 13]

for n in numbers:
    if n == 7:
        print(f"Βρέθηκε ο αριθμός 7!")
        break # Τερματίζει τον βρόχο αμέσως
    print(f"Επεξεργασία του: {n}")
```

2. continue

Χαρακτηριστικό	Περιγραφή
Χρήση	**Παραλείπει την υπόλοιπη εκτέλεση του τρέχοντος σώματος του βρόχου και μεταπηδά στην επόμενη επανάληψη.**
Περίπτωση Χρήσης	**Όταν θέλεις να αγνοήσεις ορισμένα στοιχεία ή συνθήκες και να προχωρήσεις αμέσως στο επόμενο βήμα του βρόχου.**

Παράδειγμα:

```
# Εκτυπώνουμε μόνο τους περιττούς αριθμούς
for x in range(1, 6): # x = 1, 2, 3, 4, 5
    if x % 2 == 0:
        continue # Αν είναι άρτιος, πήγαινε στο επόμενο x
    print(f"Περιττός αριθμός: {x}")
```

6.Δημιουργία Συναρτήσεων στην Python

Η δημιουργία μιας συνάρτησης στην Python γίνεται χρησιμοποιώντας την εντολή **`def`** (define).

Βασική Σύνταξη

```
def function_name(parameter1, parameter2, ...):
    """Docstring: Εδώ περιγράφεις τι κάνει η συνάρτηση."""
    # Σώμα της συνάρτησης (εντολές)
    result = parameter1 + parameter2
    return result # Προαιρετικά, επιστρέφει μια τιμή
```

1. **`def`**: Λέξη-κλειδί για τη δήλωση της συνάρτησης.
2. **`function_name`**: Το όνομα της συνάρτησης (πρέπει να ακολουθεί τους κανόνες ονοματοδοσίας).
3. **Παράμετροι (Parameters)**: Οι μεταβλητές που δέχεται η συνάρτηση ως είσοδο (μέσα στις παρενθέσεις). Είναι προαιρετικές.
4. **`:`**: Τερματίζει τη γραμμή δήλωσης.
5. **Εσοχή (Indentation)**: Όλες οι εντολές που ανήκουν στο σώμα της συνάρτησης πρέπει να έχουν **ίδια εσοχή**.
6. **`return`**: Προαιρετική εντολή που χρησιμοποιείται για την **επιστροφή** μιας τιμής (ή αντικειμένου) από τη συνάρτηση. Αν παραλειφθεί, η συνάρτηση επιστρέφει αυτόματα `None`.

Πλεονεκτήματα των Συναρτήσεων

Οι συναρτήσεις είναι θεμελιώδεις στον προγραμματισμό και προσφέρουν σημαντικά οφέλη:

1. **Επαναχρησιμοποίηση Κώδικα (Code Reusability):** Γράφεις ένα μπλοκ κώδικα μία φορά και το καλείς όσες φορές χρειάζεται από διαφορετικά σημεία του προγράμματος.
2. **Μείωση Πολυπλοκότητας (Reduced Complexity):** Χωρίζοντας ένα μεγάλο πρόβλημα σε μικρότερες, διαχειρίσιμες λειτουργίες.
3. **Βελτιωμένη Αναγνωσιμότητα (Improved Readability):** Ο κώδικας γίνεται πιο δομημένος και το όνομα της συνάρτησης περιγράφει τον σκοπό της.
4. **Ευκολότερη Αποσφαλμάτωση (Easier Debugging):** Όταν προκύψει σφάλμα, μπορείς να εστιάσεις μόνο στον κώδικα της συγκεκριμένης συνάρτησης.

Υποστήριξη Modularity (Δομοποίηση)

Το **Modularity** (Δομοποίηση) είναι η αρχή του να χωρίζεις ένα πρόγραμμα σε ανεξάρτητα, διακριτά και ανταλλάξιμα τμήματα (δομικές μονάδες ή modules).

Οι συναρτήσεις υποστηρίζουν το modularity επειδή:

- **Εγκλεισμός Λογικής (Logic Encapsulation):** Κάθε συνάρτηση είναι μια αυτόνομη μονάδα που εκτελεί μια συγκεκριμένη εργασία.
- **Modules & Packages:** Το modularity επεκτείνεται περαιτέρω στην Python με τα **Modules** (αρχεία `.py` που περιέχουν συναρτήσεις) και τα **Packages** (συλλογές modules), επιτρέποντας την οργάνωση του κώδικα σε μεγάλα έργα.

Παράδειγμα Modularity με Modules

```
# Έστω ότι έχουμε ένα αρχείο με όνομα 'math_ops.py' (το module μας)

# math_ops.py:
def add(a, b):
    """Προσθέτει δύο αριθμούς."""
    return a + b

def multiply(a, b):
    """Πολλαπλασιάζει δύο αριθμούς."""
    return a * b

# Κυρίως Πρόγραμμα (main_app.py):
import math_ops as mo # Εισάγουμε το module

num1 = 10
num2 = 5

sum_result = mo.add(num1, num2) # Καλούμε τη συνάρτηση από το module
prod_result = mo.multiply(num1, num2)

print(f"Άθροισμα: {sum_result}")
print(f"Γινόμενο: {prod_result}")
```

Είδη Συναρτήσεων στην Python με Παραδείγματα (Απλή κατηγοριοποίηση)

Οι συναρτήσεις στην Python κατηγοριοποιούνται συνήθως με βάση το αν δέχονται ορίσματα και αν επιστρέφουν τιμή.

1. Συνάρτηση χωρίς Παραμέτρους και χωρίς Επιστρεφόμενη Τιμή

Χρήση: Για την εκτέλεση μιας απλής ενέργειας ή την εκτύπωση ενός μηνύματος.

```
def greet():
    """Εμφανίζει ένα απλό χαιρετισμό."""
    print("Καλώς ήρθατε στο Python!")

# Κλήση της συνάρτησης
greet()
# Εκτύπωση: Καλώς ήρθατε στο Python!
```

2. Συνάρτηση με Παραμέτρους, αλλά χωρίς Επιστρεφόμενη Τιμή

Χρήση: Για την εκτέλεση μιας ενέργειας που εξαρτάται από δεδομένα εισόδου.

```
def print_area(length, width):
    """Υπολογίζει και εκτυπώνει το εμβαδόν ενός ορθογωνίου."""
    area = length * width
    print(f"Το εμβαδόν είναι: {area} τετραγωνικά.")

# Κλήση της συνάρτησης με ορίσματα (arguments)
print_area(5, 12)
# Εκτύπωση: Το εμβαδόν είναι: 60 τετραγωνικά.
```

3. Συνάρτηση με Παραμέτρους και Επιστρεφόμενη Τιμή

Χρήση: Για την εκτέλεση ενός υπολογισμού και την επιστροφή του αποτελέσματος για περαιτέρω χρήση.

```
def calculate_vat(price, rate=0.24):
    """
    Υπολογίζει και επιστρέφει τον ΦΠΑ.
    Η rate έχει προκαθορισμένη τιμή 0.24 (default argument).
    """
    vat_amount = price * rate
    return vat_amount

# Κλήση 1: Χρήση της προκαθορισμένης τιμής (rate=0.24)
product_price = 100
vat = calculate_vat(product_price)
total_price = product_price + vat
print(f"Τιμή χωρίς ΦΠΑ: {product_price}€, ΦΠΑ: {vat}€, Συνολική Τιμή: {total_price}
€")
# Εκτύπωση: Τιμή χωρίς ΦΠΑ: 100€, ΦΠΑ: 24.0€, Συνολική Τιμή: 124.0€

# Κλήση 2: Αλλαγή της προκαθορισμένης τιμής
vat_lower = calculate_vat(50, 0.13)
print(f"ΦΠΑ με μειωμένο συντελεστή (13%): {vat_lower}€")
# Εκτύπωση: ΦΠΑ με μειωμένο συντελεστή (13%): 6.5€
```

4. Συνάρτηση που Επιστρέφει Πολλαπλές Τιμές

Χρήση: Όταν μια συνάρτηση πρέπει να υπολογίσει περισσότερα από ένα αποτελέσματα. Η Python το κάνει επιστρέφοντας μια **πλειάδα (`tuple`)**.

```
def analyze_list(data_list):
    """Υπολογίζει το ελάχιστο, μέγιστο και το άθροισμα μιας λίστας."""
    min_val = min(data_list)
    max_val = max(data_list)
    sum_val = sum(data_list)
    # Επιστρέφουμε τις τρεις τιμές ως tuple
    return min_val, max_val, sum_val

numbers = [15, 8, 22, 11]

# Δέσμευση (unpacking) των τιμών της πλειάδας σε τρεις μεταβλητές
min_val, max_val, total_sum = analyze_list(numbers)

print(f"Λίστα: {numbers}")
print(f"Ελάχιστο: {min_val}")
print(f"Μέγιστο: {max_val}")
print(f"Άθροισμα: {total_sum}")
```

Δημιουργία Συναρτήσεων - Είδη Ορισμάτων (Προσθήκη)

Σε ακαδημαϊκό επίπεδο, είναι απαραίτητο να γίνει αναφορά στα τέσσερα είδη ορισμάτων και τη χρήση του *args και **kwargs.

Είδη Ορισμάτων στις Συναρτήσεις

Η Python υποστηρίζει τέσσερα (4) είδη τυπικών ορισμάτων16:

1. **Θέσης (Positional Arguments):** Μεταβιβάζονται με βάση τη σειρά τους.
2. **Λέξης-Κλειδιού (Keyword Arguments):** Μεταβιβάζονται με βάση το όνομά τους (όνομα=τιμή).
3. **Προεπιλογής (Default Arguments):** Παράμετροι με προκαθορισμένη τιμή (rate=0.24)17171717.
4. **Μεταβλητού Αριθμού Ορισμάτων:**

Όρισμα	**Σύνταξη**	**Περιγραφή**
Πλειάδα Ορισμάτων	*args	**Συλλέγει άγνωστο αριθμό ορισμάτων θέσης σε μια πλειάδα (**tuple**).**
Λεξικό Ορισμάτων	**kwargs	**Συλλέγει άγνωστο αριθμό ορισμάτων λέξης-κλειδιού σε ένα λεξικό (**dict**).**

Παράδειγμα Χρήσης & Κώδικας

```
def configure_settings(user_id, *args, **kwargs):
    """Δέχεται έναν user_id, μεταβλητό αριθμό ορισμάτων θέσης (*args)
    και λέξης-κλειδιού (**kwargs)."""
    print(f"User ID: {user_id}")
    print(f"Άγνωστα Ορίσματα Θέσης (*args): {args}")
    print(f"Άγνωστα Ορίσματα Λέξης-Κλειδιού (**kwargs): {kwargs}")

# Κλήση
configure_settings(
    101,                    # Positional
    "debug", "verbose",          # Συλλέγονται στο args (tuple)
    theme="dark", log_level=5     # Συλλέγονται στο kwargs (dict)
)
```

1. Modules (Ενότητες)

Ένα **Module** (Ενότητα) στην Python είναι απλώς ένα **αρχείο κώδικα Python** με επέκταση `.py`. Ένα module περιέχει συναρτήσεις, κλάσεις, μεταβλητές, ή ακόμα και εκτελέσιμες εντολές.

Σκοπός και Χρήση

- **Οργάνωση Κώδικα:** Επιτρέπει τη διαίρεση ενός μεγάλου προγράμματος σε μικρότερα, διαχειρίσιμα και λογικά τμήματα (βασική αρχή της **δομοποίησης**).
- **Επαναχρησιμοποίηση:** Μόλις οριστεί ένα module, οι ορισμοί του μπορούν να εισαχθούν και να χρησιμοποιηθούν σε πολλά άλλα αρχεία κώδικα.

Τεκμηρίωση & Σχόλια

Η τεκμηρίωση ενός module γίνεται συνήθως με ένα **docstring** (συμβολοσειρά πολλαπλών γραμμών) στην αρχή του αρχείου.

Παράδειγμα Module

Έστω ότι έχουμε ένα αρχείο με όνομα `calculations.py`.

```
# calculations.py

"""
Αυτό το module περιέχει βασικές μαθηματικές συναρτήσεις.
Χρησιμοποιείται για την επίδειξη της έννοιας του Module στην Python.
"""

PI = 3.14159  # Μεταβλητή επιπέδου module

def add(a, b):
    # Σχόλιο: Η συνάρτηση επιστρέφει το άθροισμα δύο αριθμών.
    return a + b

def circle_area(radius):
    """
    Υπολογίζει το εμβαδόν ενός κύκλου.
    Παράμετροι:
        radius (float/int): Η ακτίνα του κύκλου.
    Επιστρέφει:
        float: Το εμβαδόν.
    """
    return PI * (radius ** 2)
```

Τρόπος Χρήσης (Εισαγωγή Module)

Για να χρησιμοποιήσουμε τις συναρτήσεις και τις μεταβλητές του `calculations.py` σε ένα άλλο αρχείο (π.χ., `main_app.py`), χρησιμοποιούμε την εντολή **`import`**.

```
# main_app.py

# Τρόπος 1: Πλήρης εισαγωγή
import calculations

# Πρόσβαση με τη σύνταξη: module_name.function_name
result_add = calculations.add(10, 5)
area = calculations.circle_area(2)

print(f"Άθροισμα (μέσω module): {result_add}")
print(f"Εμβαδόν κύκλου: {area}")
print(f"Η σταθερά PI είναι: {calculations.PI}")


# Τρόπος 2: Εισαγωγή συγκεκριμένων στοιχείων (αποφεύγει το πρόθεμα module_name.)
from calculations import add, PI

sum_val = add(20, 3)
print(f"Νέο άθροισμα (χωρίς πρόθεμα): {sum_val}")
```

2. Packages (Πακέτα)

Ένα **Package** (Πακέτο) στην Python είναι ένας τρόπος για να **ομαδοποιήσουμε σχετικά modules** μαζί σε έναν ιεραρχικό κατάλογο (φάκελο), δημιουργώντας ένα οργανωμένο "χώρο ονομάτων" (**namespace**).

Δομή Package

Ένας φάκελος θεωρείται Python package αν περιέχει ένα ειδικό αρχείο: **`__init__.py`**.

```
my_project/
├── main_script.py
└── data_processing/   # <--- Αυτός είναι το Package
    ├── __init__.py    # <--- Κάνει το φάκελο package
    ├── file_ops.py    # <--- Module 1
    └── db_ops.py      # <--- Module 2
```

Το αρχείο `__init__.py` μπορεί να είναι κενό, αλλά η ύπαρξή του είναι απαραίτητη (σε παλαιότερες εκδόσεις της Python, πλέον προαιρετική από την 3.3+, αλλά καλή πρακτική). Μπορεί επίσης να περιέχει κώδικα αρχικοποίησης για το package.

Σκοπός και Χρήση

- **Ιεραρχική Οργάνωση:** Χρησιμοποιείται για τη δομή μεγάλων εφαρμογών, ομαδοποιώντας modules με βάση τη λειτουργικότητά τους (π.χ., όλα τα modules βάσης δεδομένων σε ένα package, όλα τα modules γραφικής διεπαφής σε άλλο).
- **Πρόληψη Συγκρούσεων Ονομάτων:** Επειδή κάθε module είναι εντός του namespace του package, αποφεύγονται οι συγκρούσεις ονομάτων.

Παράδειγμα Package

1. Δημιουργία Package (Φάκελοι και Αρχεία):

Έστω ότι έχουμε το package `data_processing`.

```
# data_processing/file_ops.py
def read_data(filename):
    """Διαβάζει δεδομένα από αρχείο (π.χ. CSV)."""
    return f"Διαβάστηκαν δεδομένα από: {filename}"
```

```
# data_processing/db_ops.py
def connect_db(db_name):
    """Συνδέεται σε μια βάση δεδομένων."""
    return f"Συνδέθηκε επιτυχώς στη βάση: {db_name}"
```

2. Τρόπος Χρήσης (Εισαγωγή Package Modules)

Για να χρησιμοποιήσουμε αυτά τα modules σε ένα αρχείο έξω από τον φάκελο (`main_script.py`), χρησιμοποιούμε τη σύνταξη με **τελείες** (`.`) για να δηλώσουμε την ιεραρχία.

```
# main_script.py

# Τρόπος 1: Εισαγωγή ενός ολόκληρου module από το package
import data_processing.file_ops

file_result = data_processing.file_ops.read_data("users.csv")
print(file_result)


# Τρόπος 2: Εισαγωγή συγκεκριμένης συνάρτησης από module του package
from data_processing.db_ops import connect_db

db_result = connect_db("production_db")
print(db_result)


# Τρόπος 3: Συντομογραφία (alias)
import data_processing.file_ops as fo

result = fo.read_data("logs.txt")
print(result)
```

7.Διαχείριση Αρχείων στην Python

Η Python χειρίζεται τη διαχείριση αρχείων (ανάγνωση, γραφή, προσθήκη) χρησιμοποιώντας την ενσωματωμένη συνάρτηση **open()**.

Είδη Αρχείων που Υποστηρίζονται

Η Python μπορεί να διαχειριστεί οποιονδήποτε τύπο αρχείου, καθώς αντιμετωπίζει τα δεδομένα ως **ακολουθίες από bytes**. Οι δύο κύριες κατηγορίες είναι:

1. **Αρχεία Κειμένου (Text Files):** Αρχεία που περιέχουν δεδομένα σε αναγνώσιμη μορφή (π.χ., txt, csv, html, json). Αυτά ανοίγουν σε **λειτουργία κειμένου (t)**.
2. **Δυαδικά Αρχεία (Binary Files):** Αρχεία που περιέχουν δεδομένα σε μορφή μη αναγνώσιμη από τον άνθρωπο (π.χ., εικόνες jpg, εκτελέσιμα προγράμματα exe, συμπιεσμένα αρχεία zip). Αυτά ανοίγουν σε **δυαδική λειτουργία (b)**.

1. Βασικές Μέθοδοι Διαχείρισης Αρχείων

Η συνάρτηση **open(filename, mode)** επιστρέφει ένα αντικείμενο αρχείου (file object), το οποίο έχει τις ακόλουθες βασικές μεθόδους:

Λειτουργίες Αρχείων (mode)

Το δεύτερο όρισμα της open() καθορίζει τον τρόπο με τον οποίο ανοίγουμε το αρχείο.

Λειτουργία	**Περιγραφή**
r (read)	**Άνοιγμα για ανάγνωση. (Προεπιλογή). Το αρχείο πρέπει να υπάρχει.**
w (write)	**Άνοιγμα για γραφή. Αν το αρχείο υπάρχει, το περιεχόμενό του διαγράφεται (overwrite). Αν δεν υπάρχει, δημιουργείται.**
a (append)	**Άνοιγμα για προσθήκη δεδομένων στο τέλος του αρχείου. Αν δεν υπάρχει, δημιουργείται.**
r+	**Άνοιγμα για ανάγνωση και γραφή.**
t (text)	**Λειτουργία κειμένου (Προεπιλογή).**
b (binary)	**Δυαδική λειτουργία (π.χ.,** rb**,** wb**).**

Μέθοδοι Αντικειμένου Αρχείου (File Object Methods)

Μέθοδος	Περιγραφή
read(size)	**Διαβάζει ολόκληρο το αρχείο ή** size **bytes/χαρακτήρες. Επιστρέφει συμβολοσειρά (text) ή bytes (binary).**
readline()	**Διαβάζει μία ολόκληρη γραμμή από το αρχείο.**
readlines()	**Διαβάζει όλες τις γραμμές του αρχείου και τις επιστρέφει ως λίστα από συμβολοσειρές.**
write(string)	**Γράφει τη δοθείσα συμβολοσειρά ή bytes στο αρχείο.**
writelines(list_of_strings)	**Γράφει μια λίστα από συμβολοσειρές στο αρχείο (δεν προσθέτει αυτόματα** \n**).**
close()	**Κλείνει το αντικείμενο αρχείου και απελευθερώνει τους πόρους. Πρέπει να καλείται πάντα.**

2. Χειρισμός Αρχείων Κειμένου (Text Files)

Κλασικός Τρόπος: open() και close()

Στον κλασικό τρόπο, είναι ευθύνη του προγραμματιστή να καλέσει τη μέθοδο close() για να εξασφαλίσει ότι οι αλλαγές αποθηκεύονται και οι πόροι απελευθερώνονται.

Παράδειγμα: Γραφή και Ανάγνωση

```
# 1. Γραφή (write - w): Δημιουργεί ή αντικαθιστά το περιεχόμενο
file_name = "example_classic.txt"
file_handler = open(file_name, 'w') # Άνοιγμα για γραφή
file_handler.write("Αυτή είναι η πρώτη γραμμή.\n")
file_handler.write("Αυτή είναι η δεύτερη γραμμή.")
file_handler.close() # Κλείσιμο αρχείου

# 2. Ανάγνωση (read - r)
file_handler = open(file_name, 'r')
content = file_handler.read() # Διαβάζει όλο το περιεχόμενο
print("--- Περιεχόμενο (Classic Read) ---")
print(content)
```

```
file_handler.close() # Κλείσιμο αρχείου

# 3. Προσθήκη (append - a)
file_handler = open(file_name, 'a')
file_handler.write("\nΤρίτη γραμμή (προσθήκη).")
file_handler.close()
```

Ο Προτιμώμενος Τρόπος: with open(...) as ...

Ο χειρισμός αρχείων με την εντολή **with** είναι ο **προτεινόμενος** τρόπος στην Python, καθώς εξασφαλίζει ότι η μέθοδος **close() καλείται αυτόματα**, ακόμα και αν προκύψει σφάλμα κατά τη διάρκεια της επεξεργασίας.

Παράδειγμα: Γραφή και Ανάγνωση με with

```
file_name_with = "example_with.txt"

# 1. Γραφή (write - w)
with open(file_name_with, 'w', encoding='utf-8') as f:
    # Μέσα στο 'with' μπλοκ, το αρχείο είναι ανοιχτό
    f.write("Γεια σου κόσμε!\n")
    f.write("Η χρήση του 'with' είναι η καλύτερη πρακτική.")
# Όταν βγαίνουμε από το μπλοκ 'with', το f.close() καλείται αυτόματα

# 2. Ανάγνωση γραμμή προς γραμμή (πιο αποδοτικός τρόπος)
print("\n--- Περιεχόμενο (Readline with 'with') ---")
with open(file_name_with, 'r', encoding='utf-8') as f:
    for line in f: # Επανάληψη (iteration) πάνω στο αντικείμενο του αρχείου
        print(line, end='') # Η line περιέχει ήδη τον χαρακτήρα \n
```

3. Διαχείριση Αρχείων CSV (Comma Separated Values)

Αν και τα αρχεία CSV είναι αρχεία κειμένου, η Python έχει ένα ειδικό ενσωματωμένο **csv module** για την εύκολη ανάγνωση και γραφή τους.

Παράδειγμα: Γραφή και Ανάγνωση Αρχείου CSV

Χρησιμοποιούμε το with και το module csv.

```
import csv

csv_file = "students.csv"
data_to_write = [
    ["Όνομα", "Ηλικία", "Βαθμός"],
    ["Αντώνης", 20, 8.5],
    ["Ελένη", 21, 9.2],
    ["Γιώργος", 20, 7.8]
]

# 1. Γραφή αρχείου CSV (using 'w' and csv.writer)
with open(csv_file, 'w', newline='', encoding='utf-8') as f:
    # Σημαντικό: newline='' αφαιρεί κενές γραμμές που προσθέτει το CSV module
    writer = csv.writer(f)
    writer.writerows(data_to_write) # Γράφει όλες τις γραμμές ταυτόχρονα

print(f"\nΔημιουργήθηκε αρχείο: {csv_file}")

# 2. Ανάγνωση αρχείου CSV (using 'r' and csv.reader)
print("--- Ανάγνωση CSV ---")
with open(csv_file, 'r', encoding='utf-8') as f:
    reader = csv.reader(f)
    header = next(reader) # Διαβάζει την πρώτη γραμμή (επικεφαλίδα)
    print(f"Επικεφαλίδα: {header}")

    for row in reader:
        # Κάθε γραμμή είναι μια λίστα από συμβολοσειρές
        print(f"Φοιτητής: {row[0]}, Βαθμός: {row[2]}")
```

4. Χειρισμός Άλλων Δομημένων Αρχείων (Π.χ. JSON)

Για αρχεία που έχουν πιο σύνθετη δομή από το CSV (όπως **JSON**), χρησιμοποιούμε ειδικά modules (π.χ., **json**).

Παράδειγμα: Γραφή και Ανάγνωση Αρχείου JSON

```
import json

json_file = "settings.json"
settings_data = {
    "theme": "dark",
    "notifications_enabled": True,
    "max_connections": 5
}

# 1. Γραφή αρχείου JSON
with open(json_file, 'w', encoding='utf-8') as f:
    # json.dump(): Μετατρέπει το dict σε συμβολοσειρά JSON και γράφει στο αρχείο
    json.dump(settings_data, f, indent=4) # indent=4 για ευανάγνωστη μορφή

print(f"\nΔημιουργήθηκε αρχείο: {json_file}")

# 2. Ανάγνωση αρχείου JSON
with open(json_file, 'r', encoding='utf-8') as f:
    # json.load(): Διαβάζει τη συμβολοσειρά JSON και τη μετατρέπει σε dict
    loaded_settings = json.load(f)

print("--- Ανάγνωση JSON ---")
print(f"Theme: {loaded_settings['theme']}")
print(f"Max Connections: {loaded_settings['max_connections']}")
```

8.Λίστες

1. Τι Είναι οι Λίστες στην Python

Οι **Λίστες (list)** είναι μία από τις πιο ευέλικτες και συχνά χρησιμοποιούμενες ενσωματωμένες δομές δεδομένων της Python.

Χαρακτηριστικό	**Περιγραφή**
Ορισμός	**Μια διατεταγμένη συλλογή αντικειμένων.**
Σύνταξη	**Δηλώνονται με τετράγωνες αγκύλες ([ ]), με τα στοιχεία να διαχωρίζονται με κόμμα.**
Μεταβλητότητα	**Είναι Μεταβλητές (Mutable). Αυτό σημαίνει ότι μπορείς να προσθέτεις, να αφαιρείς, ή να αλλάζεις στοιχεία *μετά* τη δημιουργία τους.**
Ετερογένεια	**Μπορούν να περιέχουν στοιχεία διαφορετικών τύπων δεδομένων (π.χ., ακέραιους, συμβολοσειρές, άλλες λίστες).**
Διαδοχή	**Υποστηρίζουν δείκτες (indexing) και τεμαχισμό (slicing), καθώς είναι διατεταγμένη ακολουθία.**

Παράδειγμα Δημιουργίας:

```
# Μια ετερογενής λίστα
my_list = [10, "apple", 3.14, True, [1, 2]]
print(f"Η λίστα μου: {my_list}")
print(f"Τύπος: {type(my_list)}")
print(f"Στοιχείο στη θέση 1: {my_list[1]}") # Πρόσβαση με δείκτη
```

2. Μέθοδοι Λιστών

Οι λίστες διαθέτουν πολλές μεθόδους που επιτρέπουν την εύκολη διαχείριση των στοιχείων τους:

Μέθοδος	Περιγραφή	Παράδειγμα
append(item)	**Προσθέτει ένα μοναδικό στοιχείο στο τέλος της λίστας.**	l.append(5)
extend(iterable)	**Προσθέτει τα στοιχεία μιας επαναληπτικής δομής (π.χ., άλλη λίστα) στο τέλος της τρέχουσας λίστας.**	l.extend([4, 5])
insert(index, item)	**Εισάγει ένα στοιχείο σε μια συγκεκριμένη θέση (δείκτη).**	l.insert(1, 'new')
remove(item)	**Αφαιρεί την πρώτη εμφάνιση της δοθείσας τιμής. Προκαλεί σφάλμα αν το στοιχείο δεν υπάρχει.**	l.remove('apple')
pop(index=-1)	**Αφαιρεί και επιστρέφει το στοιχείο στον δοθέντα δείκτη (προεπιλογή: το τελευταίο στοιχείο).**	last = l.pop()
index(item, start, end)	**Επιστρέφει τον δείκτη της πρώτης εμφάνισης του στοιχείου.**	idx = l.index(10)
count(item)	**Επιστρέφει τον αριθμό των φορών που εμφανίζεται ένα στοιχείο.**	c = l.count(3)
sort(reverse=False)	**Ταξινομεί τη λίστα επί τόπου (in-place) σε αύξουσα σειρά.**	l.sort()
reverse()	**Αντιστρέφει τη σειρά των στοιχείων επί τόπου.**	l.reverse()
copy()	**Επιστρέφει ένα ρηχό αντίγραφο (shallow copy) της λίστας.**	l2 = l.copy()
clear()	**Αφαιρεί όλα τα στοιχεία από τη λίστα.**	l.clear()

Ολοκληρωμένο Πρόγραμμα Επίδειξης Μεθόδων

```
# 1. Δημιουργία και βασικές λειτουργίες
programming_languages = ["Python", "Java", "C++", "C#"]
print(f"Αρχική λίστα: {programming_languages}")

# 2. append() - Προσθήκη στο τέλος
programming_languages.append("JavaScript")
```

```
print(f"Μετά append(): {programming_languages}") # Προστέθηκε το JavaScript

# 3. insert() - Εισαγωγή σε συγκεκριμένη θέση (δείκτης 1)
programming_languages.insert(1, "Go")
print(f"Μετά insert(): {programming_languages}") # Go στη θέση 1

# 4. extend() - Προσθήκη πολλαπλών στοιχείων από άλλη λίστα
more_langs = ["Rust", "Swift"]
programming_languages.extend(more_langs)
print(f"Μετά extend(): {programming_languages}")

# 5. remove() - Αφαίρεση με βάση την τιμή
programming_languages.remove("C#")
print(f"Μετά remove('C#'): {programming_languages}")

# 6. pop() - Αφαίρεση και επιστροφή του τελευταίου στοιχείου
last_lang = programming_languages.pop()
print(f"Το pop() αφαίρεσε: {last_lang}. Τρέχουσα λίστα: {programming_languages}")

# 7. sort() - Ταξινόμηση
programming_languages.sort(reverse=True) # Ταξινόμηση σε φθίνουσα σειρά
print(f"Μετά sort(reverse=True): {programming_languages}")

# 8. index() - Εύρεση δείκτη
index_of_python = programming_languages.index("Python")
print(f"Ο δείκτης του 'Python' είναι: {index_of_python}")
```

3. Δομές Δεδομένων που Δημιουργούνται με Λίστες

Λόγω της ευελιξίας και της μεταβλητότητας των λιστών, χρησιμοποιούνται συχνά για την υλοποίηση άλλων αφηρημένων δομών δεδομένων. Οι δύο πιο βασικές είναι η **Στοίβα** και η **Ουρά**.

Α) Στοίβα (Stack)

Η Στοίβα είναι μια αφηρημένη δομή δεδομένων που λειτουργεί με την αρχή **LIFO (Last-In, First-Out)**: το τελευταίο στοιχείο που προστέθηκε είναι το πρώτο που αφαιρείται.

- **Λειτουργίες Λίστας:** Χρησιμοποιεί **append()** για την εισαγωγή (push) και **pop()** χωρίς όρισμα για την αφαίρεση (pop) από το τέλος της λίστας.

Κώδικας Δημιουργίας & Χρήσης Στοίβας:

```
# Στοίβα (Stack) υλοποιημένη με λίστα - LIFO

stack = [] # Η λίστα μας λειτουργεί ως η στοίβα

# Λειτουργία Push (Εισαγωγή στοιχείου)
# Χρησιμοποιούμε append()
stack.append('A')
stack.append('B')
stack.append('C')
print(f"Στοίβα μετά τα Push: {stack}") # ['A', 'B', 'C']

# Λειτουργία Pop (Αφαίρεση τελευταίου στοιχείου)
# Χρησιμοποιούμε pop()
item_c = stack.pop()
print(f"Έγινε Pop (C): {item_c}")

item_b = stack.pop()
print(f"Έγινε Pop (B): {item_b}")

print(f"Στοίβα μετά τα Pop: {stack}") # ['A']
```

Β) Ουρά (Queue)

Η Ουρά είναι μια αφηρημένη δομή δεδομένων που λειτουργεί με την αρχή **FIFO (First-In, First-Out)**: το πρώτο στοιχείο που προστέθηκε είναι το πρώτο που αφαιρείται.

- **Λειτουργίες Λίστας:** Χρησιμοποιεί **append()** για την εισαγωγή (enqueue) στο τέλος και **pop(0)** για την αφαίρεση (dequeue) από την αρχή της λίστας.

Προσοχή: Η χρήση του list.pop(0) είναι **αναποτελεσματική** για μεγάλες λίστες, καθώς απαιτεί να μετακινηθούν όλα τα υπόλοιπα στοιχεία. Για ρεαλιστικές εφαρμογές Ουρών, προτιμάται η χρήση της κλάσης **collections.deque**.

Κώδικας Δημιουργίας & Χρήσης Ουράς (Αναποτελεσματικός με List):

```
# Ουρά (Queue) υλοποιημένη με λίστα - FIFO

queue = [] # Η λίστα μας λειτουργεί ως η ουρά
```

```
# Λειτουργία Enqueue (Εισαγωγή στοιχείου)
# Χρησιμοποιούμε append() για εισαγωγή στο τέλος
queue.append(10)
queue.append(20)
queue.append(30)
print(f"Ουρά μετά τα Enqueue: {queue}") # [10, 20, 30]

# Λειτουργία Dequeue (Αφαίρεση πρώτου στοιχείου)
# Χρησιμοποιούμε pop(0) για αφαίρεση από την αρχή
item_10 = queue.pop(0)
print(f"Έγινε Dequeue (10): {item_10}")

item_20 = queue.pop(0)
print(f"Έγινε Dequeue (20): {item_20}")

print(f"Ουρά μετά τα Dequeue: {queue}") # [30]
```

Σημείωση: Παρόλο που η λίστα μπορεί να υλοποιήσει την Ουρά, για **αποδοτικότερη** χρήση, ειδικά σε μεγάλα δεδομένα, χρησιμοποιούμε την κλάση **deque** (double-ended queue) από το module collections.

```
from collections import deque

# Χρήση deque για αποδοτική υλοποίηση Ουράς
efficient_queue = deque(['a', 'b', 'c'])
print(f"Αρχική deque: {efficient_queue}")

# Enqueue (προσθήκη στο τέλος)
efficient_queue.append('d')

# Dequeue (αφαίρεση από την αρχή)
first_item = efficient_queue.popleft()

print(f"Αφαιρέθηκε: {first_item}") # 'a'
print(f"Τελική deque: {efficient_queue}") # deque(['b', 'c', 'd'])
```

Υλοποίηση Στοίβας και Ουράς με Συναρτήσεις Οριζόμενες από τον Χρήστη (UDC)

Για την υλοποίηση της Στοίβας (Stack) και της Ουράς (Queue) με απλές συναρτήσεις στην Python, χρησιμοποιούμε τη **λίστα (list)** ως τη δομή βάσης, καθώς υποστηρίζει αποτελεσματικά τις απαραίτητες πράξεις (εισαγωγή/αφαίρεση στα άκρα).

1. Στοίβα (Stack) - LIFO (Last-In, First-Out)

Περιγραφή Δομής

Η Στοίβα είναι μια αφηρημένη δομή δεδομένων που λειτουργεί με την αρχή **LIFO** (Last-In, First-Out). Το τελευταίο στοιχείο που προστέθηκε (στην κορυφή) είναι το πρώτο που αφαιρείται. Στην Python, υλοποιείται χρησιμοποιώντας το τέλος της λίστας ως την κορυφή της στοίβας.

- **Λειτουργίες:**
    - **Push:** Εισαγωγή στοιχείου στην κορυφή (τέλος της λίστας).
    - **Pop:** Αφαίρεση και επιστροφή του στοιχείου από την κορυφή (τέλος της λίστας).
    - **Peek/Top:** Επιστροφή του στοιχείου της κορυφής, χωρίς αφαίρεση.

Υλοποίηση Συναρτήσεων σε Python

```
# Αρχικοποίηση: Χρησιμοποιούμε μια απλή λίστα για να αναπαραστήσουμε τη στοίβα.
# Αυτή η λίστα θα μεταβιβάζεται σε όλες τις συναρτήσεις ως όρισμα.
my_stack = []

def is_empty_stack(stack: list) -> bool:
    """
    Ελέγχει αν η στοίβα είναι άδεια.

    :param stack: Η λίστα που αναπαριστά τη στοίβα.
    :return: True αν η στοίβα είναι άδεια, False διαφορετικά.
    """
    return len(stack) == 0
```

```
def push(stack: list, item):
    """
    Προσθέτει ένα στοιχείο στην κορυφή της στοίβας (τέλος της λίστας).
    Η μέθοδος list.append() είναι O(1) και ιδανική για Push[cite: 591, 1358].

    :param stack: Η λίστα/στοίβα.
    :param item: Το στοιχείο προς εισαγωγή.
    """
    stack.append(item)
    print(f"PUSH: Προστέθηκε '{item}' στην κορυφή.")

def pop(stack: list):
    """
    Αφαιρεί και επιστρέφει το στοιχείο από την κορυφή της στοίβας (τέλος της λίστας).
    Η μέθοδος list.pop() χωρίς όρισμα είναι O(1) και ιδανική για Pop[cite: 597, 1358].

    :param stack: Η λίστα/στοίβα.
    :return: Το στοιχείο που αφαιρέθηκε ή μήνυμα σφάλματος.
    """
    if is_empty_stack(stack):
        # Χειρισμός σφάλματος: Δεν μπορούμε να κάνουμε pop σε άδεια στοίβα.
        raise ValueError("Σφάλμα POP: Η Στοίβα είναι άδεια.")

    popped_item = stack.pop()
    print(f"POP: Αφαιρέθηκε '{popped_item}' από την κορυφή.")
    return popped_item

def peek(stack: list):
    """
    Επιστρέφει το στοιχείο στην κορυφή (τέλος) χωρίς να το αφαιρεί.

    :param stack: Η λίστα/στοίβα.
    :return: Το στοιχείο της κορυφής.
    """
    if is_empty_stack(stack):
        raise ValueError("Σφάλμα PEEK: Η Στοίβα είναι άδεια.")
```

```
    # Πρόσβαση στο τελευταίο στοιχείο της λίστας
    return stack[-1]

# ------------------------------------------------------------------------------

### Αναλυτική Τεκμηρίωση Χρήσης
print("--- Επίδειξη Στοίβας (Stack) ---")

# 1. Push
push(my_stack, "URL 1")
push(my_stack, "URL 2")
push(my_stack, "URL 3")
print(f"Τρέχουσα Στοίβα (Κάτω -> Κορυφή): {my_stack}")

# 2. Peek
print(f"PEEK: Κορυφή είναι: {peek(my_stack)}") # Επιστρέφει URL 3

# 3. Pop
first_out = pop(my_stack) # Αφαιρεί το URL 3
print(f"Αφαιρέθηκε: {first_out}")
print(f"Στοίβα μετά το pop: {my_stack}")

# 4. Έλεγχος
print(f"Είναι άδεια; {is_empty_stack(my_stack)}") # False
```

2. Ουρά (Queue) - FIFO (First-In, First-Out)

Περιγραφή Δομής

Η Ουρά είναι μια αφηρημένη δομή δεδομένων που λειτουργεί με την αρχή **FIFO** (First-In, First-Out). Το πρώτο στοιχείο που προστέθηκε (στην αρχή/μπροστά) είναι το πρώτο που αφαιρείται.

- **Λειτουργίες:**
    - **Enqueue:** Εισαγωγή στοιχείου στο τέλος (πίσω) της ουράς.
    - **Dequeue:** Αφαίρεση και επιστροφή του στοιχείου από την αρχή (μπροστά) της ουράς.

- **Front:** Επιστροφή του στοιχείου της αρχής, χωρίς αφαίρεση.

Σημαντική Σημείωση για την Python: Η χρήση της απλής λίστας για **Dequeue** (αφαίρεση από την αρχή με list.pop(0)) είναι **αναποτελεσματική** (O(n)) για μεγάλες λίστες, καθώς όλα τα υπόλοιπα στοιχεία πρέπει να μετακινηθούν. Για ρεαλιστικές εφαρμογές, προτιμάται η κλάση **collections.deque**. Ωστόσο, για ακαδημαϊκή επίδειξη με απλές συναρτήσεις, χρησιμοποιούμε τη μέθοδο list.pop(0).

Υλοποίηση Συναρτήσεων σε Python

```
# Αρχικοποίηση: Χρησιμοποιούμε μια απλή λίστα για να αναπαραστήσουμε την ουρά.
my_queue = []

def is_empty_queue(queue: list) -> bool:
    """
    Ελέγχει αν η ουρά είναι άδεια.
    """
    return len(queue) == 0

def enqueue(queue: list, item):
    """
    Προσθέτει ένα στοιχείο στο τέλος (πίσω) της ουράς.
    Χρησιμοποιούμε list.append().

    :param queue: Η λίστα/ουρά.
    :param item: Το στοιχείο προς εισαγωγή.
    """
    queue.append(item)
    print(f"ENQUEUE: Προστέθηκε '{item}' στο τέλος.")

def dequeue(queue: list):
    """
    Αφαιρεί και επιστρέφει το στοιχείο από την αρχή (μπροστά) της ουράς.
    Χρησιμοποιούμε list.pop(0).

    :param queue: Η λίστα/ουρά.
```

```
    :return: Το στοιχείο που εξυπηρετήθηκε ή μήνυμα σφάλματος.
    """
    if is_empty_queue(queue):
        # Χειρισμός σφάλματος: Δεν μπορούμε να κάνουμε dequeue σε άδεια ουρά.
        raise IndexError("Σφάλμα DEQUEUE: Η Ουρά είναι άδεια.")

    # Αφαίρεση από την αρχή (δείκτης 0)
    dequeued_item = queue.pop(0)
    print(f"DEQUEUE: Εξυπηρετήθηκε '{dequeued_item}' από την αρχή.")
    return dequeued_item

def front(queue: list):
    """
    Επιστρέφει το στοιχείο στην αρχή (μπροστά) χωρίς να το αφαιρεί[cite: 1456].

    :param queue: Η λίστα/ουρά.
    :return: Το στοιχείο της αρχής.
    """
    if is_empty_queue(queue):
        raise IndexError("Σφάλμα FRONT: Η Ουρά είναι άδεια.")

    # Πρόσβαση στο πρώτο στοιχείο της λίστας
    return queue[0]

# ------------------------------------------------------------------------

### Αναλυτική Τεκμηρίωση Χρήσης
print("\n--- Επίδειξη Ουράς (Queue) ---")

# 1. Enqueue
enqueue(my_queue, "Ticket 1 (First In)") # Πρώτο μέσα (FIFO)
enqueue(my_queue, "Ticket 2")
enqueue(my_queue, "Ticket 3 (Last In)")
print(f"Τρέχουσα Ουρά (Μπροστά -> Πίσω): {my_queue}")

# 2. Front
print(f"FRONT: Επόμενο είναι: {front(my_queue)}") # Επιστρέφει Ticket 1
```

```
# 3. Dequeue
served_1 = dequeue(my_queue) # Αφαιρεί το Ticket 1
served_2 = dequeue(my_queue) # Αφαιρεί το Ticket 2
print(f"Εξυπηρετήθηκαν: {served_1}, {served_2}")
print(f"Ουρά μετά το dequeue: {my_queue}")

# 4. Έλεγχος
print(f"Είναι άδεια; {is_empty_queue(my_queue)}") # False
```

9.Πλειάδες (Tuples)

1. Τι Είναι οι Πλειάδες (Tuples) στην Python

Οι **Πλειάδες (tuple)** είναι μια ενσωματωμένη δομή δεδομένων στην Python, παρόμοια με τις λίστες, αλλά με μια κρίσιμη διαφορά.

Χαρακτηριστικό	**Περιγραφή**
Ορισμός	**Μια διατεταγμένη συλλογή αντικειμένων.**
Σύνταξη	**Δηλώνονται με παρενθέσεις (( )), με τα στοιχεία να διαχωρίζονται με κόμμα. (Οι παρενθέσεις είναι προαιρετικές, το κόμμα είναι αυτό που ορίζει την πλειάδα).**
Μεταβλητότητα	**Είναι Αμετάβλητες (Immutable). Αυτό σημαίνει ότι δεν μπορείς να προσθέσεις, να αφαιρέσεις, ή να αλλάξεις στοιχεία *μετά* τη δημιουργία τους.**
Ετερογένεια	**Μπορούν να περιέχουν στοιχεία διαφορετικών τύπων δεδομένων.**
Διαδοχή	**Υποστηρίζουν δείκτες (indexing) και τεμαχισμό (slicing).**
Κύρια Χρήση	**Χρησιμοποιούνται για την αναπαράσταση σταθερών συνόλων δεδομένων (π.χ., συντεταγμένες, ημερομηνίες) ή για συναρτήσεις που επιστρέφουν πολλαπλές τιμές.**

Παράδειγμα Δημιουργίας:

```
# 1. Πλειάδα με παρενθέσεις (καλή πρακτική)
coordinates = (10.0, 20.0)
# 2. Πλειάδα χωρίς παρενθέσεις (το κόμμα είναι το κλειδί)
user_data = "Alice", 30, "admin"
# 3. Πλειάδα με ένα στοιχείο (χρειάζεται κόμμα)
single_tuple = (5,)

print(f"Συντεταγμένες: {coordinates}, Τύπος: {type(coordinates)}")
print(f"Δεδομένα χρήστη: {user_data}")
print(f"Πλειάδα με 1 στοιχείο: {single_tuple}, Τύπος: {type(single_tuple)}")
```

2. Μέθοδοι Πλειάδων

Λόγω της **αμεταβλητότητάς** τους, οι πλειάδες έχουν πολύ λιγότερες μεθόδους σε σχέση με τις λίστες, καθώς δεν υποστηρίζουν λειτουργίες αλλαγής (π.χ., append, insert, remove, sort).

Μέθοδος	**Περιγραφή**	**Παράδειγμα**
count(item)	**Επιστρέφει τον αριθμό των φορών που εμφανίζεται ένα στοιχείο στην πλειάδα.**	t.count('a')
index(item)	**Επιστρέφει τον δείκτη της πρώτης εμφάνισης του στοιχείου. Προκαλεί σφάλμα αν το στοιχείο δεν υπάρχει.**	idx = t.index(30)

Ολοκληρωμένο Πρόγραμμα Επίδειξης Μεθόδων

```
# Πλειάδα με επαναλαμβανόμενα στοιχεία για επίδειξη του count()
grades = (85, 90, 85, 95, 85, 70)

# 1. index() - Εύρεση του δείκτη
index_90 = grades.index(90)
print(f"Ο βαθμός 90 βρίσκεται στη θέση: {index_90}") # Θέση 1

# 2. count() - Καταμέτρηση εμφάνισης στοιχείου
count_85 = grades.count(85)
print(f"Ο βαθμός 85 εμφανίζεται {count_85} φορές.") # 3 φορές

# Σημαντικό: Προσπάθεια αλλαγής προκαλεί σφάλμα
try:
    grades[0] = 100
except TypeError as e:
    print(f"\nΣφάλμα: {e}")
    print("Οι πλειάδες είναι αμετάβλητες και δεν μπορείς να αλλάξεις στοιχεία.")
```

3. Δομές Δεδομένων που Δημιουργούνται με Πλειάδες

Οι πλειάδες, λόγω της αμεταβλητότητας και της ταχύτητάς τους, χρησιμοποιούνται ως **σταθερές** δομικές μονάδες για άλλες δομές δεδομένων.

Α) Κλειδιά Λεξικού (Dictionary Keys)

Τα κλειδιά ενός λεξικού στην Python πρέπει να είναι **αμετάβλητα (immutable)**. Επειδή οι λίστες είναι μεταβλητές, δεν μπορούν να χρησιμοποιηθούν ως κλειδιά. Οι πλειάδες, όντας αμετάβλητες, είναι ιδανικές για σύνθετα κλειδιά.

Κώδικας Δημιουργίας & Χρήσης:

```
# Λεξικό όπου το κλειδί είναι μια πλειάδα (π.χ., Συντεταγμένες)

# Το κλειδί είναι: (lat, lon)
location_traffic = {
    (38.001, 23.705): "Υψηλή κίνηση",  # Πλειάδα ως κλειδί
    (38.005, 23.710): "Κανονική κίνηση",
}

# Εύρεση τιμής χρησιμοποιώντας το σύνθετο κλειδί
traffic_at_point = location_traffic.get((38.001, 23.705))
print(f"Κίνηση στο (38.001, 23.705): {traffic_at_point}")

# Λόγω της αμεταβλητότητας της πλειάδας, το κλειδί είναι hashable.
# Αν δοκιμάζαμε να χρησιμοποιήσουμε λίστα [38.001, 23.705] ως κλειδί, θα είχαμε TypeError.
```

Β) Εγγραφές Βάσης Δεδομένων/Σταθερά Αρχεία (Fixed Records)

Οι πλειάδες είναι η ιδανική δομή για την αναπαράσταση **σταθερών εγγραφών (records)**, όπου κάθε θέση έχει ένα συγκεκριμένο νόημα και δεν αναμένεται να αλλάξει.

Κώδικας Δημιουργίας & Χρήσης:

```
# Εγγραφές με σταθερή δομή (Immutable Records)

def create_product(product_id, name, price):
    """
    Δημιουργεί μια νέα εγγραφή προϊόντος ως πλειάδα.
    Επιστρέφει: (int, str, float)
    """
    return (product_id, name, price) # Η σειρά των πεδίων είναι σταθερή

# Λίστα που περιέχει εγγραφές-πλειάδες
inventory = [
    create_product(101, "Laptop", 1200.50),
    create_product(102, "Mouse", 25.00),
    create_product(103, "Monitor", 300.99)
]

print("\nΛίστα Αποθέματος (Λίστα από Πλειάδες):")
for record in inventory:
    # Πρόσβαση με δείκτη (Αποσυμπίεση πλειάδας)
    product_id, name, price = record

    # Επεξεργασία
    vat = price * 0.24

    print(f"ID: {product_id}, Όνομα: {name}, Τιμή με ΦΠΑ: {price + vat:.2f}€")
```

10.Λεξικά.

1. Τι Είναι τα Λεξικά (Dictionaries) στην Python

Τα **Λεξικά (dict)** είναι μια ισχυρή και ευέλικτη δομή δεδομένων στην Python που αποθηκεύει δεδομένα με τη μορφή ζευγών **κλειδιού-τιμής (key-value pairs)**.

Χαρακτηριστικό	**Περιγραφή**
Ορισμός	**Μια συλλογή ζευγών κλειδιού-τιμής. Κάθε κλειδί είναι μοναδικό και αντιστοιχεί σε μια τιμή.**
Σύνταξη	**Δηλώνονται με άγκιστρα ({ }), με τα ζεύγη να διαχωρίζονται με κόμμα και το κλειδί από την τιμή με άνω και κάτω τελεία (:).**
Μεταβλητότητα	**Είναι Μεταβλητά (Mutable). Μπορείς να προσθέτεις, να αφαιρείς, ή να αλλάζεις ζεύγη κλειδιού-τιμής.**
Διατεταγμένα	**Από την Python 3.7 και μετά, τα λεξικά είναι διατεταγμένα, διατηρώντας τη σειρά εισαγωγής των κλειδιών.**
Κλειδιά	**Τα κλειδιά πρέπει να είναι αμετάβλητα (immutable) και hashable (π.χ.,** str**,** int**,** tuple**).**
Πρόσβαση	**Η πρόσβαση στις τιμές γίνεται μέσω των κλειδιών (όχι με δείκτη θέσης).**

Παράδειγμα Δημιουργίας:

```
# Δημιουργία λεξικού με πληροφορίες χρήστη
person = {
    "name": "Alex",
    "age": 42,
    "is_active": True,
    "hobbies": ["reading", "hiking"] # Η τιμή μπορεί να είναι λίστα
}

print(f"Το λεξικό: {person}")
print(f"Όνομα: {person['name']}") # Πρόσβαση στην τιμή μέσω του κλειδιού 'name'
```

```
# Αλλαγή τιμής
person['age'] = 43
print(f"Νέα ηλικία: {person['age']}")

# Προσθήκη νέου ζεύγους
person['city'] = "Athens"
print(f"Μετά την προσθήκη: {person}")
```

2. Μέθοδοι Λεξικών

Τα λεξικά διαθέτουν μεθόδους για την αποτελεσματική διαχείριση των ζευγών κλειδιού-τιμής:

Μέθοδος	**Περιγραφή**	**Παράδειγμα**
get(key, default)	**Επιστρέφει την τιμή για το δοθέν κλειδί. Αν το κλειδί δεν υπάρχει, επιστρέφει** None **ή την προεπιλεγμένη τιμή (**default**).**	v = d.get('key', 0)
keys()	**Επιστρέφει μια προβολή (view) όλων των κλειδιών του λεξικού.**	k = d.keys()
values()	**Επιστρέφει μια προβολή (view) όλων των τιμών του λεξικού.**	v = d.values()
items()	**Επιστρέφει μια προβολή (view) όλων των ζευγών ως πλειάδες** (κλειδί, τιμή)**.**	i = d.items()
pop(key, default)	**Αφαιρεί το κλειδί και επιστρέφει την αντίστοιχη τιμή. Προαιρετικά, επιστρέφει** default **αν το κλειδί δεν βρεθεί.**	p = d.pop('key')
popitem()	**Αφαιρεί και επιστρέφει το τελευταίο ζεύγος κλειδιού-τιμής ως πλειάδα.**	item = d.popitem()
update(iterable)	**Εισάγει ή ενημερώνει ζεύγη κλειδιού-τιμής από άλλο λεξικό ή επαναληπτικό αντικείμενο.**	d.update({'c': 3})
setdefault(key, default)	**Επιστρέφει την τιμή του κλειδιού. Αν το κλειδί δεν υπάρχει, το εισάγει με την τιμή** default **και επιστρέφει αυτήν την τιμή.**	d.setdefault('new', [])

Μέθοδος	Περιγραφή	Παράδειγμα
clear()	**Αφαιρεί όλα τα ζεύγη από το λεξικό.**	d.clear()

Ολοκληρωμένο Πρόγραμμα Επίδειξης Μεθόδων

```
# Λεξικό για την παρακολούθηση αποθέματος
inventory = {
    'laptop': 50,
    'monitor': 120,
    'keyboard': 300
}
print(f"Αρχικό Απόθεμα: {inventory}")

# 1. get() - Ασφαλής πρόσβαση
stock_mouse = inventory.get('mouse', 0) # Το 'mouse' δεν υπάρχει, επιστρέφει 0
print(f"Απόθεμα 'mouse': {stock_mouse}")

# 2. keys(), values(), items() - Προβολές
all_products = list(inventory.keys()) # Μετατροπή σε λίστα για εκτύπωση
all_stock = list(inventory.values())
print(f"Προϊόντα: {all_products}")
print(f"Ποσότητες: {all_stock}")

# 3. pop() - Αφαίρεση ενός στοιχείου
removed_stock = inventory.pop('monitor')
print(f"Αφαιρέθηκε το 'monitor' με ποσότητα: {removed_stock}")

# 4. update() - Ενημέρωση ή προσθήκη
new_items = {'mouse': 150, 'keyboard': 280} # Το keyboard ενημερώνεται, το mouse προστίθεται
inventory.update(new_items)
print(f"Μετά το update(): {inventory}")

# 5. setdefault() - Προσθέτει μόνο αν δεν υπάρχει (χρησιμοποιείται συχνά με λίστες)
inventory.setdefault('mouse', 500) # Το mouse υπάρχει, δεν αλλάζει.
inventory.setdefault('headphones', 100) # Το headphones προστίθεται.
print(f"Μετά setdefault(): {inventory}")
```

3. Δομές Δεδομένων που Δημιουργούνται με Λεξικά

Τα λεξικά είναι θεμελιώδη για τη δημιουργία πιο σύνθετων και αφηρημένων δομών δεδομένων, κυρίως λόγω της αποτελεσματικής τους αναζήτησης με κλειδί.

Α) Πίνακας Κατακερματισμού / Χάρτης (Hash Table / Map)

Στην Python, το ίδιο το λεξικό **είναι** η υλοποίηση του **Hash Table** (ή **Map**). Αυτό επιτρέπει την εισαγωγή, διαγραφή και αναζήτηση δεδομένων σε μέσο χρόνο O(1), ανεξάρτητα από το μέγεθος της δομής.

Χρήση: Για γρήγορη εύρεση δεδομένων με βάση ένα μοναδικό αναγνωριστικό.

```
# Hash Table (Λεξικό) για γρήγορη αναζήτηση
# Κλειδί: Αριθμός Μητρώου (int), Τιμή: Όνομα (str)
student_registry = {
    1204: "Μαρία Παππά",
    1205: "Νίκος Γεωργίου",
    1206: "Ελένη Βλάχου"
}

# Γρήγορη αναζήτηση με O(1)
student_id = 1205
if student_id in student_registry:
    name = student_registry[student_id]
    print(f"Βρέθηκε φοιτητής: {name}")

# Προσθήκη
student_registry[1207] = "Δημήτρης Κώστας"
print(f"Μέγεθος μητρώου: {len(student_registry)}")
```

Β) Λεξικό Λίστας (Dictionary of Lists - Grouping)

Είναι μια πολύ συνηθισμένη και ισχυρή δομή, όπου η **τιμή** κάθε κλειδιού είναι μια **Λίστα**. Χρησιμοποιείται για ομαδοποίηση δεδομένων με βάση ένα κοινό κριτήριο.

Χρήση: Ομαδοποίηση χρηστών ανά πόλη, ομαδοποίηση προϊόντων ανά κατηγορία.

Κώδικας Δημιουργίας & Χρήσης:

```
# Δομή: Λεξικό Λίστας (Grouping by Category)

# Λεξικό για ομαδοποίηση προϊόντων ανά κατηγορία
products_by_category = {}

def add_product(category, product_name):
    """
    Προσθέτει ένα προϊόν στη λίστα της αντίστοιχης κατηγορίας.
    Αν η κατηγορία δεν υπάρχει, τη δημιουργεί.
    """
    # Χρησιμοποιούμε setdefault για να διασφαλίσουμε ότι το κλειδί υπάρχει με κενή λίστα []
    products_by_category.setdefault(category, []).append(product_name)

# Προσθήκη προϊόντων
add_product("Electronics", "Smartphone")
add_product("Books", "The Odyssey")
add_product("Electronics", "Laptop") # Το 'Laptop' προστίθεται στην υπάρχουσα λίστα
add_product("Books", "1984")

print("\nΟμαδοποίηση Προϊόντων ανά Κατηγορία:")
for category, product_list in products_by_category.items():
    print(f"Κατηγορία '{category}': {product_list}")

# Έλεγχος:
# Κατηγορία 'Electronics': ['Smartphone', 'Laptop']
```

Γ) Δέντρο / Ενσωματωμένη Δομή (Tree / Nested Structure)

Τα λεξικά μπορούν να περιέχουν άλλα λεξικά ή λίστες ως τιμές, δημιουργώντας ιεραρχικές ή **ενσωματωμένες δομές (nested structures)** που προσομοιάζουν δέντρα ή δεδομένα τύπου JSON.

Χρήση: Αναπαράσταση πολύπλοκων δεδομένων, όπως ρυθμίσεις εφαρμογών ή αποτελέσματα API.

Κώδικας Δημιουργίας & Χρήσης:

```
# Ενσωματωμένο Λεξικό (Nested Dictionary) - Αναπαράσταση ιεραρχικών δεδομένων

system_config = {
    "server": {
        "host": "localhost",
        "port": 8080,
        "enabled": True
    },
    "users": [ # Λίστα από λεξικά
        {"id": 1, "username": "admin"},
        {"id": 2, "username": "guest"}
    ]
}

print("\nΑνάγνωση Ενσωματωμένων Δεδομένων:")

# Πρόσβαση στο port
server_port = system_config['server']['port']
print(f"Port του Server: {server_port}") # 8080

# Πρόσβαση στο username του πρώτου χρήστη
first_user_name = system_config['users'][0]['username']
print(f"Όνομα πρώτου χρήστη: {first_user_name}") # admin
```

11.Σύνολα.

1. Τι Είναι τα Σύνολα (Sets) στην Python

Το **Σύνολο (set)** είναι μια μη διατεταγμένη συλλογή μοναδικών στοιχείων. Είναι βασισμένο στην έννοια του συνόλου από τα Μαθηματικά.

Χαρακτηριστικό	Περιγραφή
Ορισμός	**Μια μη διατεταγμένη συλλογή μοναδικών (distinct) αντικειμένων.**
Σύνταξη	**Δηλώνονται με άγκιστρα ({ }) ή με τη συνάρτηση set().**
Μεταβλητότητα	**Είναι Μεταβλητά (Mutable). Μπορείς να προσθέτεις ή να αφαιρείς στοιχεία, αλλά τα ίδια τα στοιχεία πρέπει να είναι αμετάβλητα.**
Μοναδικότητα	**Κάθε στοιχείο μπορεί να εμφανιστεί μόνο μία φορά. Όλες οι διπλότυπες τιμές αγνοούνται κατά τη δημιουργία.**
Κύρια Χρήση	**Χρησιμοποιούνται για γρήγορους ελέγχους ύπαρξης (O(1) μέσος χρόνος), αφαίρεση διπλοτύπων, και μαθηματικές πράξεις συνόλων.**

Παράδειγμα Δημιουργίας:

```
# 1. Δημιουργία με άγκιστρα (το 3 εισάγεται μόνο μία φορά)
numbers = {1, 2, 3, 3, 4, 5}
print(f"Σύνολο: {numbers}")  # {1, 2, 3, 4, 5} - Η σειρά είναι απρόβλεπτη

# 2. Δημιουργία από Λίστα (για αφαίρεση διπλοτύπων)
duplicate_list = [5, 6, 7, 6, 8, 5]
unique_set = set(duplicate_list)
print(f"Μοναδικά στοιχεία: {unique_set}") # {5, 6, 7, 8}

# 3. Έλεγχος ύπαρξης (πολύ γρήγορος)
is_member = 4 in numbers
print(f"Το 4 είναι μέλος του συνόλου; {is_member}") # True

# Σημαντικό: Για να δημιουργήσουμε κενό σύνολο, χρησιμοποιούμε set()
```

```
empty_set = set()
# Το {} δημιουργεί κενό λεξικό, όχι κενό σύνολο.
```

2. Μέθοδοι Συνόλων

Οι μέθοδοι των συνόλων επικεντρώνονται στην προσθήκη/αφαίρεση και στις μαθηματικές πράξεις συνόλων.

Μέθοδοι Τροποποίησης (Mutable Operations)

Μέθοδος	Περιγραφή	Παράδειγμα
add(item)	**Προσθέτει ένα μοναδικό στοιχείο στο σύνολο.**	s.add(10)
remove(item)	**Αφαιρεί το στοιχείο. Προκαλεί σφάλμα αν το στοιχείο δεν υπάρχει.**	s.remove(5)
discard(item)	**Αφαιρεί το στοιχείο. Δεν προκαλεί σφάλμα αν το στοιχείο δεν υπάρχει.**	s.discard(5)
pop()	**Αφαιρεί και επιστρέφει ένα τυχαίο στοιχείο από το σύνολο.**	random_item = s.pop()
update(iterable)	**Προσθέτει όλα τα στοιχεία από μια επαναληπτική δομή (π.χ., λίστα, άλλο σύνολο) στο τρέχον σύνολο.**	s.update([6, 7])
clear()	**Αφαιρεί όλα τα στοιχεία από το σύνολο.**	s.clear()

Μέθοδοι Μαθηματικών Πράξεων (Set Operations)

Μέθοδος	Μαθηματική Πράξη	Περιγραφή
union(other)	**Ένωση**	**Επιστρέφει ένα νέο σύνολο με όλα τα στοιχεία και των δύο συνόλων.**
intersection(other)	**Τομή**	**Επιστρέφει ένα νέο σύνολο με τα κοινά στοιχεία (τομή).**
difference(other)	**Διαφορά**	**Επιστρέφει ένα νέο σύνολο με τα**

Μέθοδος	Μαθηματική Πράξη	Περιγραφή
		στοιχεία που υπάρχουν στο πρώτο σύνολο, αλλά όχι στο δεύτερο (διαφορά).
symmetric_difference(other)		**Επιστρέφει ένα νέο σύνολο με τα στοιχεία που ανήκουν ακριβώς σε ένα από τα δύο σύνολα.**
issubset(other)	**Γνήσιο υποσύνολο**	**Επιστρέφει** True **αν όλα τα στοιχεία του τρέχοντος συνόλου περιέχονται στο** other**.**
issuperset(other)	**Υπερσύνολο**	**Επιστρέφει** True **αν το** other **είναι υποσύνολο του τρέχοντος συνόλου.**

Ολοκληρωμένο Πρόγραμμα Επίδειξης Μεθόδων

```
# Δημιουργία αρχικών συνόλων
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}

# 1. Πράξη Ένωσης (Union)
union_set = set_a.union(set_b)
print(f"A ένωση B: {union_set}") # {1, 2, 3, 4, 5, 6, 7, 8}

# 2. Πράξη Τομής (Intersection)
intersection_set = set_a.intersection(set_b)
print(f"A τομή B: {intersection_set}") # {4, 5}

# 3. Πράξη Διαφοράς (Difference)
diff_a_b = set_a.difference(set_b)
print(f"A διαφορά B: {diff_a_b}") # {1, 2, 3}

# 4. Προσθήκη και Αφαίρεση
set_a.add(10) # Προσθήκη
set_a.discard(1) # Αφαίρεση χωρίς κίνδυνο σφάλματος
print(f"A μετά την τροποποίηση: {set_a}") # {2, 3, 4, 5, 10}

# 5. Υποσύνολο
```

```
set_c = {2, 3}
is_subset = set_c.issubset(set_a)
print(f"{set_c} είναι υποσύνολο του {set_a}; {is_subset}") # True
```

3. Δομές Δεδομένων που Δημιουργούνται με Σύνολα

Τα σύνολα είναι η βασική δομή για την υλοποίηση δύο εννοιών: το ίδιο το σύνολο και το αμετάβλητο σύνολο.

Α) Σύνολο (set) - Επαναλήπτης Διπλοτύπων και Φίλτρο

Όπως αναφέρθηκε, η κύρια χρήση του μεταβλητού συνόλου είναι η αφαίρεση διπλοτύπων και ο γρήγορος έλεγχος ύπαρξης.

Κώδικας Δημιουργίας & Χρήσης:

```
# Χρήση 1: Αφαίρεση Διπλοτύπων από μια λίστα
all_orders = ["Laptop", "Monitor", "Keyboard", "Laptop", "Mouse", "Monitor"]
unique_orders = list(set(all_orders)) # Μετατροπή σε σύνολο και πίσω σε λίστα
print(f"Μοναδικές παραγγελίες: {unique_orders}")

# Χρήση 2: Γρήγορη Αφαίρεση Στοιχείων (Filtering)
allowed_products = {"Laptop", "Monitor", "Webcam"}
requested_products = {"Mouse", "Monitor", "Chair"}

# Χρήση τομής για να βρούμε τα κοινά (επιτρεπτά) προϊόντα
valid_intersection = allowed_products.intersection(requested_products)
print(f"Επιτρεπτά κοινά προϊόντα: {valid_intersection}") # {'Monitor'}
```

Β) Αμετάβλητο Σύνολο (frozenset)

Το **frozenset** είναι η **αμετάβλητη** έκδοση του συνόλου. Επειδή είναι αμετάβλητο, μπορεί να χρησιμοποιηθεί ως στοιχείο μέσα σε ένα άλλο σύνολο ή ως κλειδί σε ένα λεξικό.

- **Λειτουργίες:** Υποστηρίζει όλες τις μεθόδους μαθηματικών πράξεων συνόλων, αλλά **όχι** τις μεθόδους τροποποίησης (add, remove, update, κλπ.).

Κώδικας Δημιουργίας & Χρήσης:

```
# Δημιουργία frozenset
immutable_set = frozenset([1, 2, 3])
print(f"Frozen Set: {immutable_set}")
# immutable_set.add(4) # Θα προκαλούσε σφάλμα (AttributeError)

# Χρήση ως στοιχείο σε ένα κανονικό (μεταβλητό) σύνολο
# Ένα σύνολο μπορεί να περιέχει μόνο αμετάβλητα (hashable) στοιχεία.
groups_of_numbers = {
    frozenset({10, 20}), # Αυτό είναι ένα στοιχείο (frozenset)
    frozenset({30, 40})
}

# Προσθήκη ενός νέου frozenset στο σύνολο
new_group = frozenset({50, 60})
groups_of_numbers.add(new_group)

print(f"\nΣύνολο από Frozensets (Grouping): {groups_of_numbers}")
# Αυτό είναι αδύνατο με κανονικά set, καθώς τα set δεν είναι hashable.
```

12. Διαχείριση Εξαιρέσεων.

Τι Είναι η Διαχείριση Εξαιρέσεων;

Η **Διαχείριση Εξαιρέσεων (Exception Handling)** είναι ο μηχανισμός που επιτρέπει στον κώδικα να διαχειρίζεται σφάλματα που προκύπτουν κατά τον χρόνο εκτέλεσης (**runtime errors**), χωρίς να τερματίζεται απροσδόκητα το πρόγραμμα.

- **Σφάλμα (Error):** Γενικός όρος για ένα πρόβλημα στον κώδικα (π.χ., σφάλμα σύνταξης).
- **Εξαίρεση (Exception):** Ένα συμβάν που διακόπτει την κανονική ροή εκτέλεσης του προγράμματος. Οι εξαιρέσεις **μπορούν να προβλεφθούν και να αντιμετωπιστούν**.

Στόχος της διαχείρισης εξαιρέσεων είναι η **ομαλή αντιμετώπιση** απρόβλεπτων καταστάσεων και η παροχή ενός **ευανάγνωστου** μηνύματος ή η εκτέλεση ενός **καθαρισμού** (cleanup) των πόρων.

Υλοποίηση της Διαχείρισης Εξαιρέσεων στην Python

Στην Python, η διαχείριση εξαιρέσεων υλοποιείται χρησιμοποιώντας τέσσερις βασικές λέξεις-κλειδιά: **try**, **except**, **else**, και **finally**.

Λέξη-Κλειδί	**Περιγραφή**
try	**Περιέχει το μπλοκ κώδικα που ενδέχεται να προκαλέσει εξαίρεση.**
except	**Περιέχει το μπλοκ κώδικα που εκτελείται μόνο αν συμβεί μια εξαίρεση μέσα στο μπλοκ** try**. Μπορεί να στοχεύσει συγκεκριμένους τύπους εξαιρέσεων.**
else	**Περιέχει το μπλοκ κώδικα που εκτελείται μόνο αν το μπλοκ** try **ολοκληρωθεί χωρίς καμία εξαίρεση.**
finally	**Περιέχει το μπλοκ κώδικα που εκτελείται ΠΑΝΤΑ, ανεξάρτητα από το αν συνέβη ή όχι εξαίρεση. Χρησιμοποιείται για καθαρισμό πόρων.**

Βασική Σύνταξη

```
try:
    # Κώδικας που μπορεί να προκαλέσει σφάλμα
    ...
except ΤύποςΕξαίρεσης as e:
    # Χειρισμός του σφάλματος
    print(f"Συνέβη σφάλμα: {e}")
except (ΆλληΕξαίρεση, ΚιΆλλη) as e:
    # Χειρισμός πολλαπλών εξαιρέσεων
    ...
else:
    # Κώδικας που τρέχει μόνο αν το try ΔΕΝ έβγαλε εξαίρεση
    ...
finally:
    # Κώδικας που τρέχει πάντα (π.χ., κλείσιμο αρχείου)
    ...
```

Ολοκληρωμένα Προγράμματα Επίδειξης

1. Αριθμητικά Δεδομένα: Διαίρεση με το Μηδέν (ZeroDivisionError)

Εκφώνηση: Γράψε ένα πρόγραμμα που ζητά δύο αριθμούς και εκτελεί τη διαίρεσή τους. Χειρίσου την περίπτωση όπου ο δεύτερος αριθμός είναι μηδέν.

```
def safe_division():
    """Διαιρεί δύο αριθμούς, χειριζόμενο σφάλματα τύπου και διαίρεσης με το μηδέν."""

    try:
        # 1. Προσπαθούμε να πάρουμε έγκυρη είσοδο και να κάνουμε διαίρεση
        numerator_str = input("Εισάγετε τον αριθμητή (πρώτος αριθμός): ")
        denominator_str = input("Εισάγετε τον παρονομαστή (δεύτερος αριθμός): ")

        # Μετατροπή σε float
        numerator = float(numerator_str)
        denominator = float(denominator_str)

        result = numerator / denominator
```

```
    except ValueError:
        # 2. Χειρισμός εξαίρεσης: Όταν η μετατροπή σε float αποτύχει
        print("ΑΠΑΝΤΗΣΗ: Λάθος! Πρέπει να εισάγετε αριθμητικά δεδομένα.")

    except ZeroDivisionError:
        # 3. Χειρισμός εξαίρεσης: Όταν ο παρονομαστής είναι 0
        print("ΑΠΑΝΤΗΣΗ: Λάθος! Η διαίρεση με το μηδέν δεν επιτρέπεται.")

    else:
        # 4. Εκτελείται μόνο αν δεν υπήρξε καμία εξαίρεση
        print(f"ΑΠΑΝΤΗΣΗ: Το αποτέλεσμα της διαίρεσης είναι: {result:.2f}")

# Κλήση της συνάρτησης για επίδειξη
# safe_division()
```

2. Διαχείριση Αρχείων: Ανάγνωση Μη Υπάρχοντος Αρχείου (FileNotFoundError)

Εκφώνηση: Γράψε ένα πρόγραμμα που προσπαθεί να ανοίξει και να διαβάσει ένα αρχείο. Χειρίσου την εξαίρεση που προκύπτει αν το αρχείο δεν υπάρχει.

```
def read_config_file(filename):
    """Διαβάζει το περιεχόμενο ενός αρχείου, χειριζόμενο την περίπτωση που δεν υπάρχει."""

    file_handler = None # Αρχικοποίηση εκτός try

    try:
        # 1. Προσπαθούμε να ανοίξουμε το αρχείο
        file_handler = open(filename, 'r', encoding='utf-8')
        content = file_handler.read()
        print("--- Περιεχόμενο Αρχείου ---")
        print(content)

    except FileNotFoundError:
        # 2. Χειρισμός εξαίρεσης: Όταν το αρχείο δεν βρεθεί
        print(f"ΑΠΑΝΤΗΣΗ: Σφάλμα! Το αρχείο '{filename}' δεν βρέθηκε στο σύστημα.")

    except Exception as e:
        # 3. Χειρισμός οποιασδήποτε άλλης απρόβλεπτης εξαίρεσης
        print(f"ΑΠΑΝΤΗΣΗ: Προέκυψε ένα απροσδόκητο σφάλμα: {e}")
```

```
    finally:
        # 4. Εκτελείται πάντα για να εξασφαλιστεί ο καθαρισμός των πόρων
        if file_handler: # Έλεγχος αν το αρχείο όντως άνοιξε πριν κλείσει
            file_handler.close()
            print("INFO: Ο πόρος (αρχείο) έκλεισε επιτυχώς.")

# Κλήση της συνάρτησης
# read_config_file("non_existent_file.txt")
```

5 Εκφωνήσεις & Απαντήσεις με Χειρισμό Εξαιρέσεων

3. Χειρισμός Δύο Αριθμητικών Εξαιρέσεων Ταυτόχρονα (ValueError και ZeroDivisionError)

Εκφώνηση: Υλοποίησε μια συνάρτηση που δέχεται δύο ορίσματα και εκτυπώνει το αποτέλεσμα της διαίρεσης. Χρησιμοποίησε ένα μόνο μπλοκ except για να χειριστείς ταυτόχρονα τα σφάλματα **μη αριθμητικής εισόδου** και **διαίρεσης με το μηδέν**.

```
def division_handler(a, b):
    """Διαιρεί a/b, χειριζόμενη ταυτόχρονα ValueError και ZeroDivisionError."""
    try:
        # Εδώ κάνουμε τη διαίρεση
        result = float(a) / float(b)

    except (ValueError, ZeroDivisionError) as e:
        # Χειρισμός και των δύο εξαιρέσεων μαζί
        print(f"ΑΠΑΝΤΗΣΗ: Σφάλμα Καταχώρησης/Αριθμητικής Πράξης: {e}")
        return None

    print(f"Αποτέλεσμα διαίρεσης: {result:.2f}")

# Παραδείγματα:
# division_handler(10, 2)     # Αποτέλεσμα διαίρεσης: 5.00
# division_handler(10, 'zero') # Σφάλμα Καταχώρησης/Αριθμητικής Πράξης: could not convert string to float: 'zero'
# division_handler(10, 0)      # Σφάλμα Καταχώρησης/Αριθμητικής Πράξης: float division by zero
```

4. Διαχείριση Αρχείων με else (Επιτυχής Εκτέλεση)

Εκφώνηση: Γράψε κώδικα που προσπαθεί να γράψει σε ένα αρχείο. Εάν η εγγραφή είναι επιτυχής, εκτύπωσε ένα μήνυμα επιτυχίας. Χρησιμοποίησε το μπλοκ else.

```
def write_and_confirm(data):
    """Γράφει δεδομένα σε αρχείο και επιβεβαιώνει την επιτυχία με 'else'."""
    filename = "log.txt"
    try:
        # Χρήση 'with' για αυτόματο κλείσιμο (καλύτερη πρακτική)
        with open(filename, 'w', encoding='utf-8') as f:
            f.write(data)

    except IOError as e:
        # Χειρισμός σφαλμάτων εισόδου/εξόδου
        print(f"ΑΠΑΝΤΗΣΗ: Σφάλμα κατά τη γραφή του αρχείου: {e}")

    else:
        # Εκτελείται μόνο αν η γραφή ήταν επιτυχής
        print(f"ΑΠΑΝΤΗΣΗ: Τα δεδομένα γράφτηκαν επιτυχώς στο αρχείο '{filename}'.")

# write_and_confirm("Η σημερινή καταγραφή.")
```

5. Διαχείριση Λιστών: Λάθος Δείκτης (IndexError)

Εκφώνηση: Δημιούργησε μια λίστα και προσπάθησε να αποκτήσεις πρόσβαση σε έναν δείκτη που είναι εκτός ορίων. Χειρίσου την εξαίρεση IndexError.

```
def access_list_element(data_list, index):
    """Προσπαθεί να προσπελάσει στοιχείο λίστας με δεδομένο δείκτη, χειριζόμενο IndexError."""
    try:
        # Προσπάθεια πρόσβασης στο στοιχείο
        value = data_list[index]
    except IndexError:
        # Χειρισμός: Ο δείκτης είναι εκτός των ορίων [0, len-1]
        print(f"ΑΠΑΝΤΗΣΗ: Σφάλμα! Ο δείκτης {index} είναι εκτός ορίων της λίστας (0 έως
{len(data_list)-1}).")
    except Exception as e:
        # Γενικός χειρισμός για άλλα σφάλματα
```

```
            print(f"Απροσδόκητο σφάλμα: {e}")
        else:
            print(f"ΑΠΑΝΤΗΣΗ: Η τιμή στη θέση {index} είναι: {value}")

# my_data = [10, 20, 30]
# access_list_element(my_data, 1) # Επιτυχής: 20
# access_list_element(my_data, 5) # Αποτυχία: IndexError
```

13.Αντικειμενοστραφής προγραμματισμός.

Βασικές Αρχές του Αντικειμενοστραφούς Προγραμματισμού (OOP)

Οι τέσσερις (4) βασικές αρχές του OOP είναι:

1. **Εγκλεισμός (Encapsulation):** Η ομαδοποίηση δεδομένων (attributes) και μεθόδων (methods) που λειτουργούν πάνω σε αυτά, σε μία ενιαία μονάδα, την **Κλάση (Class)**. Κρύβει την εσωτερική κατάσταση του αντικειμένου από τον εξωτερικό κόσμο, περιορίζοντας την άμεση πρόσβαση.
2. **Αφαίρεση (Abstraction):** Η διαδικασία εμφάνισης μόνο των απαραίτητων πληροφοριών στον χρήστη και απόκρυψης των λεπτομερειών υλοποίησης. Ο χρήστης αλληλεπιδρά με το αντικείμενο μέσω μιας απλής διεπαφής (Interface).
3. **Κληρονομικότητα (Inheritance):** Η δυνατότητα δημιουργίας μιας νέας κλάσης (υποκλάση/παιδί) από μια υπάρχουσα (υπερκλάση/γονέας), κληρονομώντας τις ιδιότητες και τη συμπεριφορά της. Προωθεί την επαναχρησιμοποίηση κώδικα.
4. **Πολυμορφισμός (Polymorphism):** Η ικανότητα ενός αντικειμένου να παίρνει πολλές μορφές ή η δυνατότητα μιας μεθόδου να λειτουργεί διαφορετικά ανάλογα με τον τύπο του αντικειμένου που την καλεί.

Υλοποίηση Αρχών και Εννοιών στην Python

Έννοια/Αρχή	**Τρόπος Υλοποίησης στην Python**	**Παρατηρήσεις**
Κλάσεις	**Ορίζονται με τη λέξη κλειδί** class**.**	**Το σχέδιο για τη δημιουργία αντικειμένων.**
Αντικείμενα	**Δημιουργούνται με την κλήση της κλάσης ως συνάρτησης, π.χ.** αντικείμενο = Κλάση()**.**	**Στιγμιότυπα της κλάσης.**
Κατασκευαστής (Constructor)	**Η ειδική μέθοδος** __init__(self, ...)**. Καλείται αυτόματα κατά τη δημιουργία του**	**Χρησιμοποιείται για την αρχικοποίηση των ιδιοτήτων του αντικειμένου.**

Έννοια/Αρχή	Τρόπος Υλοποίησης στην Python	Παρατηρήσεις
	αντικειμένου.	
Αποδομητής (Destructor)	**Η ειδική μέθοδος** __del__(self)**. Καλείται όταν ένα αντικείμενο πρόκειται να καταστραφεί.**	**Σπάνια χρησιμοποιείται άμεσα, καθώς η Python έχει αυτόματο Garbage Collector (συλλέκτη απορριμμάτων).**
Public Μέλη	**Όλες οι ιδιότητες και μέθοδοι είναι Public από προεπιλογή.**	**Προσπελάσιμα απευθείας από οπουδήποτε.**
Protected Μέλη	**Μεμονωμένο underscore ως πρόθεμα:** _όνομα**.**	**Σύμβαση (convention), όχι αυστηρός περιορισμός. Υποδεικνύει ότι προορίζονται για εσωτερική χρήση ή χρήση από υποκλάσεις.**
Private Μέλη	**Διπλό underscore ως πρόθεμα:** __όνομα**.**	**Η Python υλοποιεί "name mangling" (π.χ.** __name **γίνεται** _ClassName__name**) για να δυσκολέψει την τυχαία πρόσβαση, αλλά δεν είναι αυστηρά Private.**
Overriding (Υπερκάλυψη Μεθόδου)	**Ορισμός μιας μεθόδου στην υποκλάση με το ίδιο όνομα και ίδια υπογραφή (αριθμός παραμέτρων) όπως η μέθοδος στον γονέα.**	**Επιτρέπει στην υποκλάση να παρέχει τη δική της υλοποίηση μιας κληρονομούμενης μεθόδου.**
Overloading (Υπερφόρτωση Μεθόδου)	**Η Python δεν υποστηρίζει την παραδοσιακή υπερφόρτωση μεθόδου (δηλαδή πολλαπλές μεθόδους με το ίδιο όνομα αλλά διαφορετικές υπογραφές παραμέτρων)**	**Μπορεί να προσομοιωθεί με προαιρετικά ορίσματα (***args**,** **kwargs**) ή ορίσματα με προεπιλεγμένες τιμές.**

Έννοια/Αρχή	Τρόπος Υλοποίησης στην Python	Παρατηρήσεις
	όπως άλλες γλώσσες.	
Κληρονομικότητα	**Η κλάση-παιδί δηλώνεται ως** class Παιδί(Γονέας):**.**	**Το παιδί κληρονομεί τις ιδιότητες και τις μεθόδους του γονέα.**
Interfaces (Διεπαφές)	**Η Python δεν έχει την έννοια της διεπαφής όπως η Java/C#.**	**Υλοποιείται συνήθως μέσω Αφηρημένων Βασικών Κλάσεων (Abstract Base Classes - ABC) χρησιμοποιώντας το module** abc **και τον decorator** @abstractmethod**.**

10 Ολοκληρωμένα Παραδείγματα OOP στην Python

Παράδειγμα 1: Κλάση και Αντικείμενο (Βασική Δομή)

Εκφώνηση: Δημιουργήστε μια κλάση Person με ιδιότητες name και age. Δημιουργήστε ένα αντικείμενο της κλάσης και εμφανίστε τα στοιχεία του.

Απάντηση & Λύση:

```
# Ορισμός της κλάσης Person
class Person:
    # Κατασκευαστής (Constructor) - Αρχικοποιεί τις ιδιότητες
    def __init__(self, name, age):
        self.name = name  # Public ιδιότητα
        self.age = age    # Public ιδιότητα

    # Μέθοδος για την εμφάνιση των στοιχείων
    def display_info(self):
        print(f"Όνομα: {self.name}, Ηλικία: {self.age}")

# Δημιουργία αντικειμένου (στιγμιότυπο της κλάσης)
person1 = Person("Γιώργος", 30)
```

```
# Κλήση της μεθόδου του αντικειμένου
print("Παράδειγμα 1: Κλάση και Αντικείμενο")
person1.display_info()
print("-" * 30)
```

Παράδειγμα 2: Εγκλεισμός & Protected Μέλος (Convention)

Εκφώνηση: Δημιουργήστε μια κλάση Car με μια protected ιδιότητα _max_speed. Δημιουργήστε μια public μέθοδο για την εμφάνισή της.

Απάντηση & Λύση:

```
class Car:
    def __init__(self, model, max_speed):
        self.model = model
        # Protected ιδιότητα (σύμβαση, όχι αυστηρός περιορισμός)
        self._max_speed = max_speed

    # Public μέθοδος για πρόσβαση στην protected ιδιότητα
    def get_max_speed(self):
        print(f"Το μοντέλο {self.model} έχει μέγιστη ταχύτητα: {self._max_speed} km/h")

# Δημιουργία αντικειμένου
my_car = Car("Skylark", 220)

print("Παράδειγμα 2: Εγκλεισμός & Protected Μέλος")
my_car.get_max_speed()
# ΠΡΟΣΟΧΗ: Μπορείτε ακόμα να αποκτήσετε πρόσβαση απ' έξω (λόγω σύμβασης Python)
print(f"Άμεση πρόσβαση στο _max_speed (αποθαρρύνεται): {my_car._max_speed}")
print("-" * 30)
```

Παράδειγμα 3: Εγκλεισμός & Private Μέλος (Name Mangling)

Εκφώνηση: Δημιουργήστε μια κλάση Account με μια private ιδιότητα __balance. Δημιουργήστε public μεθόδους deposit και get_balance.

Απάντηση & Λύση:

```
class Account:
    def __init__(self, initial_balance):
```

```
        # Private ιδιότητα (διπλό underscore, ενεργοποιεί name mangling)
        self.__balance = initial_balance

    # Public μέθοδος για κατάθεση
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Κατατέθηκαν {amount}. Νέο υπόλοιπο: {self.__balance}")

    # Public μέθοδος για εμφάνιση υπολοίπου
    def get_balance(self):
        return self.__balance

# Δημιουργία αντικειμένου
account = Account(1000)

print("Παράδειγμα 3: Εγκλεισμός & Private Μέλος")
account.deposit(500)
print(f"Τρέχον υπόλοιπο: {account.get_balance()}")

# Προσπάθεια άμεσης πρόσβασης (θα αποτύχει - AttributeError)
# print(account.__balance)

# Πρόσβαση μέσω name mangling (αποθαρρύνεται, αλλά δυνατή)
print(f"Πρόσβαση μέσω name mangling: {account._Account__balance}")
print("-" * 30)
```

Παράδειγμα 4: Κληρονομικότητα (Μονοεπίπεδη)

Εκφώνηση: Δημιουργήστε μια κλάση Animal και μια υποκλάση Dog που κληρονομεί από την Animal.

Απάντηση & Λύση:

```
# Υπερκλάση (Γονέας)
class Animal:
    def __init__(self, species):
        self.species = species
```

```
    def make_sound(self):
        print("Γενικός ήχος ζώου")

# Υποκλάση (Παιδί) - Κληρονομεί από την Animal
class Dog(Animal):
    def __init__(self, name):
        # Καλεί τον κατασκευαστή του γονέα
        super().__init__("Σκύλος")
        self.name = name

    # Υπερκάλυψη (Overriding) της μεθόδου make_sound
    def make_sound(self):
        print(f"{self.name} (είδος: {self.species}) λέει: Woof!")

# Δημιουργία αντικειμένου Dog
dog1 = Dog("Μπρούνο")

print("Παράδειγμα 4: Κληρονομικότητα")
dog1.make_sound()
print("-" * 30)
```

Παράδειγμα 5: Υπερκάλυψη Μεθόδου (Method Overriding)

Εκφώνηση: Δημιουργήστε μια κλάση Shape με μια μέθοδο area(). Δημιουργήστε μια υποκλάση Rectangle που υπερκαλύπτει τη μέθοδο area() για τον υπολογισμό του εμβαδού.

Απάντηση & Λύση:

```
class Shape:
    def area(self):
        # Βασική υλοποίηση (καθόλου εμβαδό ή γενικό μήνυμα)
        raise NotImplementedError("Η μέθοδος area() πρέπει να υλοποιηθεί από την υποκλάση")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    # Υπερκάλυψη της μεθόδου area()
    def area(self):
        return self.width * self.height

# Δημιουργία αντικειμένου
rect = Rectangle(10, 5)

print("Παράδειγμα 5: Υπερκάλυψη Μεθόδου (Overriding)")
print(f"Εμβαδόν Ορθογωνίου: {rect.area()}")
print("-" * 30)
```

Παράδειγμα 6: Πολυμορφισμός (με υπερκάλυψη)

Εκφώνηση: Δημιουργήστε μια συνάρτηση που δέχεται οποιοδήποτε αντικείμενο τύπου Shape και καλεί τη μέθοδο area() του.

Απάντηση & Λύση:

```
# Χρησιμοποιούμε τις κλάσεις Shape και Rectangle από το Παράδειγμα 5
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    # Υπερκάλυψη της μεθόδου area()
    def area(self):
        return 3.14 * self.radius ** 2

# Συνάρτηση που εκμεταλλεύεται τον Πολυμορφισμό
def print_area(shape):
    # Η ίδια κλήση `shape.area()` λειτουργεί διαφορετικά ανάλογα με τον τύπο του αντικειμένου
    print(f"Το εμβαδόν είναι: {shape.area():.2f}")

rect = Rectangle(4, 6)
circle = Circle(5)

print("Παράδειγμα 6: Πολυμορφισμός")
print_area(rect)
```

```
print_area(circle)
print("-" * 30)
```

Παράδειγμα 7: Προσομοίωση Υπερφόρτωσης Μεθόδου (Overloading)

Εκφώνηση: Δημιουργήστε μια μέθοδο calculate στην κλάση Calculator που μπορεί να προσθέσει είτε δύο αριθμούς είτε τρεις.

Απάντηση & Λύση:

```
class Calculator:
    # Προσομοίωση Overloading με προεπιλεγμένες τιμές
    def calculate_sum(self, a, b, c=None):
        if c is None:
            return a + b
        else:
            return a + b + c

# Δημιουργία αντικειμένου
calc = Calculator()

print("Παράδειγμα 7: Προσομοίωση Υπερφόρτωσης Μεθόδου")
# Κλήση με 2 ορίσματα
result2 = calc.calculate_sum(5, 10)
print(f"Άθροισμα 2 αριθμών: {result2}")

# Κλήση με 3 ορίσματα
result3 = calc.calculate_sum(5, 10, 15)
print(f"Άθροισμα 3 αριθμών: {result3}")
print("-" * 30)
```

Παράδειγμα 8: Χρήση super() στον Κατασκευαστή

Εκφώνηση: Επεκτείνετε τον κατασκευαστή της κλάσης Dog (από το Π.4) ώστε να δέχεται και το color και να καλεί τον κατασκευαστή του γονέα.

Απάντηση & Λύση:

```
class Animal:
    def __init__(self, species):
        self.species = species
```

```
        print(f"Animal Constructor called for: {species}")

class Dog(Animal):
    def __init__(self, name, color):
        # Καλεί τον __init__ του Animal
        super().__init__("Σκύλος")
        self.name = name
        self.color = color

    def describe(self):
        print(f"Είμαι ο {self.name}, ένα {self.species}, χρώματος {self.color}.")

dog2 = Dog("Λάκι", "καφέ")

print("Παράδειγμα 8: Χρήση super()")
dog2.describe()
print("-" * 30)
```

Παράδειγμα 9: Αφαίρεση (Abstraction) με ABC

Εκφώνηση: Δημιουργήστε μια αφηρημένη κλάση Payment και βεβαιωθείτε ότι οποιαδήποτε υποκλάση υλοποιεί τη μέθοδο process_payment.

Απάντηση & Λύση:

```
from abc import ABC, abstractmethod

# Αφηρημένη Βασική Κλάση (ABC) που λειτουργεί ως Interface
class Payment(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass # Αυτή η μέθοδος πρέπει να υλοποιηθεί

    def get_details(self):
        print("Εμφάνιση γενικών στοιχείων πληρωμής.")

# Υποκλάση που υλοποιεί την αφηρημένη μέθοδο
class CreditCardPayment(Payment):
    def process_payment(self, amount):
```

```
        print(f"Επεξεργασία πληρωμής με Πιστωτική Κάρτα: {amount} €.")

    # Δεν είναι υποχρεωτικό να καλέσουμε το super().get_details()

print("Παράδειγμα 9: Αφαίρεση (Abstraction) και Interfaces (ABC)")
cc_payment = CreditCardPayment()
cc_payment.process_payment(99.99)
cc_payment.get_details()

# Αν αφαιρούσαμε την process_payment, θα είχαμε TypeError κατά τη δημιουργία αντικειμένου
print("-" * 30)
```

Παράδειγμα 10: Κατασκευαστής και Αποδομητής

Εκφώνηση: Δημιουργήστε μια κλάση Resource με __init__ και __del__ για να δείξετε πότε δημιουργείται και καταστρέφεται ένα αντικείμενο.

Απάντηση & Λύση:

```
class Resource:
    def __init__(self, name):
        self.name = name
        # Προσομοίωση δέσμευσης πόρου (π.χ. άνοιγμα αρχείου)
        print(f"Κατασκευαστής: Δημιουργία πόρου: {self.name}")

    def __del__(self):
        # Προσομοίωση απελευθέρωσης πόρου (π.χ. κλείσιμο αρχείου)
        print(f"Αποδομητής: Καταστροφή και απελευθέρωση πόρου: {self.name}")

def create_and_destroy():
    # Δημιουργία αντικειμένου
    res1 = Resource("Αρχείο_1.txt")
    # Το αντικείμενο res1 καταστρέφεται (καλείται η __del__)
    # όταν η συνάρτηση τελειώσει και η αναφορά χαθεί.

print("Παράδειγμα 10: Κατασκευαστής και Αποδομητής")
# Όταν η συνάρτηση κληθεί, το αντικείμενο res1 θα δημιουργηθεί και μετά θα καταστραφεί.
```

```
create_and_destroy()
print("Η συνάρτηση create_and_destroy ολοκληρώθηκε.")
# Σημείωση: Η ακριβής στιγμή κλήσης του __del__ εξαρτάται από τον Garbage Collector της Python.
print("-" * 30)
```

14.Υπερφόρτωση τελεστών

Στην **Python**, η υπερφόρτωση τελεστών (**Operator Overloading**) γίνεται μέσω της υλοποίησης ειδικών μεθόδων που ονομάζονται **"Magic Methods"** ή **"Dunder Methods"** (από το "Double Underscore", π.χ. __add__). Αυτές οι μέθοδοι επιτρέπουν στις κλάσεις σας να ορίζουν τη δική τους συμπεριφορά για τους ενσωματωμένους τελεστές της Python.

Τρόπος Υπερφόρτωσης Τελεστών στην Python (Operator Overloading)

1. **Ορισμός Magic Method:** Για να υπερφορτώσετε έναν τελεστή, πρέπει να ορίσετε την αντίστοιχη Magic Method μέσα στην κλάση σας.
2. **Αυτόματη Κλήση:** Όταν ο τελεστής χρησιμοποιείται σε στιγμιότυπα της κλάσης σας, η Python καλεί αυτόματα την κατάλληλη Magic Method.
3. **Ορισμός Λογικής:** Μέσα στη Magic Method, ορίζετε τη λογική που θα εκτελεστεί. Αυτή η λογική συνήθως επιστρέφει ένα **νέο αντικείμενο** του ίδιου τύπου με το αποτέλεσμα της πράξης.

Για παράδειγμα, όταν χρησιμοποιείτε τον τελεστή πρόσθεσης (+) μεταξύ δύο αντικειμένων a και b (a + b), η Python καλεί τη μέθοδο a.__add__(b).

Υπερφόρτωση Αριθμητικών Τελεστών στην Python

Οι αριθμητικοί τελεστές χωρίζονται σε τρεις κατηγορίες με βάση τον τρόπο αλληλεπίδρασης:

1. Απλοί Αριθμητικοί Τελεστές (Binary Operators)

Αυτές οι μέθοδοι καθορίζουν τη συμπεριφορά για τις βασικές αριθμητικές πράξεις. Χρησιμοποιούνται κυρίως για πράξεις της μορφής a op b.

Τελεστής	**Μέθοδος (Προτιμώμενη)**	**Αντίστροφη Μέθοδος (Fallback)**	**Σκοπός**
+	__add__(self, other)	__radd__(self, other)	**Πρόσθεση**
-	__sub__(self, other)	__rsub__(self, other)	**Αφαίρεση**
*	__mul__(self, other)	__rmul__(self, other)	**Πολλαπλασιασμός**
/	__truediv__(self, other)	__rtruediv__(self, other)	**Διαίρεση (float)**

Τελεστής	Μέθοδος (Προτιμώμενη)	Αντίστροφη Μέθοδος (Fallback)	Σκοπός
//	__floordiv__(self, other)	__rfloordiv__(self, other)	**Διαίρεση (ακέραιος)**
%	__mod__(self, other)	__rmod__(self, other)	**Υπόλοιπο διαίρεσης (Modulo)**
**	__pow__(self, other)	__rpow__(self, other)	**Ύψωση σε δύναμη**

Λειτουργία Αντίστροφης Μεθόδου (__r...):

Εάν η έκφραση είναι b + a (όπου b είναι βασικός τύπος και a είναι αντικείμενο της κλάσης σας) ή αν η __add__ του b δεν υποστηρίζει την πράξη, η Python προσπαθεί να καλέσει την a.__radd__(b).

Παράδειγμα 1: Υπερφόρτωση Πρόσθεσης (+) και Πολλαπλασιασμού (*)

Εκφώνηση: Δημιουργήστε μια κλάση Vector για να αναπαραστήσετε 2D διανύσματα. Υπερφορτώστε τους τελεστές + (πρόσθεση διανυσμάτων) και * (πολλαπλασιασμός με βαθμωτό μέγεθος).

Λύση Κώδικα:

```
class Vector:
    """Κλάση για 2D διανύσματα."""
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Μέθοδος αναπαράστασης για ευανάγνωστη έξοδο
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    # 1. Υπερφόρτωση του + (τελεστής πρόσθεσης)
    def __add__(self, other):
        """Υπερφορτώνει τη συμπεριφορά του self + other."""
        # Ελέγχουμε αν το 'other' είναι επίσης Vector
        if isinstance(other, Vector):
```

```
            # Πρόσθεση συντεταγμένων
            new_x = self.x + other.x
            new_y = self.y + other.y
            # Επιστρέφουμε ένα ΝΕΟ αντικείμενο Vector
            return Vector(new_x, new_y)
        raise TypeError("Ο τελεστής '+' υποστηρίζει μόνο πρόσθεση με άλλο Vector")

    # 2. Υπερφόρτωση του * (τελεστής πολλαπλασιασμού με βαθμωτό μέγεθος)
    def __mul__(self, scalar):
        """Υπερφορτώνει τη συμπεριφορά του self * scalar (π.χ. Vector * 2)."""
        if isinstance(scalar, (int, float)):
            new_x = self.x * scalar
            new_y = self.y * scalar
            return Vector(new_x, new_y)
        raise TypeError("Ο τελεστής '*' υποστηρίζει μόνο πολλαπλασιασμό με αριθμό")

    # 3. Υπερφόρτωση του * (αντίστροφη μέθοδος, για το 2 * Vector)
    def __rmul__(self, scalar):
        """Υπερφορτώνει τη συμπεριφορά του scalar * self (π.χ. 2 * Vector)."""
        # Καλούμε απλά την __mul__ αφού η πράξη είναι αντιμεταθετική
        return self.__mul__(scalar)

# --- Τεκμηρίωση Κώδικα ---
v1 = Vector(2, 4)
v2 = Vector(3, 6)

print("Παράδειγμα 1: Υπερφόρτωση + και *")

# 1. Χρήση υπερφορτωμένου + (v1 + v2)
v3 = v1 + v2
print(f"Πρόσθεση: {v1} + {v2} = {v3}")  # Έξοδος: Vector(5, 10)

# 2. Χρήση υπερφορτωμένου * (v3 * 2)
v4 = v3 * 2
print(f"Πολλαπλασιασμός (δεξιά): {v3} * 2 = {v4}")  # Έξοδος: Vector(10, 20)

# 3. Χρήση υπερφορτωμένου * (Αντίστροφη μέθοδος) (3 * v1)
```

```
v5 = 3 * v1
print(f"Πολλαπλασιασμός (αριστερά): 3 * {v1} = {v5}")  # Έξοδος: Vector(6, 12)

# Έλεγχος τύπου (πρέπει να επιστρέψει σφάλμα)
try:
    _ = v1 + 5
except TypeError as e:
    print(f"Σφάλμα κατά την πρόσθεση με λάθος τύπο: {e}")
```

2. Τελεστές Σύνθετης Ανάθεσης (In-Place Operators)

Αυτές οι μέθοδοι καθορίζουν τη συμπεριφορά για πράξεις της μορφής a op= b (π.χ. a += b). Τροποποιούν το αντικείμενο **επί τόπου** και επιστρέφουν το ίδιο το αντικείμενο (self).

Τελεστής	**Μέθοδος**	**Σκοπός**
+=	__iadd__(self, other)	**Πρόσθεση και ανάθεση**
-=	__isub__(self, other)	**Αφαίρεση και ανάθεση**
*=	__imul__(self, other)	**Πολλαπλασιασμός και ανάθεση**
/=	__itruediv__(self, other)	**Διαίρεση και ανάθεση**
//=	__ifloordiv__(self, other)	**Διαίρεση (ακέραιος) και ανάθεση**
%=	__imod__(self, other)	**Υπόλοιπο και ανάθεση**
**=	__ipow__(self, other)	**Ύψωση σε δύναμη και ανάθεση**

Παράδειγμα 2: Υπερφόρτωση Πρόσθεσης Επί Τόπου (+=)

Εκφώνηση: Επεκτείνετε την κλάση Vector ώστε ο τελεστής += να τροποποιεί το διάνυσμα επί τόπου (in-place).

Λύση Κώδικα:

```
class MutableVector:
    """Κλάση για 2D διανύσματα με υποστήριξη in-place πράξεων."""
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"MutableVector({self.x}, {self.y})"

    # Υπερφόρτωση του += (in-place πρόσθεση)
    def __iadd__(self, other):
        """Υπερφορτώνει τη συμπεριφορά του self += other."""
        if isinstance(other, MutableVector):
            # Τροποποίηση του αντικειμένου self επί τόπου
            self.x += other.x
            self.y += other.y
            # Πρέπει να επιστρέψει το ίδιο το αντικείμενο self
            return self
        raise TypeError("Ο τελεστής '+=' υποστηρίζει μόνο πρόσθεση με άλλο
MutableVector")

# --- Τεκμηρίωση Κώδικα ---
vA = MutableVector(1, 2)
vB = MutableVector(10, 20)

print("Παράδειγμα 2: Υπερφόρτωση +=")

# Αρχικές τιμές και αναγνωριστικό μνήμης
print(f"Αρχικό vA: {vA}, ID: {id(vA)}")

# Χρήση υπερφορτωμένου += (in-place)
vA += vB
print(f"Μετά το vA += vB: {vA}, ID: {id(vA)}")

# Το ID πρέπει να παραμένει το ίδιο, αποδεικνύοντας την in-place τροποποίηση.

vC = MutableVector(5, 5)
```

```
vD = vC # vD είναι απλώς μια αναφορά στο vC

vC += MutableVector(1, 1)
print(f"Μετά το vC += (1, 1): vC={vC}, vD={vD}") # Και τα δύο αλλάζουν

print("-" * 30)
```

3. Μοναδιαίοι Τελεστές (Unary Operators)

Αυτές οι μέθοδοι καθορίζουν τη συμπεριφορά για τελεστές που δρουν σε ένα μόνο αντικείμενο (π.χ. +a, -a, abs(a)).

Τελεστής	**Μέθοδος**	**Σκοπός**
+	__pos__(self)	**Μοναδιαίο + (π.χ.** +a**)**
-	__neg__(self)	**Μοναδιαίο - (π.χ.** -a**)**
abs()	__abs__(self)	**Απόλυτη τιμή**
~	__invert__(self)	**Bitwise αναστροφή**

Παράδειγμα 3: Υπερφόρτωση Μοναδιαίων Τελεστών (- και abs())

Εκφώνηση: Στην κλάση Vector, υπερφορτώστε τον μοναδιαίο τελεστή - (αντιστροφή διανύσματος) και τη συνάρτηση abs() (υπολογισμός μέτρου διανύσματος).

Λύση Κώδικα:

```
import math

class Vector:
    # ... (ο κώδικας __init__ και __repr__ είναι ο ίδιος με πριν)
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"
```

```
    # 1. Υπερφόρτωση του μοναδιαίου - (αρνητικό πρόσημο)
    def __neg__(self):
        """Υπερφορτώνει τη συμπεριφορά του -self (αντιστροφή διανύσματος)."""
        # Επιστρέφει ένα ΝΕΟ αντικείμενο με αντίστροφες συντεταγμένες
        return Vector(-self.x, -self.y)

    # 2. Υπερφόρτωση της συνάρτησης abs() (μέτρο διανύσματος)
    def __abs__(self):
        """Υπερφορτώνει τη συμπεριφορά του abs(self) (υπολογίζει το μέτρο)."""
        # Μέτρο: sqrt(x^2 + y^2)
        return math.sqrt(self.x**2 + self.y**2)

# --- Τεκμηρίωση Κώδικα ---
vA = Vector(3, -4)

print("Παράδειγμα 3: Υπερφόρτωση Μοναδιαίων Τελεστών")

# 1. Χρήση υπερφορτωμένου - (-vA)
vNeg = -vA
print(f"Αντίστροφο: -{vA} = {vNeg}") # Έξοδος: Vector(-3, 4)

# 2. Χρήση υπερφορτωμένης abs()
vAbs = abs(vA)
print(f"Μέτρο: abs({vA}) = {vAbs:.2f}") # Έξοδος: 5.00 (sqrt(3^2 + (-4)^2) = 5)

print("-" * 30)
```

15.Υλοποίηση στοίβας και ουράς σαν αντικείμενα.

Η υλοποίηση της **Στοίβας (Stack)** και της **Ουράς (Queue)** ως αντικείμενα (χρησιμοποιώντας κλάσεις) στην Python είναι ένας κλασικός τρόπος επίδειξης του **Αντικειμενοστραφούς Προγραμματισμού (OOP)**. Χρησιμοποιούμε λίστες της Python ως τη βάση για την αποθήκευση δεδομένων, καθώς υποστηρίζουν αποτελεσματικά τις απαραίτητες λειτουργίες.

1. Υλοποίηση Στοίβας (Stack - LIFO)

Εκφώνηση

Να δημιουργηθεί η κλάση Stack η οποία θα αναπαριστά τη δομή δεδομένων Στοίβα (**L**ast **I**n, **F**irst **O**ut). Να υλοποιηθούν οι βασικές λειτουργίες:

1. **__init__**: Ο κατασκευαστής για την αρχικοποίηση της στοίβας.
2. **is_empty()**: Επιστρέφει True αν η στοίβα είναι άδεια.
3. **push(item)**: Προσθέτει ένα στοιχείο στην κορυφή της στοίβας.
4. **pop()**: Αφαιρεί και επιστρέφει το στοιχείο από την κορυφή της στοίβας. Αν η στοίβα είναι άδεια, να εμφανίζεται κατάλληλο μήνυμα.
5. **peek()**: Επιστρέφει το στοιχείο που βρίσκεται στην κορυφή, χωρίς να το αφαιρεί.

Απάντηση & Υλοποίηση Κώδικα

```
class Stack:
    """
    Υλοποίηση της δομής δεδομένων Στοίβα (Stack) με τη χρήση λίστας.
    Λειτουργεί με την αρχή LIFO (Last In, First Out).
    """

    # Κατασκευαστής
    def __init__(self):
        # Χρησιμοποιούμε μια λίστα της Python για την αποθήκευση των στοιχείων.
        # Η λίστα είναι ιδανική για Stack, καθώς το pop() και το append()
```

```
    # στο τέλος της λίστας είναι O(1) (πολύ γρήγορες).
    self._items = []
    print("-> Η Στοίβα δημιουργήθηκε.")

# Έλεγχος αν η Στοίβα είναι άδεια
def is_empty(self):
    """Ελέγχει αν η στοίβα δεν περιέχει στοιχεία."""
    return len(self._items) == 0

# Προσθήκη στοιχείου (Push)
def push(self, item):
    """Προσθέτει ένα στοιχείο στην κορυφή (τέλος) της στοίβας."""
    self._items.append(item)
    print(f"PUSH: Προστέθηκε '{item}'")

# Αφαίρεση στοιχείου (Pop)
def pop(self):
    """
    Αφαιρεί και επιστρέφει το στοιχείο από την κορυφή (τέλος) της στοίβας.
    Πετάει εξαίρεση ValueError αν η στοίβα είναι άδεια.
    """
    if self.is_empty():
        # Κατάλληλη διαχείριση σφάλματος για μια άδεια στοίβα.
        raise ValueError("Σφάλμα POP: Η Στοίβα είναι άδεια.")
    # Η μέθοδος pop() της λίστας αφαιρεί το τελευταίο στοιχείο.
    popped_item = self._items.pop()
    print(f"POP: Αφαιρέθηκε '{popped_item}'")
    return popped_item

# Επιστροφή κορυφής (Peek)
def peek(self):
    """Επιστρέφει το στοιχείο στην κορυφή (τέλος) χωρίς να το αφαιρεί."""
    if self.is_empty():
        raise ValueError("Σφάλμα PEEK: Η Στοίβα είναι άδεια.")
    # Επιστρέφουμε το τελευταίο στοιχείο της λίστας.
    return self._items[-1]
```

```
    # Εμφάνιση περιεχομένων (για ευκολία ελέγχου)
    def display(self):
        """Εμφανίζει τα περιεχόμενα της στοίβας, από κάτω προς την κορυφή."""
        print(f"Περιεχόμενα Στοίβας (Κάτω -> Κορυφή): {self._items}")

    def size(self):
        """Επιστρέφει το μέγεθος της στοίβας."""
        return len(self._items)

# --------------------------- Πρόγραμμα Επίδειξης Στοίβας ---------------------------
print("========== Επίδειξη Υλοποίησης Στοίβας (Stack - LIFO) ==========")
my_stack = Stack() # Δημιουργία αντικειμένου Στοίβας

# 1. Προσθήκη στοιχείων (PUSH)
my_stack.push("Βιβλίο Α")
my_stack.push("Βιβλίο Β")
my_stack.push("Βιβλίο Γ") # Τελευταίο μέσα (LIFO)

my_stack.display()
print(f"Μέγεθος: {my_stack.size()}")

# 2. Έλεγχος κορυφής (PEEK)
top_item = my_stack.peek()
print(f"PEEK: Το στοιχείο στην κορυφή είναι: '{top_item}'")

# 3. Αφαίρεση στοιχείων (POP)
my_stack.pop() # Αφαιρεί το 'Βιβλίο Γ'
my_stack.pop() # Αφαιρεί το 'Βιβλίο Β'

my_stack.display()

# 4. Έλεγχος αν είναι άδεια
print(f"Είναι η Στοίβα άδεια; {my_stack.is_empty()}")

# 5. Προσπάθεια αφαίρεσης από άδεια στοίβα (Διαχείριση σφάλματος)
try:
    my_stack.pop() # Αφαιρεί το 'Βιβλίο Α'
    my_stack.pop() # Θα προκαλέσει σφάλμα
```

```
except ValueError as e:
    print(f"Αποτυχία POP: {e}")

print("-" * 60)
```

2. Υλοποίηση Ουράς (Queue - FIFO)

Εκφώνηση

Να δημιουργηθεί η κλάση Queue η οποία θα αναπαριστά τη δομή δεδομένων Ουρά (**F**irst **I**n, **F**irst **O**ut). Να υλοποιηθούν οι βασικές λειτουργίες:

1. **__init__**: Ο κατασκευαστής για την αρχικοποίηση της ουράς.
2. **is_empty()**: Επιστρέφει True αν η ουρά είναι άδεια.
3. **enqueue(item)**: Προσθέτει ένα στοιχείο στο τέλος (πίσω) της ουράς.
4. **dequeue()**: Αφαιρεί και επιστρέφει το στοιχείο από την αρχή (μπροστά) της ουράς. Αν η ουρά είναι άδεια, να εμφανίζεται κατάλληλο μήνυμα.
5. **front()**: Επιστρέφει το στοιχείο που βρίσκεται στην αρχή, χωρίς να το αφαιρεί.

Απάντηση & Υλοποίηση Κώδικα

Σημείωση για την Python: Η χρήση της απλής λίστας για την υλοποίηση της Ουράς δεν είναι ιδανική, καθώς η αφαίρεση από την αρχή της λίστας (list.pop(0)) έχει πολυπλοκότητα O(n) (είναι αργή). Για αποδοτικότερη υλοποίηση, χρησιμοποιούμε την κλάση **collections.deque** (double-ended queue), η οποία επιτρέπει O(1) πολυπλοκότητα για πρόσθεση και αφαίρεση και στα δύο άκρα.

```
from collections import deque

class Queue:
    """
    Υλοποίηση της δομής δεδομένων Ουρά (Queue) με τη χρήση collections.deque.
    Λειτουργεί με την αρχή FIFO (First In, First Out).
    """

    # Κατασκευαστής
    def __init__(self):
```

```
        # Χρησιμοποιούμε deque (double-ended queue) για αποδοτικές πράξεις O(1)
        # σε πρόσθεση και αφαίρεση και στα δύο άκρα.
        self._items = deque()
        print("-> Η Ουρά δημιουργήθηκε.")

    # Έλεγχος αν η Ουρά είναι άδεια
    def is_empty(self):
        """Ελέγχει αν η ουρά δεν περιέχει στοιχεία."""
        return len(self._items) == 0

    # Προσθήκη στοιχείου (Enqueue) - Στο τέλος/πίσω
    def enqueue(self, item):
        """Προσθέτει ένα στοιχείο στο τέλος (δεξιά) της ουράς."""
        # Η μέθοδος append() προσθέτει στοιχείο στο δεξί άκρο (το πίσω μέρος της ουράς).
        self._items.append(item)
        print(f"ENQUEUE: Προστέθηκε '{item}'")

    # Αφαίρεση στοιχείου (Dequeue) - Από την αρχή/μπροστά
    def dequeue(self):
        """
        Αφαιρεί και επιστρέφει το στοιχείο από την αρχή (αριστερά) της ουράς.
        Πετάει εξαίρεση IndexError αν η ουρά είναι άδεια.
        """
        if self.is_empty():
            raise IndexError("Σφάλμα DEQUEUE: Η Ουρά είναι άδεια.")
        # Η μέθοδος popleft() αφαιρεί το στοιχείο από το αριστερό άκρο (το μπροστινό
μέρος της ουράς).
        dequeued_item = self._items.popleft()
        print(f"DEQUEUE: Εξυπηρετήθηκε '{dequeued_item}'")
        return dequeued_item

    # Επιστροφή αρχής (Front)
    def front(self):
        """Επιστρέφει το στοιχείο στην αρχή (αριστερά) χωρίς να το αφαιρεί."""
        if self.is_empty():
            raise IndexError("Σφάλμα FRONT: Η Ουρά είναι άδεια.")
        # Επιστρέφουμε το πρώτο στοιχείο.
```

```
        return self._items[0]

    # Εμφάνιση περιεχομένων
    def display(self):
        """Εμφανίζει τα περιεχόμενα της ουράς, από μπροστά προς πίσω."""
        print(f"Περιεχόμενα Ουράς (Μπροστά -> Πίσω): {list(self._items)}")

    def size(self):
        """Επιστρέφει το μέγεθος της ουράς."""
        return len(self._items)


# ---------------------------- Πρόγραμμα Επίδειξης Ουράς ----------------------------
print("\n========== Επίδειξη Υλοποίησης Ουράς (Queue - FIFO) ==========")
my_queue = Queue() # Δημιουργία αντικειμένου Ουράς

# 1. Προσθήκη στοιχείων (ENQUEUE)
my_queue.enqueue("Πελάτης Α") # Πρώτος μέσα (FIFO)
my_queue.enqueue("Πελάτης Β")
my_queue.enqueue("Πελάτης Γ")

my_queue.display()
print(f"Μέγεθος: {my_queue.size()}")

# 2. Έλεγχος αρχής (FRONT)
front_item = my_queue.front()
print(f"FRONT: Ο επόμενος πελάτης είναι: '{front_item}'")

# 3. Αφαίρεση στοιχείων (DEQUEUE)
my_queue.dequeue() # Εξυπηρετεί τον 'Πελάτη Α' (πρώτος μέσα)
my_queue.dequeue() # Εξυπηρετεί τον 'Πελάτη Β'

my_queue.display()

# 4. Έλεγχος αν είναι άδεια
print(f"Είναι η Ουρά άδεια; {my_queue.is_empty()}")

# 5. Προσπάθεια αφαίρεσης από άδεια ουρά (Διαχείριση σφάλματος)
```

```
try:
    my_queue.dequeue() # Εξυπηρετεί τον 'Πελάτη Γ'
    my_queue.dequeue() # Θα προκαλέσει σφάλμα
except IndexError as e:
    print(f"Αποτυχία DEQUEUE: {e}")
```

16.Όλα είναι αντικείμενα στην Python.

Στην Python, **τα πάντα είναι αντικείμενα** (objects). Αυτή είναι μια θεμελιώδης αρχή της γλώσσας, η οποία είναι πλήρως **αντικειμενοστραφής** (Object-Oriented).

Όλοι οι Βασικοί Τύποι Δεδομένων ως Αντικείμενα στην Python

Στην Python, όταν δημιουργείτε μια τιμή οποιουδήποτε βασικού τύπου δεδομένων (π.χ. έναν ακέραιο, μια συμβολοσειρά, μια λίστα), στην πραγματικότητα δημιουργείτε μια **στιγμιότυπη (instance)** μιας **κλάσης** (class) που ορίζει αυτόν τον τύπο.

1. **Κλάση και Αντικείμενο (Class and Object):**
    - Κάθε τύπος δεδομένων (όπως int, str, list, dict, tuple, set, float, bool) είναι στην πραγματικότητα μια **κλάση**.
    - Μια συγκεκριμένη τιμή (π.χ., ο αριθμός 5, η συμβολοσειρά "hello") είναι ένα **αντικείμενο** ή ένα **στιγμιότυπο** αυτής της κλάσης.
2. **Μέθοδοι (Methods):**
    - Εφόσον οι τιμές είναι αντικείμενα, κληρονομούν **ιδιότητες** και **μεθόδους** (συναρτήσεις που ανήκουν στο αντικείμενο) από την κλάση τους.
    - Αυτές οι μέθοδοι επιτρέπουν στο αντικείμενο να εκτελεί ενέργειες ή να επιστρέφει πληροφορίες για τον εαυτό του. Για παράδειγμα, μια συμβολοσειρά έχει μια μέθοδο upper() για να μετατρέπει τα γράμματά της σε κεφαλαία.
3. **Εσωτερική Υλοποίηση:**
    - Μπορείτε να επιβεβαιώσετε τον τύπο οποιασδήποτε τιμής χρησιμοποιώντας τη συνάρτηση type(), η οποία επιστρέφει την κλάση του αντικειμένου.

      ```
      a = 10
      print(type(a))  # Output: <class 'int'>
      s = "Python"
      print(type(s))  # Output: <class 'str'>
      ```
    - Όλα τα αντικείμενα στην Python έχουν ένα σύνολο **ειδικών μεθόδων** (ή "magic methods" όπως __add__, __len__, __init__) που καθορίζουν τη

συμπεριφορά τους (π.χ. πώς προστίθενται, πώς υπολογίζεται το μήκος τους, πώς δημιουργούνται).

10 Βασικότερες Μέθοδοι για Κάθε Βασικό Τύπο Δεδομένων

Ακολουθούν οι πιο βασικές μέθοδοι για τους πιο συνηθισμένους τύπους δεδομένων, με σύντομη περιγραφή και παράδειγμα.

1. String (str) (Αμετάβλητος Τύπος - Immutable)

Μέθοδος	**Περιγραφή**	**Παράδειγμα**
upper()	**Επιστρέφει μια νέα συμβολοσειρά με όλα τα γράμματα κεφαλαία.**	s = "hello".upper() **->** "HELLO"
lower()	**Επιστρέφει μια νέα συμβολοσειρά με όλα τα γράμματα πεζά.**	s = "WORLD".lower() **->** "world"
strip()	**Επιστρέφει μια νέα συμβολοσειρά με αφαίρεση των κενών (ή συγκεκριμένων χαρακτήρων) από την αρχή και το τέλος.**	s = " data ".strip() **->** "data"
split(sep)	**Χωρίζει τη συμβολοσειρά σε μια λίστα συμβολοσειρών, χρησιμοποιώντας τον** sep **ως διαχωριστικό. Αν παραλειφθεί, χρησιμοποιεί κενά.**	l = "A B C".split() **->** ['A', 'B', 'C']
join(iterable)	**Ενώνει τα στοιχεία ενός** iterable **(π.χ. λίστας) σε μια νέα συμβολοσειρά, χρησιμοποιώντας την αρχική συμβολοσειρά ως διαχωριστικό.**	s = "-".join(['1', '2']) **→** "1-2"
find(sub)	**Επιστρέφει τον δείκτη (index) της πρώτης εμφάνισης του υποστρίγνκ** sub**, ή** -1 **αν δεν βρεθεί.**	i = "abc".find("b") **→** 1
replace(old, new)	**Επιστρέφει μια νέα συμβολοσειρά αντικαθιστώντας όλες τις εμφανίσεις του** old **με το** new**.**	s = "dog cat".replace("dog", "fox") **→** "fox cat"
startswith(prefix)	**Επιστρέφει** True **αν η συμβολοσειρά ξεκινά με το δεδομένο πρόθεμα.**	b = "Python".startswith("Py")

Μέθοδος	Περιγραφή	Παράδειγμα
		→ True
endswith(suffix)	**Επιστρέφει** True **αν η συμβολοσειρά τελειώνει με το δεδομένο επίθεμα.**	b = "Python".endswith("on") → True
format(...)	**Μορφοποιεί τη συμβολοσειρά εισάγοντας τιμές σε συγκεκριμένες θέσεις (π.χ.** {}**).**	s = "My age is {}".format(30) → "My age is 30"

2. List (list) (Μεταβλητός Τύπος - Mutable)

Μέθοδος	Περιγραφή	Παράδειγμα
append(item)	**Προσθέτει ένα στοιχείο στο τέλος της λίστας.**	l = [1, 2]**;** l.append(3) → [1, 2, 3]
insert(index, item)	**Εισάγει ένα στοιχείο σε μια συγκεκριμένη θέση (**index**).**	l = [1, 3]**;** l.insert(1, 2) → [1, 2, 3]
extend(iterable)	**Προσθέτει όλα τα στοιχεία ενός** iterable **(π.χ. άλλης λίστας) στο τέλος της λίστας.**	l = [1]**;** l.extend([2, 3]) → [1, 2, 3]
remove(item)	**Αφαιρεί την πρώτη εμφάνιση ενός στοιχείου. Εγείρει** ValueError **αν δεν βρεθεί.**	l = [1, 2, 1]**;** l.remove(1) → [2, 1]
pop([index])	**Αφαιρεί και επιστρέφει το στοιχείο στη θέση** index**. Αν παραλειφθεί, αφαιρεί το τελευταίο.**	l = [1, 2]**;** x = l.pop() → x=2**,** l=[1]
index(item)	**Επιστρέφει τον δείκτη (index) της πρώτης εμφάνισης ενός στοιχείου.**	i = [10, 20].index(20) → 1
count(item)	**Επιστρέφει τον αριθμό των φορών που εμφανίζεται ένα στοιχείο στη λίστα.**	c = [1, 2, 1].count(1) → 2
sort()	**Ταξινομεί τη λίστα επί τόπου (in-place) σε αύξουσα σειρά.**	l = [3, 1]**;** l.sort() → [1, 3]
reverse()	**Αντιστρέφει τη σειρά των στοιχείων επί**	l = [1, 2]**;** l.reverse() → [2,

Μέθοδος	Περιγραφή	Παράδειγμα
	τόπου (in-place).	1]
clear()	**Αφαιρεί όλα τα στοιχεία από τη λίστα, αφήνοντάς την κενή.**	l = [1, 2]**;** l.clear() → []

3. Dictionary (dict) (Μεταβλητός Τύπος - Mutable)

Μέθοδος	Περιγραφή	Παράδειγμα
get(key, default)	**Επιστρέφει την τιμή για το** key**. Αν το** key **δεν υπάρχει, επιστρέφει το** default **(ή** None **αν παραλειφθεί), χωρίς να εγείρει σφάλμα.**	d={'a':1}**;** x=d.get('b', 0) → 0
keys()	**Επιστρέφει μια προβολή (view) όλων των κλειδιών του λεξικού.**	v = {'a':1}.keys() → dict_keys(['a'])
values()	**Επιστρέφει μια προβολή (view) όλων των τιμών του λεξικού.**	v = {'a':1}.values() → dict_values([1])
items()	**Επιστρέφει μια προβολή (view) όλων των ζευγαριών (κλειδί, τιμή) ως πλειάδες (tuples).**	v = {'a':1}.items() → dict_items([('a', 1)])
pop(key, default)	**Αφαιρεί και επιστρέφει την τιμή για το** key**. Αν το** key **δεν υπάρχει, επιστρέφει το** default**.**	d={'a':1}**;** v=d.pop('a') → v=1**,** d={}
popitem()	**Αφαιρεί και επιστρέφει το τελευταίο ζευγάρι (κλειδί, τιμή) ως πλειάδα (tuple).**	d={'a':1}**;** kv=d.popitem() → kv=('a', 1)**,** d={}
update(other_dict)	**Ενημερώνει το λεξικό προσθέτοντας ζευγάρια από ένα άλλο λεξικό (**other_dict**).**	d={'a':1}**;** d.update({'b':2}) → {'a':1, 'b':2}
clear()	**Αφαιρεί όλα τα ζευγάρια από το λεξικό, αφήνοντάς το κενό.**	d={'a':1}**;** d.clear() → {}
copy()	**Επιστρέφει ένα ρηχό αντίγραφο (shallow**	d1={'a':1}**;** d2=d1.copy() →

Μέθοδος	Περιγραφή	Παράδειγμα
	copy) του λεξικού.	d2={'a':1}
setdefault(key, default)	**Επιστρέφει την τιμή του** key**. Αν το** key **δεν υπάρχει, το εισάγει με την τιμή** default **και επιστρέφει αυτή την τιμή.**	d={'a':1}**;** x=d.setdefault('b', 2) **→** x=2**,** d={'a':1, 'b':2}

4. Tuple (tuple) (Αμετάβλητος Τύπος - Immutable)

Οι πλειάδες, επειδή είναι **αμετάβλητες**, έχουν πολύ λίγες μεθόδους.

Μέθοδος	Περιγραφή	Παράδειγμα
count(item)	**Επιστρέφει τον αριθμό των φορών που εμφανίζεται ένα στοιχείο στην πλειάδα.**	c = (1, 2, 1).count(1) **→** 2
index(item)	**Επιστρέφει τον δείκτη (index) της πρώτης εμφάνισης ενός στοιχείου. Εγείρει** ValueError **αν δεν βρεθεί.**	i = (10, 20).index(20) **→** 1
(Δεν υπάρχουν άλλες βασικές μέθοδοι λόγω της αμετάβλητης φύσης τους)		
Σημείωση: Συνήθεις ενέργειες γίνονται με ενσωματωμένες συναρτήσεις (len()**,** sorted()**,** max()**,** min()**)**		

5. Set (set) (Μεταβλητός Τύπος - Mutable)

Τα σύνολα αποθηκεύουν **μοναδικά** στοιχεία και είναι **μη ταξινομημένα**.

Μέθοδος	Περιγραφή	Παράδειγμα
add(item)	**Προσθέτει ένα στοιχείο στο σύνολο. Αγνοείται αν το στοιχείο υπάρχει ήδη.**	s = {1, 2}**;** s.add(3) **→** {1, 2, 3}

Μέθοδος	Περιγραφή	Παράδειγμα
remove(item)	**Αφαιρεί ένα στοιχείο. Εγείρει** KeyError **αν το στοιχείο δεν υπάρχει.**	s = {1, 2}**;** s.remove(2) **→** {1}
discard(item)	**Αφαιρεί ένα στοιχείο. Δεν εγείρει σφάλμα αν το στοιχείο δεν υπάρχει.**	s = {1, 2}**;** s.discard(3) **→** {1, 2}
pop()	**Αφαιρεί και επιστρέφει ένα τυχαίο στοιχείο από το σύνολο.**	s = {1, 2}**;** x = s.pop() **→** x **είναι** 1 **ή** 2**,** s **έχει το άλλο.**
union(other_set)	**Επιστρέφει ένα νέο σύνολο που περιέχει όλα τα στοιχεία από το σύνολο και το** other_set **(ένωση).**	u = {1}.union({2}) **→** {1, 2}
intersection(other_set)	**Επιστρέφει ένα νέο σύνολο με τα κοινά στοιχεία (τομή).**	i = {1, 2}.intersection({2, 3}) **→** {2}
difference(other_set)	**Επιστρέφει ένα νέο σύνολο με τα στοιχεία που υπάρχουν στο πρώτο σύνολο αλλά όχι στο** other_set **(διαφορά).**	d = {1, 2}.difference({2, 3}) **→** {1}
issubset(other_set)	**Επιστρέφει** True **αν όλα τα στοιχεία του συνόλου υπάρχουν στο** other_set**.**	b = {1}.issubset({1, 2}) **→** True
issuperset(other_set)	**Επιστρέφει** True **αν το σύνολο περιέχει όλα τα στοιχεία του** other_set**.**	b = {1, 2}.issuperset({1}) **→** True
update(iterable)	**Προσθέτει στο σύνολο όλα τα μοναδικά στοιχεία από ένα** iterable**.**	s = {1}**;** s.update([2, 3, 2]) **→** {1, 2, 3}

6. Int & Float & Bool (Αμετάβλητοι Τύποι - Immutable)

Οι αριθμητικοί τύποι και ο Boolean έχουν **ελάχιστες** άμεσα ορατές μεθόδους, καθώς οι περισσότερες λειτουργίες γίνονται με τους **αριθμητικούς τελεστές** (π.χ. +, -, *). Όμως, έχουν ειδικές μεθόδους (Dunder methods) και κάποιες χρήσιμες.

Int (int)

Μέθοδος	Περιγραφή	Παράδειγμα
bit_length()	**Επιστρέφει τον ελάχιστο αριθμό bits που απαιτούνται για την αναπαράσταση του ακεραίου στο δυαδικό σύστημα (χωρίς το πρόσημο).**	b = 10.bit_length() → 4 **(το 10 είναι 1010_2)**
to_bytes(length, byteorder)	**Επιστρέφει μια αναπαράσταση bytes του ακεραίου.**	b = (10).to_bytes(2, 'big') → b'\x00\n'

Float (float)

Μέθοδος	Περιγραφή	Παράδειγμα
is_integer()	**Επιστρέφει** True **αν ο δεκαδικός αριθμός είναι ακέραιος (π.χ. 5.0).**	b = (5.0).is_integer() → True
as_integer_ratio()	**Επιστρέφει μια πλειάδα** (numerator, denominator) **που ισούται με τον αριθμό.**	r = (0.5).as_integer_ratio() → (1, 2)
hex()	**Επιστρέφει μια αναπαράσταση του αριθμού σε δεκαεξαδική μορφή.**	s = (10.0).hex() → '0x1.4000000000000p+3'

Bool (bool)

Ο τύπος bool είναι υποκλάση του int (True είναι 1, False είναι 0), οπότε κληρονομεί τις μεθόδους του int, αλλά σπάνια χρησιμοποιούνται.

Μέθοδος	Περιγραφή	Παράδειγμα
__and__(other)	**Αντιστοιχεί στον τελεστή λογικού AND (&).**	b = True.__and__(False) → False
__or__(other)	**Αντιστοιχεί στον τελεστή λογικού OR (`**	**`).**

Μέθοδος	Περιγραφή	Παράδειγμα
__xor__(other)	**Αντιστοιχεί στον τελεστή λογικού XOR (^).**	b = True.__xor__(True) → False
(Δεν υπάρχουν άλλες βασικές μέθοδοι πέραν των λογικών τελεστών που υλοποιούνται μέσω ειδικών μεθόδων)		

17. Αλγόριθμοι

17.1 Αναζήτηση.

Παρακάτω περιγράφονται και υλοποιούνται οι αλγόριθμοι **Σειριακής (Γραμμικής) Αναζήτησης** (Linear Search) και **Δυαδικής Αναζήτησης** (Binary Search) στην Python.

1. Σειριακή (Γραμμική) Αναζήτηση (Linear Search)

Περιγραφή Αλγορίθμου

Η Σειριακή Αναζήτηση είναι η πιο **απλή** μέθοδος αναζήτησης. Λειτουργεί ως εξής:

1. Ξεκινά από το πρώτο στοιχείο της λίστας.
2. Συγκρίνει διαδοχικά κάθε στοιχείο με την **τιμή-στόχο** (target value).
3. Αν βρει την τιμή, επιστρέφει τη **θέση** (δείκτη/index) όπου βρέθηκε.
4. Αν φτάσει στο τέλος της λίστας χωρίς να βρει την τιμή, επιστρέφει ένα σήμα (συνήθως -1) ότι η τιμή δεν υπάρχει.
5. **Δεν απαιτεί** η λίστα να είναι ταξινομημένη.

Υλοποίηση σε Python

```
def linear_search(data_list, target_value):
    """
    Αναζητά μια τιμή σε μια λίστα διαδοχικά.
    Επιστρέφει τον δείκτη (index) της τιμής ή -1 αν δεν βρεθεί.
    """
    # Χρησιμοποιούμε τη συνάρτηση enumerate για να έχουμε ταυτόχρονα
    # τον δείκτη (index) και την τιμή (item) κάθε στοιχείου.
    for index, item in enumerate(data_list):
        if item == target_value:
            return index  # Επιστρέφει τον δείκτη μόλις βρεθεί
    return -1  # Αν ολοκληρωθεί ο βρόχος, σημαίνει ότι η τιμή δεν βρέθηκε
```

Τρόπος Κλήσης και Χρήσης

```
# 1. Δημιουργία δεδομένων (μη ταξινομημένα)
data = [15, 30, 10, 5, 25]
target = 10

# 2. Κλήση του αλγορίθμου
result_index = linear_search(data, target)

# 3. Χρήση του αποτελέσματος
if result_index != -1:
   print(f"Σειριακή Αναζήτηση: Η τιμή {target} βρέθηκε στον δείκτη {result_index}.")
else:
   print(f"Σειριακή Αναζήτηση: Η τιμή {target} δεν βρέθηκε.")
```

Χρησιμότητα

Η Σειριακή Αναζήτηση είναι χρήσιμη σε:

- **Μικρές λίστες**, όπου η ταχύτητα εκτέλεσης δεν είναι κρίσιμη.
- **Μη ταξινομημένες λίστες**, καθώς δεν απαιτεί προεπεξεργασία (ταξινόμηση).
- Όταν είναι απαραίτητο να βρεθεί η **πρώτη εμφάνιση** ενός στοιχείου.
- Η πολυπλοκότητα χρόνου είναι O(n) (Γραμμική).

2. Δυαδική Αναζήτηση (Binary Search)

Περιγραφή Αλγορίθμου

Η Δυαδική Αναζήτηση είναι ένας πολύ **αποδοτικός** αλγόριθμος, αλλά απαιτεί η λίστα να είναι **ταξινομημένη**. Λειτουργεί ως εξής:

1. Ορίζει ένα κάτω όριο (low) και ένα άνω όριο (high) για το διάστημα αναζήτησης.
2. Υπολογίζει το **μεσαίο στοιχείο** (mid) του τρέχοντος διαστήματος.
3. Συγκρίνει το μεσαίο στοιχείο με την **τιμή-στόχο**.

- Αν είναι ίσα, η αναζήτηση ολοκληρώθηκε.
- Αν η τιμή-στόχος είναι **μεγαλύτερη**, αγνοεί το αριστερό μισό και συνεχίζει την αναζήτηση στο δεξί μισό (θέτει low = mid + 1).
- Αν η τιμή-στόχος είναι **μικρότερη**, αγνοεί το δεξί μισό και συνεχίζει την αναζήτηση στο αριστερό μισό (θέτει high = mid - 1).

4. Επαναλαμβάνει τη διαδικασία μέχρι να βρει την τιμή ή μέχρι το διάστημα αναζήτησης να αδειάσει.

Υλοποίηση σε Python

```
def binary_search(sorted_list, target_value):
    """
    Αναζητά μια τιμή σε μια ΤΑΞΙΝΟΜΗΜΕΝΗ λίστα, διαιρώντας το διάστημα στο μισό.
    Επιστρέφει τον δείκτη (index) της τιμής ή -1 αν δεν βρεθεί.
    """
    low = 0                  # Κάτω όριο (αρχή της λίστας)
    high = len(sorted_list) - 1 # Άνω όριο (τέλος της λίστας)

    while low <= high:
        mid = (low + high) // 2  # Υπολογισμός του μεσαίου δείκτη (ακέραια διαίρεση)
        guess = sorted_list[mid] # Η τιμή στο μεσαίο σημείο

        if guess == target_value:
            return mid           # Βρέθηκε η τιμή
        elif guess < target_value:
            low = mid + 1        # Η τιμή-στόχος είναι μεγαλύτερη, ψάχνουμε στο δεξί μισό
        else: # guess > target_value
            high = mid - 1       # Η τιμή-στόχος είναι μικρότερη, ψάχνουμε στο αριστερό μισό

    return -1                    # Αν ο βρόχος τελειώσει, η τιμή δεν βρέθηκε
```

Τρόπος Κλήσης και Χρήσης

```
# 1. Δημιουργία δεδομένων (ΠΡΕΠΕΙ να είναι ταξινομημένα!)
sorted_data = [5, 10, 15, 20, 25, 30]
target = 20

# 2. Κλήση του αλγορίθμου
result_index = binary_search(sorted_data, target)

# 3. Χρήση του αποτελέσματος
if result_index != -1:
    print(f"Δυαδική Αναζήτηση: Η τιμή {target} βρέθηκε στον δείκτη {result_index}.")
else:
    print(f"Δυαδική Αναζήτηση: Η τιμή {target} δεν βρέθηκε.")
```

Χρησιμότητα

Η Δυαδική Αναζήτηση είναι η **προτιμώμενη** μέθοδος αναζήτησης για:

- **Μεγάλες συλλογές δεδομένων** που είναι ήδη ταξινομημένες.
- Εφαρμογές όπου η **ταχύτητα** είναι κρίσιμη.
- Η πολυπλοκότητα χρόνου είναι O(log n) (Λογαριθμική), καθιστώντας την πολύ πιο γρήγορη από τη Σειριακή Αναζήτηση για μεγάλες εισόδους.

17.2 Ταξινόμηση.

Οι δύο απλούστεροι και κλασικότεροι αλγόριθμοι ταξινόμησης είναι η **Ταξινόμηση Φυσαλίδας (Bubble Sort)** και η **Ταξινόμηση Επιλογής (Selection Sort)**.

1. Ταξινόμηση Φυσαλίδας (Bubble Sort)

Περιγραφή Αλγορίθμου

Ο αλγόριθμος περνά επανειλημμένα μέσα από τη λίστα, **συγκρίνει** κάθε ζεύγος διπλανών στοιχείων και τα **ανταλλάσσει** αν βρίσκονται σε λάθος σειρά. Σε κάθε πέρασμα, το μεγαλύτερο μη ταξινομημένο στοιχείο "ανεβαίνει" σαν φυσαλίδα στο τέλος της μη ταξινομημένης περιοχής. Η διαδικασία επαναλαμβάνεται μέχρι να μη γίνει καμία ανταλλαγή σε ένα πέρασμα, υποδεικνύοντας ότι η λίστα είναι ταξινομημένη.

Υλοποίηση σε Python

```
def bubble_sort(data_list):
    """
    Ταξινομεί μια λίστα χρησιμοποιώντας τον αλγόριθμο Bubble Sort.
    Η ταξινόμηση γίνεται επί τόπου (in-place).
    """
    n = len(data_list)

    # Βρόχος για κάθε πέρασμα (pass)
    for i in range(n - 1):
        swapped = False # Σημαία για βελτιστοποίηση: αν δεν έγινε ανταλλαγή, η λίστα είναι ταξινομημένη

        # Βρόχος για σύγκριση διπλανών στοιχείων
        # Το (n - i - 1) μειώνει τον έλεγχο, εξαιρώντας τα ήδη ταξινομημένα στο τέλος
        for j in range(n - i - 1):
            # Αν το τρέχον στοιχείο είναι μεγαλύτερο από το επόμενο, τα ανταλλάσσουμε
            if data_list[j] > data_list[j + 1]:
                # Ανταλλαγή
                data_list[j], data_list[j + 1] = data_list[j + 1], data_list[j]
                swapped = True

        # Αν δεν έγινε καμία ανταλλαγή στο πέρασμα, σταματάμε
        if not swapped:
            break

    return data_list
```

Τρόπος Κλήσης και Χρήσης

```
my_list = [5, 1, 4, 2, 8]

# Κλήση
sorted_list = bubble_sort(my_list)
```

```
# Χρήση
print(f"Bubble Sort: Ταξινομημένη λίστα: {sorted_list}")
# Έξοδος: Bubble Sort: Ταξινομημένη λίστα: [1, 2, 4, 5, 8]
```

Χρησιμότητα

- **Χρήση:** Κυρίως για **εκπαιδευτικούς σκοπούς** λόγω της απλής και άμεσης λογικής του. Είναι εύκολο να κατανοηθεί πώς λειτουργεί η ταξινόμηση.
- **Πολυπλοκότητα:** Έχει πολυπλοκότητα χρόνου O(n^2) (Τετραγωνική), καθιστώντας τον **πολύ αργό** για μεγάλες λίστες.

2. Ταξινόμηση Επιλογής (Selection Sort)

Περιγραφή Αλγορίθμου

Ο αλγόριθμος λειτουργεί διαιρώντας τη λίστα σε δύο μέρη: την **ταξινομημένη** περιοχή (αριστερά) και τη **μη ταξινομημένη** περιοχή (δεξιά). Λειτουργεί ως εξής:

1. Βρίσκει το **μικρότερο** (ή μεγαλύτερο) στοιχείο στη μη ταξινομημένη περιοχή.
2. Το ανταλλάσσει με το **πρώτο** στοιχείο της μη ταξινομημένης περιοχής (δηλαδή, το τοποθετεί στο τέλος της ταξινομημένης περιοχής).
3. Μετακινεί το όριο μεταξύ των δύο περιοχών κατά μία θέση προς τα δεξιά.
4. Επαναλαμβάνει μέχρι να ταξινομηθεί ολόκληρη η λίστα.

Υλοποίηση σε Python

```
def selection_sort(data_list):
    """
    Ταξινομεί μια λίστα χρησιμοποιώντας τον αλγόριθμο Selection Sort.
    Η ταξινόμηση γίνεται επί τόπου (in-place).
    """
    n = len(data_list)

    # Βρόχος για το όριο της ταξινομημένης περιοχής
    for i in range(n):
```

```
        # Βρίσκουμε τον δείκτη του ελάχιστου στοιχείου στην υπόλοιπη (μη ταξινομημένη) λίστα
        min_idx = i

        # Βρόχος για την εύρεση του ελάχιστου
        for j in range(i + 1, n):
            if data_list[j] < data_list[min_idx]:
                min_idx = j

        # Ανταλλάσσουμε το ελάχιστο στοιχείο που βρέθηκε
        # με το πρώτο στοιχείο της μη ταξινομημένης περιοχής (στη θέση i)
        data_list[i], data_list[min_idx] = data_list[min_idx], data_list[i]

    return data_list
```

Τρόπος Κλήσης και Χρήσης

```
my_list = [64, 25, 12, 22, 11]

# Κλήση
sorted_list = selection_sort(my_list)

# Χρήση
print(f"Selection Sort: Ταξινομημένη λίστα: {sorted_list}")
# Έξοδος: Selection Sort: Ταξινομημένη λίστα: [11, 12, 22, 25, 64]
```

Χρησιμότητα

- **Χρήση:** Όταν ο **αριθμός των ανταλλαγών** πρέπει να ελαχιστοποιηθεί. Ο Selection Sort κάνει το πολύ n ανταλλαγές, κάτι που μπορεί να είναι σημαντικό αν οι ανταλλαγές είναι "ακριβές" (π.χ. σε μεγάλες δομές δεδομένων).
- **Πολυπλοκότητα:** Έχει επίσης πολυπλοκότητα χρόνου O(n^2), καθιστώντας τον επίσης **αργό** για μεγάλες λίστες, αλλά είναι πιο αποδοτικός από τον Bubble Sort σε όρους μνήμης (λόγω λιγότερων εγγραφών).