ΓΙΑ ΑΝΘΡΩΠΟΥΣ ΠΟΥ ΞΕΡΟΥΝ ΗΔΗ RUBY

ΑΠΟ RUBY ΣΕ PYTHON



Από το πρώτο script μέχρι intermediate Python.

19 ΚΕΦΑΛΑΙΑ / 57 ΑΣΚΗΣΕΙΣ / ΠΛΗΡΕΙΣ ΛΥΣΕΙΣ

PYTHON 3.14+ · ΜΕ ΤΙΣ ΠΡΟΣΘΗΚΕΣ ΤΗΣ 3.15 · 2026

Ο ΧΑΡΤΗΣ ΤΟΥ ΒΙΒΛΙΟΥ

Περιεχόμενα

Πριν ανοίξεις τον editor 3
1 Πρώτη επαφή χωρίς να ξεκινάς από το μηδέν 5
2 Τιμές, ονόματα και truthiness 8
3 Strings, bytes και κείμενο που δεν σε προδίδει 11
4 Lists, tuples, dicts και sets 14
5 Control flow και loops 18
6 Functions που λένε καθαρά πώς καλούνται 21
7 Από το Enumerable στα comprehensions 25
8 Iterators και generators: ένα στοιχείο κάθε φορά 28
9 Exceptions και resource cleanup 31
10 Modules, packages και μικρά προγράμματα 35
11 Classes, dataclasses και protocols 38
12 Type hints που βοηθούν χωρίς να λένε ψέματα 42
13 Pattern matching: διάβασε τη δομή των δεδομένων 46
14 Closures και decorators χωρίς μυστήριο 49
15 Αρχεία και δεδομένα με τη standard library 53
16 Tests και debugging που ελέγχουν τη σωστή υπόθεση 56
17 Async και await: να ξέρεις τι περιμένει τι 59
18 Η σημερινή Python: 3.14 και 3.15 63
19 Τελικό project: ένα μικρό readlog 67
Λύσεις και τρόπος σκέψης 72
Ruby → Python: γρήγορη αναφορά 93
Πηγές και επόμενη ανάγνωση 97

ΟΔΗΓΟΣ ΑΝΑΓΝΩΣΗΣ

Πριν ανοίξεις τον editor

Ξέρεις ήδη να προγραμματίζεις. Έχεις γράψει Ruby, έχεις περάσει blocks σε methods και έχεις αλλάξει ένα Hash χωρίς να το σκεφτείς ιδιαίτερα. Αυτό το βιβλίο χρησιμοποιεί αυτή τη γνώση ως αφετηρία. Στόχος είναι να διαβάζεις και να γράφεις άνετα καθημερινή Python, μέχρι και το intermediate συντακτικό της, καταλαβαίνοντας τι πραγματικά εκτελείται.

Το δύσκολο στη μετάβαση δεν είναι να θυμηθείς ότι το `elsif` γράφεται `elif`. Είναι ότι το `[]` είναι falsey, το τελευταίο expression μιας function δεν επιστρέφεται μόνο του, το `yield` δημιουργεί generator και μια default λίστα μπορεί να επιβιώσει ανάμεσα σε κλήσεις. Εδώ θα σταθούμε αρκετά.

Τι θα μπορείς να κάνεις

Στο τέλος θα μπορείς να σχεδιάζεις functions με σωστά arguments, να διαλέγεις collections, να χρησιμοποιείς comprehensions χωρίς να φτιάχνεις αινίγματα και να ξεχωρίζεις iterable από iterator. Θα μπορείς να οργανώνεις μικρό package, να γράφεις classes και dataclasses, να διαβάζεις σύγχρονα type hints, decorators, pattern matching και `async` κώδικα. Το τελικό project ενώνει αυτά τα κομμάτια σε ένα εργαλείο που εκτελείται από το terminal.

Το βιβλίο δεν προϋποθέτει Rails. Δεν διδάσκει web framework, machine learning ή deployment. Το ζητούμενο είναι η ίδια η γλώσσα και η standard library που θα χρησιμοποιείς ανεξάρτητα από το επόμενο framework σου.

Πώς να το δουλέψεις

Διάβασε τα κεφάλαια με τη σειρά την πρώτη φορά. Γράψε τα μικρά παραδείγματα, πρόβλεψε το αποτέλεσμα και μετά τρέξε τα. Τα `assert` μέσα στα παραδείγματα δηλώνουν ακριβώς ποια συμπεριφορά περιμένουμε. Αν είναι σωστή, δεν εμφανίζουν τίποτα. Αν είναι λάθος, αποτυγχάνουν με `AssertionError`.

Κάθε κεφάλαιο έχει τρεις ασκήσεις. Η πρώτη ελέγχει το νοητικό μοντέλο, η δεύτερη ζητά μικρή υλοποίηση και η τρίτη βάζει μια χρήσιμη παραλλαγή ή διόρθωση. Υπάρχει hint χωρίς ολόκληρη τη λύση. Οι αναλυτικές λύσεις είναι συγκεντρωμένες στο τέλος, για να μπορείς να προσπαθήσεις χωρίς να σου αποκαλύπτεται ο κώδικας στην επόμενη παράγραφο.

Μην κυνηγήσεις τη συντομότερη απάντηση. Μια απλή function με σαφή ονόματα είναι καλύτερη άσκηση από ένα expression που καταλαβαίνεις μόνο τη στιγμή που το γράφεις. Δοκίμαζε τουλάχιστον μία συνηθισμένη περίπτωση και μία οριακή: κενή λίστα, μηδέν, απουσία τιμής, διπλό στοιχείο ή διαφορετικό μήκος εισόδου.

Εκδόσεις και έλεγχος

Η ενημέρωση του βιβλίου έγινε στις **16 Οκτωβρίου 2026**. Η νεότερη σταθερή σειρά που ανακοινώνει το python.org είναι η **Python 3.15**, με την 3.15.0 να έχει κυκλοφορήσει στις 9 Οκτωβρίου. Ο κορμός του βιβλίου και το τελικό project στοχεύουν **Python 3.14+**. Έτσι χρησιμοποιούμε τη σημερινή γλώσσα χωρίς να απαιτούμε την απολύτως τελευταία έκδοση για κάθε άσκηση.

Τα εκτελέσιμα παραδείγματα του κορμού και οι λύσεις ελέγχονται με **CPython 3.14.7**. Το κεφάλαιο 18 ξεχωρίζει τις προσθήκες της 3.15. Τα αντίστοιχα snippets έχουν ελεγχθεί απέναντι στην επίσημη τεκμηρίωση, αλλά δεν έχουν εκτελεστεί τοπικά σε interpreter 3.15. Δεν παρουσιάζονται ως δοκιμασμένα στην 3.14.

Οι συγκρίσεις αφορούν σύγχρονη Ruby 3.x και τον βασικό πυρήνα της. Το `blank?`, το `present?` και το `try` του Active Support δεν είναι ενσωματωμένες δυνατότητες της Ruby. Τα ονόματα και τα δεδομένα των ασκήσεων είναι συνθετικά.

Ο χάρτης

Διαδρομή	Κεφάλαια	Τι χτίζεις
Η μετάβαση	1-5	Setup, τιμές, strings, collections, loops
Η καθημερινή γλώσσα	6-10	Functions, comprehensions, generators, errors, modules
Intermediate Python	11-14	Classes, types, patterns, decorators
Από τη γνώση στην πράξη	15-19	Αρχεία, tests, async, νέες εκδόσεις, project

Στο τέλος υπάρχουν **57 ασκήσεις με λύσεις**, πίνακας Ruby → Python και κριτήρια αυτοελέγχου. Το συνοδευτικό αρχείο με τον κώδικα περιέχει τα παραδείγματα, τις λύσεις και το project. Όλες οι βασικές ασκήσεις λύνονται με τη standard library, χωρίς εγκατάσταση τρίτου package.

Οι πρωτογενείς πηγές βρίσκονται στο τελευταίο παράρτημα, οργανωμένες ανά θέμα. Είναι χρήσιμες για δεύτερη ανάγνωση και για επαλήθευση όταν αναβαθμίζεις Python.

ΚΕΦΑΛΑΙΟ 1

Πρώτη επαφή χωρίς να ξεκινάς από το μηδέν

Στη Ruby μπορείς να γράψεις ένα σύντομο script και να το τρέξεις με `ruby file.rb`. Στην Python κάνεις το ίδιο με `python file.py`. Πριν όμως συζητήσουμε σύνταξη, θέλουμε να ξέρουμε ποιος interpreter εκτελεί το αρχείο μας.

Setup με virtual environment

Εγκατάστησε Python 3.14 ή 3.15 από τον επίσημο διανομέα ή τον package manager που ήδη εμπιστεύεσαι. Το βιβλίο δεν χρειάζεται συγκεκριμένο editor. Από terminal σε macOS ή Linux:

TERMINAL

```
python3 --version
mkdir python-lab
cd python-lab
python3 -m venv .venv
source .venv/bin/activate
python --version
python -c "import sys; print(sys.executable)"
```

Σε Windows, το αντίστοιχο ξεκίνημα σε PowerShell είναι `py -3 -m venv .venv`. Μετά μπορείς να εκτελείς απευθείας `.venv\Scripts\python.exe`, ακόμη κι αν η ενεργοποίηση του περιβάλλοντος δεν επιτρέπεται από την πολιτική του shell.

Το virtual environment κρατά τα packages ενός project χωριστά από τα packages των άλλων. Δεν είναι container και δεν περιορίζει την πρόσβαση του προγράμματος στο filesystem. Η ενεργοποίηση απλώς αλλάζει την προτεραιότητα των executables στο PATH. Αν χρειαστείς package αργότερα, χρησιμοποίησε `python -m pip ...` ώστε το pip να ανήκει στον interpreter που μόλις έλεγξες. Για το βιβλίο δεν χρειάζεται να εγκαταστήσεις κάτι.

Το πρώτο πέρασμα από Ruby

RUBY

```
def greeting(name)
  "Hello, #{name}!"
end

puts greeting("Ada")
```

PYTHON

```
def greeting(name):
    return f"Hello, {name}!"

print(greeting("Ada"))
assert greeting("Ada") == "Hello, Ada!"
```

Το `:` ανοίγει το block και το indentation ορίζει ποιες γραμμές ανήκουν σε αυτό. Χρησιμοποιούμε τέσσερα spaces ανά επίπεδο και αποφεύγουμε την ανάμειξη tabs και spaces. Δεν υπάρχει `end`. Τα parentheses στην κλήση είναι απαραίτητα: το `greeting` αναφέρεται στη function, το `greeting("Ada")` την καλεί.

Η διαφορά που αξίζει να μείνει είναι το `return`. Μια Python function που φτάνει στο τέλος της χωρίς `return` επιστρέφει `None`. Το ίδιο κάνει ένα σκέτο `return`.

PYTHON

```
def almost_greeting(name):
    f"Hello, {name}!"

assert almost_greeting("Ada") is None
```

Το expression υπολογίζεται και το αποτέλεσμα εγκαταλείπεται. Δεν είναι syntax error. Είναι έγκυρο πρόγραμμα με διαφορετική συμπεριφορά από αυτή που συνηθίζεις στη Ruby.

Statements, expressions και συνέχεια γραμμής

Ένα expression δίνει τιμή: `2 + 3`, μια κλήση function, ένα conditional expression. Ένα statement οργανώνει εκτέλεση: assignment, `if`, `for`, `return`. Το `if` ως block δεν επιστρέφει τιμή όπως ένα Ruby `if`. Όταν θέλεις τιμή, γράφεις `a if condition else b`.

PYTHON

```
points = 12
label = "ready" if points >= 10 else "practice"
assert label == "ready"

total = (
    15
    + 20
    + 7
)
assert total == 42
```

Μέσα σε parentheses, brackets και braces μπορείς να συνεχίσεις σε νέα γραμμή χωρίς backslash. Αυτό βοηθά όταν μια κλήση έχει πολλά arguments. Τα σχόλια αρχίζουν με #. Ένα string στην πρώτη θέση του σώματος μιας function είναι docstring, που διαβάζεται μέσω `help()` και εργαλείων τεκμηρίωσης.

Τα ονόματα δεν κρύβουν κλήσεις

Τα κοινά ονόματα ακολουθούν `snake_case`, οι classes `PascalCase` και οι συμβατικές constants `UPPER_CASE`. Δεν υπάρχουν method names με τελικό `?` ή `!`. Το όνομα `is_ready` είναι σύμβαση, όχι ειδική σύνταξη. Το `MAX_SIZE = 100` δεν εμποδίζει μεταγενέστερο assignment.

Πρόσεχε επίσης να μην ονομάζεις μεταβλητές `list`, `str`, `sum` ή `id`: κάνεις shadowing στα built-in ονόματα μέσα στο ίδιο scope. Ο interpreter το επιτρέπει, αλλά μετά το `list(values)` μπορεί να προσπαθεί να καλέσει τη δική σου λίστα.

> **Κράτα αυτό:** η Python αφήνει λιγότερα πράγματα να εννοούνται στη βασική κλήση. Γράφεις τα parentheses, γράφεις το `return` και αργότερα θα γράφεις ρητά το `self`.

Ασκήσεις

Λύσεις κεφαλαίου 1 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

1.1 · Το χαμένο αποτέλεσμα

Τι επιστρέφει η παρακάτω function; Διόρθωσέ τη ώστε το `double(6)` να δίνει `12`.

```python
def double(number):
    number * 2
```

Hint: το ότι υπολογίζεται μια τιμή δεν σημαίνει ότι γίνεται return.

1.2 · Μια κάρτα ονόματος

Γράψε `badge(name, language)` που επιστρέφει `"Ada · Python"` για τα arguments `"Ada"`, `"Python"`. Μην τυπώνεις μέσα στη function. Έλεγξε και το `badge("Matz", "Ruby")`.

Hint: θέλεις ένα f-string και ένα `return`.

1.3 · Διάβασε το indentation

Γράψε `access_label(age)` που επιστρέφει `"adult"` όταν το age είναι τουλάχιστον 18 και `"minor"` αλλιώς. Δοκίμασε 17, 18 και 19. Θεώρησε ότι το input είναι ήδη έγκυρος ακέραιος.

Hint: μία έκδοση με `if` και μία με conditional expression είναι εξίσου σωστές.

ΚΕΦΑΛΑΙΟ 2

Τιμές, ονόματα και truthiness

Στην Python, όπως και στη Ruby, οι μεταβλητές αναφέρονται σε objects. Το assignment συνδέει ένα όνομα με ένα object. Δεν αντιγράφει αυτόματα ό,τι βρίσκεται στη δεξιά πλευρά.

Το ίδιο object ή η ίδια τιμή;

PYTHON

```
original = ["ruby"]
alias = original
copy = original.copy()
alias.append("python")

assert original == ["ruby", "python"]
assert alias is original
assert copy == ["ruby"]
assert copy is not original
```

Το == συγκρίνει ισότητα, σύμφωνα με τον τύπο του object. Το `is` συγκρίνει ταυτότητα: είναι το ίδιο ακριβώς object; Στη Ruby η κοντινή αναλογία του `is` είναι το `equal?`, όχι το ==.

Γράφε `value is None` και `value is not None`. Μην χρησιμοποιείς `is` για αριθμούς ή strings. Κάποιες μικρές τιμές επαναχρησιμοποιούνται εσωτερικά και μπορεί να φαίνεται ότι η σύγκριση δουλεύει, μέχρι να πάψει να δουλεύει με άλλο input ή interpreter.

Truthy και falsey: οι διαφορές από τη Ruby

Στη Ruby μόνο τα `nil` και `false` είναι falsey. Στην Python falsey είναι επίσης το αριθμητικό μηδέν και οι κενές built-in collections. Τα υπόλοιπα objects είναι συνήθως truthy, εκτός αν ο τύπος τους ορίζει διαφορετική συμπεριφορά μέσω `__bool__` ή `__len__`.

Τιμή	Ruby	Python
`nil` / `None`	falsey	falsey
`false` / `False`	falsey	falsey
`0`, `0.0`	truthy	falsey
`""`, `[]`, `{}`	truthy	falsey
`"0"`, `"False"`	truthy	truthy

PYTHON

```
assert bool(0) is False
assert bool([]) is False
assert bool("False") is True

requested = 0
wrong = requested or 10
right = 10 if requested is None else requested
assert wrong == 10
assert right == 0
```

Το `x or default` είναι σωστό μόνο όταν *κάθε* falsey τιμή πρέπει να αντικατασταθεί. Αν το μηδέν ή το κενό string έχουν νόημα, έλεγξε ειδικά το `None`. Δεν υπάρχει operator `??` ή σύνταξη `||=` στον βασικό κορμό της Python.

Τα `and` και `or` κάνουν short-circuit και επιστρέφουν έναν από τους operands, όχι υποχρεωτικά boolean. Το `not` δίνει boolean. Η Python έχει επίσης `&` και `|`, αλλά αυτά είναι bitwise operators για integers και έχουν άλλες σημασίες σε sets και ορισμένους άλλους τύπους. Δεν αντικαθιστούν τα `and` και `or`.

Αριθμοί χωρίς σιωπηρές εκπλήξεις

Οι συνηθισμένοι ακέραιοι είναι `int` και αυξάνονται μέχρι το όριο της διαθέσιμης μνήμης. Το `float` χρησιμοποιεί floating-point αναπαράσταση, οπότε δεν μπορεί να αποθηκεύσει με ακρίβεια κάθε δεκαδική τιμή.

PYTHON

```
assert 7 / 2 == 3.5
assert 7 // 2 == 3
assert -7 // 2 == -4
assert -7 % 2 == 1
assert 2 ** 5 == 32
assert divmod(17, 5) == (3, 2)

from math import isclose
assert isclose(0.1 + 0.2, 0.3)
```

Το `/` μεταξύ integers δίνει float. Το `//` είναι floor division: στρογγυλεύει προς το αρνητικό άπειρο, όχι προς το μηδέν. Το `%` συνδέεται με αυτή τη διαίρεση μέσω `a == (a // b) * b + a % b`. Το `**` είναι ύψωση σε δύναμη.

Η μετατροπή `int("12")` είναι ρητή. Το `int("12x")` αποτυγχάνει με `ValueError`, δεν επιστρέφει το αρχικό αριθμητικό κομμάτι όπως το Ruby `"12x".to_i`. Το `str(12)` φτιάχνει string. Το `bool("false")` δεν είναι parser boolean configuration: δίνει `True` επειδή το string δεν είναι κενό.

Συγκρίσεις που διαβάζονται φυσικά

```python
age = 27
assert 18 <= age < 65
assert "py" in "python"
assert "java" not in ["ruby", "python"]
assert (1 == True) is True
```

Τα chained comparisons δεν σημαίνουν `(18 <= age) < 65`. Ελέγχουν και τα δύο όρια, υπολογίζοντας το μεσαίο expression μία φορά. Το `bool` είναι subtype του `int`, γι' αυτό το τελευταίο assert ισχύει. Αυτό έχει σημασία στο validation και στα dict keys: τα `True` και `1` δεν αποτελούν δύο διαφορετικά keys. Συνήθως δεν χρειάζεται να το εκμεταλλευτείς, χρειάζεται να μην εκπλαγείς.

> **Παγίδα από Ruby:** το `if items` στην Python σημαίνει «υπάρχει truthy τιμή», που για μια λίστα σημαίνει «δεν είναι κενή». Το `if items is not None` σημαίνει κάτι διαφορετικό: αποδέχεται και την κενή λίστα.

Ασκήσεις

Λύσεις κεφαλαίου 2 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

2.1 · Μηδέν ή απουσία;

Για καθεμία από τις τιμές `None`, `0`, `5`, πρόβλεψε το αποτέλεσμα του `value or 20`. Μετά γράψε `default_limit(value)` που χρησιμοποιεί 20 μόνο όταν το value είναι `None`.

Hint: οι δύο εκδοχές διαφέρουν ακριβώς στο μηδέν.

2.2 · Χρόνος ανάγνωσης

Γράψε `split_minutes(total)` που δέχεται μη αρνητικό ακέραιο και επιστρέφει tuple `(hours, minutes)`. Για 135 θέλουμε `(2, 15)`. Για 0 θέλουμε `(0, 0)`.

Hint: το `divmod` κάνει και τις δύο πράξεις μαζί.

2.3 · Δύο ονόματα, μία λίστα

Ξεκίνα με `a = [1, 2]`, `b = a`, `c = a.copy()`. Κάνε `b.append(3)` και μετά `b = [9]`. Ποιες είναι τώρα οι τρεις λίστες; Εξήγησε γιατί το δεύτερο assignment δεν αλλάζει το `a`.

Hint: ξεχώρισε το mutation από το rebinding ενός ονόματος.

ΚΕΦΑΛΑΙΟ 3

Strings, bytes και κείμενο που δεν σε προδίδει

Η Python χωρίζει καθαρά το κείμενο από τα bytes. Ένα `str` κρατά Unicode κείμενο. Ένα `bytes` κρατά μια σειρά από bytes. Η διάκριση είναι χρήσιμη σε αρχεία, δίκτυο και encodings, ακόμη κι αν τα περισσότερα καθημερινά strings σου είναι απλά αγγλικά.

Immutable strings

Στη Ruby μπορείς να αλλάξεις ένα String με <<, εκτός αν είναι frozen. Τα Python strings είναι immutable. Οι methods επεξεργασίας επιστρέφουν νέο string. Δεν υπάρχει `strip!`.

PYTHON

```
raw_name = "  Ada  "
name = raw_name.strip()
assert name == "Ada"
assert raw_name == "  Ada  "
assert name.lower() == "ada"
assert "book.py".removesuffix(".py") == "book"
```

Τα μονά και τα διπλά εισαγωγικά συμπεριφέρονται με τον ίδιο τρόπο. Και στα δύο λειτουργούν escapes όπως `\n`. Χρησιμοποίησε το είδος που κάνει το string ευανάγνωστο. Τα triple quotes επιτρέπουν πολλές γραμμές και χρησιμοποιούνται συχνά στα docstrings. Το prefix `r` δημιουργεί raw string, χρήσιμο σε patterns, αλλά ένα raw string δεν μπορεί να τελειώνει με μονό backslash.

Slicing και indexing

PYTHON

```
word = "python"
assert word[0] == "p"
assert word[-1] == "n"
assert word[1:4] == "yth"
assert word[:2] == "py"
assert word[::2] == "pto"
assert word[::-1] == "nohtyp"
assert word[100:] == ""
```

Το slice είναι `start:stop:step`. Το `stop` αποκλείεται. Όταν λείπει το step, είναι 1. Τα slices ανέχονται όρια έξω από το string, ενώ ένα μεμονωμένο `word[100]` σηκώνει `IndexError`. Το ίδιο μοντέλο slicing θα χρησιμοποιήσεις σε lists και tuples.

Το `len()` ενός string μετρά Unicode code points. Δεν μετρά bytes και δεν εγγυάται τον αριθμό των ορατών χαρακτήρων. Ένα γράμμα και ένας συνδυαστικός τόνος μπορούν να είναι δύο code points που εμφανίζονται ως ένα γράμμα. Για σύγκριση κειμένου από διαφορετικές πηγές μπορεί να χρειάζεται normalization.

PYTHON

```
import unicodedata

composed = "é"
decomposed = "e\u0301"
assert composed != decomposed
assert len(decomposed) == 2
assert unicodedata.normalize("NFC", decomposed) == composed
```

Από interpolation σε f-strings

PYTHON

```
title = "Python"
pages = 128
progress = 0.375
assert f"{title}: {pages} pages" == "Python: 128 pages"
assert f"{progress:.1%}" == "37.5%"
assert f"{pages:04d}" == "0128"
assert f"{title!r}" == "'Python'"
assert f"{pages=}" == "pages=128"
```

Μέσα στα `{}` μπαίνει expression. Μετά το `:` μπαίνει format specification. Το `!r` ζητά `repr`, χρήσιμο όταν θέλεις να βλέπεις εισαγωγικά ή escapes στο debugging. Το `=` εμφανίζει και το όνομα του expression. Για κυριολεκτικά braces γράφεις `{{` και `}}`.

Μη συγχέεις το formatting με μετατροπή δεδομένων ή με ασφαλή παραγωγή query. Το f-string δίνει κείμενο. Δεν κάνει escaping για HTML ούτε parameter binding για SQL. Θα χρησιμοποιήσεις την αντίστοιχη λειτουργία της library που καταναλώνει το κείμενο.

Split, join και αναζήτηση

PYTHON

```
assert "  one   two\nthree ".split() == ["one", "two", "three"]
assert "a,,b".split(",") == ["a", "", "b"]
assert ", ".join(["Ruby", "Python"]) == "Ruby, Python"
assert "name=Ada".partition("=") == ("name", "=", "Ada")
assert "name".partition("=") == ("name", "", "")
assert "Python".casefold() == "python"
```

Το `.split()` χωρίς separator αντιμετωπίζει συνεχόμενα whitespace ως ένα όριο. Το `.split(",")` διατηρεί τα κενά πεδία. Το `.join()` ανήκει στο separator, όχι στη

λίστα όπως το Ruby `array.join`. Τα στοιχεία πρέπει να είναι strings, αλλιώς θα πάρεις `TypeError`.

Το `in` αρκεί για απλό έλεγχο παρουσίας. Το `.find()` δίνει index ή `-1`, ενώ το `.index()` σηκώνει `ValueError` αν δεν βρει το substring. Το `.casefold()` είναι πιο κατάλληλο από το `.lower()` για Unicode σύγκριση χωρίς διάκριση πεζών και κεφαλαίων, χωρίς να λύνει μόνο του κάθε θέμα τοπικών κανόνων και normalization.

Το όριο με τα bytes

PYTHON

```
text = "γεια"
payload = text.encode("utf-8")
assert isinstance(payload, bytes)
assert payload.decode("utf-8") == text
assert len(payload) > len(text)
```

Κάνεις encode όταν μετατρέπεις κείμενο σε bytes και decode για την αντίστροφη μετατροπή. Το encoding πρέπει να συμφωνεί με την πηγή, όχι να επιλεγεί επειδή κάνει το error να εξαφανιστεί. Το `str(payload)` δεν αντικαθιστά το `.decode()`.

Ασκήσεις

Λύσεις κεφαλαίου 3 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

3.1 · Ποιο split;

Για το `" red  blue "`, πρόβλεψε τα αποτελέσματα του `.split()` και του `.split(" ")`. Ποιο θα διάλεγες για λέξεις από ελεύθερο κείμενο;

Hint: ο ρητός separator διατηρεί και κενά πεδία.

3.2 · Καθαρά tags

Γράψε `clean_tags(text)` που μετατρέπει το `" Ruby, python , , RUBY "` σε `["ruby", "python", "ruby"]`. Κάνε strip και casefold, πέτα τα κενά. Μην αφαιρέσεις ακόμη τα διπλότυπα. Ένα απλό loop αρκεί.

Hint: χώρισε με κόμμα και έλεγξε την καθαρισμένη τιμή.

3.3 · Μικρή προεπισκόπηση

Γράψε `preview(text, limit)` που επιστρέφει το ίδιο string αν χωρά σε `limit` code points. Αλλιώς επιστρέφει τους πρώτους `limit` και μετά `"..."`. Θεώρησε `limit >= 0`. Για `preview("python", 3)` θέλουμε `"pyt..."`, για `preview("hi", 3)` θέλουμε `"hi"`.

Hint: το limit αφορά το κομμάτι του αρχικού κειμένου, όχι τις τρεις τελείες.

ΚΕΦΑΛΑΙΟ 4

Lists, tuples, dicts και sets

Στη Ruby χρησιμοποιείς συχνά Array και Hash. Στην Python διαλέγεις list όταν θέλεις να προσθέτεις και να αλλάζεις στοιχεία, tuple για να κρατήσεις μαζί τιμές που δεν θα αντικαταστήσεις, dict για key/value pairs και set για μοναδικές τιμές.

List: κοντά στο Ruby Array

PYTHON

```
languages = ["ruby", "python"]
languages.append("go")
languages.extend(["rust", "elixir"])
assert languages[-2:] == ["rust", "elixir"]
last = languages.pop()
assert last == "elixir"
assert languages.index("python") == 1
```

Το `append` προσθέτει *ένα* object, το `extend` τα στοιχεία ενός iterable. Το `append([1, 2])` δημιουργεί nested list. Το `extend([1, 2])` προσθέτει δύο στοιχεία. Το `pop()` αφαιρεί και επιστρέφει, ενώ το `remove(value)` αφαιρεί την πρώτη ίση τιμή και αποτυγχάνει αν δεν υπάρχει.

PYTHON

```
scores = [8, 3, 5]
ordered = sorted(scores)
assert ordered == [3, 5, 8]
assert scores == [8, 3, 5]
result = scores.sort()
assert scores == [3, 5, 8]
assert result is None
```

Η `sorted` δίνει νέα λίστα από οποιοδήποτε κατάλληλο iterable. Η `.sort()` αλλάζει την υπάρχουσα λίστα και επιστρέφει `None`. Το ίδιο κάνουν πολλές methods που κάνουν mutation, όπως `append` και `reverse`. Άρα το `scores = scores.sort()` χάνει τη σύνδεση με τη λίστα.

Tuple: το κόμμα κάνει τη διαφορά

PYTHON

```
point = (10, 20)
x, y = point
single = (10,)
assert x == 10 and y == 20
assert len(single) == 1
first, *middle, last = [1, 2, 3, 4]
assert middle == [2, 3]
```

Η tuple δεν επιτρέπει αντικατάσταση των στοιχείων της. Αυτό δεν κάνει immutable κάθε object που περιέχει: μια tuple μπορεί να κρατά μια mutable list. Χωρίς starred target, το unpacking απαιτεί ίδιο πλήθος στοιχείων και ονομάτων. Με `*middle`, τα περισσευούμενα συγκεντρώνονται σε λίστα.

Πολλαπλό return, όπως `return total, count`, δημιουργεί tuple. Το `(10)` είναι απλώς ο αριθμός 10 μέσα σε parentheses. Το `(10,)` είναι tuple ενός στοιχείου.

Dict: πιο αυστηρό στο missing key

PYTHON

```
book = {"title": "Python", "pages": 160}
assert book["title"] == "Python"
assert book.get("author") is None
assert book.get("author", "unknown") == "unknown"
assert "pages" in book

book["pages"] = 180
merged = book | {"pages": 200, "edition": 2}
assert merged["pages"] == 200
assert book["pages"] == 180
assert list(book) == ["title", "pages"]
```

Το `book["missing"]` σηκώνει `KeyError`, αντί να δώσει `nil` όπως ένα συνηθισμένο Ruby Hash χωρίς ειδικό default. Το `.get()` δίνει προαιρετικό default αλλά δεν ξεχωρίζει από μόνο του missing key από υπάρχον key με τιμή `None`. Για αυτή τη διάκριση χρησιμοποίησε membership ή sentinel.

Το `in` ελέγχει keys. Το iteration σε dict δίνει keys. Τα `.items()` δίνουν ζεύγη key/value και τα `.values()` τιμές. Η σειρά εισαγωγής διατηρείται. Αυτό δεν σημαίνει ότι το dict είναι ταξινομημένο. Το `|` δημιουργεί νέο merged dict με τις δεξιές τιμές να υπερισχύουν. Το `.update()` αλλάζει το αρχικό.

Τα keys πρέπει να είναι hashable. Strings, integers και tuples από hashable στοιχεία είναι συνηθισμένα keys. Lists και dicts δεν είναι. Δεν υπάρχει ξεχωριστός τύπος Symbol όπως στη Ruby. Για συνηθισμένα ονόματα πεδίων χρησιμοποιούμε strings.

Set: παρουσία και μοναδικότητα

PYTHON

```
known = {"ruby", "python", "ruby"}
wanted = {"python", "rust"}
assert known & wanted == {"python"}
assert wanted - known == {"rust"}
assert known | wanted == {"ruby", "python", "rust"}
assert len(set()) == 0
```

Τα sets δεν έχουν indexing και δεν υπόσχονται σειρά εμφάνισης. Το `{}` είναι κενό dict, όχι set. Για κενό set γράφεις `set()`. Για immutable set υπάρχει `frozenset`. Η συνηθισμένη χρήση είναι membership και πράξεις συνόλων, όχι διατήρηση της σειράς ενός αρχείου.

Shallow copies και κοινά references

PYTHON

```
rows = [[1], [2]]
cloned = rows.copy()
cloned[0].append(9)
assert rows == [[1, 9], [2]]

bad_grid = [[0] * 2] * 3
bad_grid[0][0] = 7
assert bad_grid == [[7, 0], [7, 0], [7, 0]]
```

Η εξωτερική λίστα έχει αντιγραφεί, οι εσωτερικές λίστες όχι. Αυτό είναι shallow copy. Αντίστοιχα, ο πολλαπλασιασμός λίστας επαναλαμβάνει references. Για ανεξάρτητες γραμμές φτιάξε νέα εσωτερική λίστα κάθε φορά. Το `copy.deepcopy()` είναι διαθέσιμο όταν θέλεις deep copy, αλλά πρώτα ξεκαθάρισε ποια objects χρειάζεται να παραμένουν κοινά.

Ασκήσεις

Λύσεις κεφαλαίου 4 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

4.1 · Διόρθωσε το sorting

Γιατί το `names = names.sort()` δεν αφήνει μια sorted λίστα στο `names`; Δώσε μία λύση που αλλάζει την αρχική λίστα και μία που την αφήνει ανέπαφη.

Hint: mutation και return value είναι δύο διαφορετικές αποφάσεις.

4.2 · Μοναδικά με σειρά

Γράψε `unique_in_order(values)` που μετατρέπει το `["py", "rb", "py", "go", "rb"]` σε `["py", "rb", "go"]`. Θεώρησε hashable στοιχεία. Για κενή είσοδο θέλουμε κενή λίστα.

Hint: ένα dict κρατά τη σειρά της πρώτης εισαγωγής των keys.

4.3 · Ανεξάρτητες σειρές

Φτιάξε λίστα τριών ανεξάρτητων λιστών `[0, 0]`. Άλλαξε το πρώτο στοιχείο της πρώτης σε 7 και έλεγξε ότι οι άλλες δύο παραμένουν `[0, 0]`.

Hint: βάλε το `[0, 0]` μέσα στο σώμα ενός loop και κάνε append.

ΚΕΦΑΛΑΙΟ 5

Control flow και loops

Το Ruby `each` περνά ένα block σε method. Το Python `for` είναι σύνταξη της γλώσσας και ζητά διαδοχικά στοιχεία από ένα iterable. Το αποτέλεσμα φαίνεται παρόμοιο, αλλά θα έχει σημασία όταν φτάσουμε στα generators.

If, elif και μικρές αποφάσεις

PYTHON

```python
def reading_band(minutes):
    if minutes >= 60:
        return "long"
    elif minutes >= 20:
        return "steady"
    else:
        return "short"

assert reading_band(45) == "steady"
```

Το `elif` αποφεύγει περιττό nesting. Μπορείς επίσης να χρησιμοποιείς early returns. Δεν υπάρχει `unless`: γράψε `if not condition`. Οι guard clauses είναι εξίσου χρήσιμες εδώ όσο στη Ruby. Απόφυγε όμως διπλές αρνήσεις που σε αναγκάζουν να ξαναδιαβάζεις τη συνθήκη.

For πάνω στα δεδομένα

RUBY

```ruby
["Ada", "Matz"].each_with_index do |name, index|
  puts "#{index + 1}. #{name}"
end
```

PYTHON

```python
names = ["Ada", "Matz"]
labels = []
for number, name in enumerate(names, start=1):
    labels.append(f"{number}. {name}")
assert labels == ["1. Ada", "2. Matz"]

pages = {"Ruby": 40, "Python": 60}
total = 0
for title, count in pages.items():
    total += count
assert total == 100
```

Αν θέλεις μόνο την τιμή, κάνε iteration στις τιμές. Αν χρειάζεσαι και το index, χρησιμοποίησε `enumerate`. Το `range(len(names))` είναι χρήσιμο όταν δουλεύεις με τα ίδια τα indices. Δεν χρειάζεται για να περάσεις απλώς από τα στοιχεία.

Το `range(start, stop, step)` αποκλείει το `stop` και δεν κατασκευάζει λίστα με όλους τους αριθμούς. Τα `range(3)` και `range(0, 3)` περιγράφουν τις τιμές 0, 1, 2. Για να μετρήσεις προς τα πίσω, χρησιμοποίησε αρνητικό step, όπως `range(5, 0, -1)`.

Zip: iteration σε δύο iterables μαζί

PYTHON

```
titles = ["Ruby", "Python"]
minutes = [15, 30]
pairs = list(zip(titles, minutes, strict=True))
assert pairs == [("Ruby", 15), ("Python", 30)]
```

Το συνηθισμένο `zip(a, b)` σταματά μόλις εξαντληθεί η πιο μικρή είσοδος. Αν τα δεδομένα *πρέπει* να έχουν ίδιο μήκος, το `strict=True` εντοπίζει τη διαφορά με `ValueError` κατά την κατανάλωση. Το ίδιο το `zip` είναι lazy: η κατασκευή του δεν διαβάζει αμέσως όλα τα στοιχεία.

Break, continue και το else του loop

PYTHON

```
values = [3, -1, 8, 2]
found = None
for value in values:
    if value < 0:
        continue
    if value >= 7:
        found = value
        break
else:
    found = "missing"

assert found == 8
```

Το `continue` προχωρά στο επόμενο iteration. Το `break` βγαίνει από το πιο εσωτερικό loop. Το `else` του `for` εκτελείται όταν το loop τελειώσει χωρίς `break`, ακόμη και αν η είσοδος ήταν κενή. Δεν σημαίνει «το loop δεν έτρεξε». Αν γίνει `return` ή ξεφύγει exception, δεν φτάνουμε σε αυτό το `else`.

Το `while` επαναλαμβάνει όσο η συνθήκη είναι truthy. Υποστηρίζει επίσης `break`, `continue` και `else`. Φρόντισε να υπάρχει τρόπος να αλλάξει η συνθήκη, εκτός αν γράφεις σκόπιμα infinite loop με έξοδο μέσω `break`.

Assignment expression με μέτρο

PYTHON

```
line = "  python  "
if (cleaned := line.strip()):
    assert cleaned == "python"
```

Το `:=` κάνει assignment και επιστρέφει την ίδια τιμή ως expression. Είναι χρήσιμο όταν θες να κρατήσεις ένα αποτέλεσμα και να το ελέγξεις στην ίδια συνθήκη. Δεν αντικαθιστά γενικά το `=`. Αν κάνει τη γραμμή δυσκολότερη, χώρισε assignment και condition.

> **Πρόσεχε το mutation κατά το iteration.** Αν αφαιρείς στοιχεία από τη λίστα πάνω στην οποία κάνεις iteration, μπορεί να προσπεράσεις κάποια. Φτιάξε νέα λίστα ή κάνε iteration σε αντίγραφο. Η αλλαγή μεγέθους dict κατά το iteration συνήθως προκαλεί `RuntimeError`.

Ασκήσεις

Λύσεις κεφαλαίου 5 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

5.1 · Το else που τρέχει

Γράψε loop που ψάχνει το 4 στο `[1, 2, 3]`. Με `for/else`, όρισε `message = "found"` αν βρεθεί και `"missing"` αλλιώς. Δοκίμασε και κενή λίστα.

Hint: το `break` είναι το στοιχείο που ακυρώνει το `else`.

5.2 · Η πρώτη αρκετά μεγάλη τιμή

Γράψε `first_at_least(values, limit)` που επιστρέφει την πρώτη τιμή τουλάχιστον ίση με το όριο ή `None`. Για `[2, 8, 5]` και 5 θέλουμε 8, όχι τη μικρότερη κατάλληλη τιμή.

Hint: ένα early return μέσα στο loop αρκεί.

5.3 · Δύο στήλες που πρέπει να ταιριάζουν

Γράψε `make_scores(names, scores)` που επιστρέφει dict αντιστοιχίζοντας κάθε name στο score που βρίσκεται στην ίδια θέση. Τα ονόματα θεωρούνται μοναδικά. Αν διαφέρουν τα μήκη, θέλουμε `ValueError`. Δοκίμασε και δύο κενές λίστες.

Hint: το `dict()` μπορεί να καταναλώσει ζεύγη από `zip`.

ΚΕΦΑΛΑΙΟ 6

Functions που λένε καθαρά πώς καλούνται

Μια Python function είναι object. Μπορείς να την αποθηκεύσεις, να την περάσεις ως argument και να την επιστρέψεις. Δεν χρειάζεσαι `Proc` ή `lambda` για να πάρεις reference σε μια function που έχει ήδη όνομα: απλώς χρησιμοποιείς το όνομά της χωρίς parentheses.

Positional και keyword arguments

PYTHON

```
def describe(title, pages=0, *, finished=False):
    state = "done" if finished else "reading"
    return f"{title}: {pages} ({state})"

assert describe("Python", 80) == "Python: 80 (reading)"
assert describe("Ruby", pages=40, finished=True) == (
    "Ruby: 40 (done)"
)
```

Τα arguments πριν από το `*` μπορούν εδώ να δοθούν positional ή με όνομα. Τα arguments μετά το σκέτο `*` είναι keyword-only. Έτσι το `finished=True` δεν γίνεται ένα αινιγματικό τρίτο positional boolean. Αν χρειάζεσαι positional-only parameters, τοποθετείς `/` μετά από αυτά.

PYTHON

```
def scale(value, /, factor=2, *, offset=0):
    return value * factor + offset

assert scale(3, factor=4, offset=1) == 13
```

Η signature διαβάζεται από αριστερά: το `value` μόνο positional, το `factor` με τους δύο τρόπους, το `offset` μόνο με όνομα. Θα συναντήσεις το `/` στην τεκμηρίωση πολλών built-ins. Δεν χρειάζεται να το βάζεις σε κάθε δική σου function.

Args, kwargs και unpacking

PYTHON

```
def capture(first, *rest, **options):
    return first, rest, options

assert capture(1, 2, 3, mode="fast") == (
    1, (2, 3), {"mode": "fast"}
)

numbers = [3, 4]
settings = {"offset": 1}
assert scale(*numbers, **settings) == 13
```

Το `*rest` συλλέγει τα υπόλοιπα positional arguments σε tuple. Το `**options` συλλέγει τα υπόλοιπα keyword arguments σε dict. Τα ονόματα `args` και `kwargs` είναι συμβατικά, όχι keywords. Στην *κλήση*, τα `*` και `**` κάνουν την αντίστροφη δουλειά: ξεδιπλώνουν iterable και mapping. Τα keys του mapping των keyword arguments πρέπει να είναι strings. Αν δώσεις δύο φορές το ίδιο argument, παίρνεις `TypeError`.

Το default που θυμάται

PYTHON

```
def collect_bad(item, bucket=[]):
    bucket.append(item)
    return bucket

first_result = collect_bad("ruby")
second_result = collect_bad("python")
assert first_result is second_result
assert second_result == ["ruby", "python"]
```

Το `[]` αξιολογείται όταν εκτελείται το `def`, μία φορά για αυτόν τον ορισμό. Δεν ξαναδημιουργείται σε κάθε κλήση. Αυτό διαφέρει από default argument expressions στη Ruby, που αξιολογούνται στην κλήση όταν λείπει το argument.

PYTHON

```
def collect(item, bucket=None):
    if bucket is None:
        bucket = []
    bucket.append(item)
    return bucket

assert collect("ruby") == ["ruby"]
assert collect("python") == ["python"]
```

Τώρα δημιουργούμε λίστα μέσα στην κλήση. Αν ο caller περάσει λίστα, η function εξακολουθεί σκόπιμα να την αλλάζει. Αν θέλεις πάντα νέο αποτέλεσμα, αυτό είναι άλλη απόφαση: ξεκίνα από `list(bucket)` όταν υπάρχει bucket.

Scope: ποιο όνομα βλέπω;

Η συνηθισμένη αναζήτηση ονόματος περνά από local scope, enclosing function scopes, module globals και built-ins. Ένα assignment μέσα στη function κάνει το όνομα local, εκτός αν δηλωθεί `global` ή `nonlocal`. Γι' αυτό το να διαβάσεις ένα όνομα και αργότερα να του κάνεις assignment μπορεί να προκαλέσει `UnboundLocalError`.

PYTHON

```
def make_counter():
    count = 0

    def increment():
        nonlocal count
        count += 1
        return count

    return increment

counter = make_counter()
assert counter() == 1
assert counter() == 2
```

Το `nonlocal` αναφέρεται σε υπάρχον binding της εξωτερικής function. Το `global` αναφέρεται στο module. Οι functions που θυμούνται το περιβάλλον τους λέγονται closures. Αν απλώς κάνεις mutation σε μια λίστα του εξωτερικού scope, δεν χρειάζεσαι `nonlocal`: δεν αλλάζεις το binding, αλλά το object.

Τα `if` και `for` δεν δημιουργούν ξεχωριστό local scope. Το loop variable ενός `for` μπορεί να παραμένει διαθέσιμο μετά το loop. Τα comprehensions έχουν δικό τους scope για τα loop variables. Θα το δούμε αμέσως μετά.

Μικρή function ως argument

PYTHON

```
def apply_twice(function, value):
    return function(function(value))

def add_one(value):
    return value + 1

assert apply_twice(add_one, 3) == 5
assert apply_twice(lambda n: n * 2, 3) == 12
```

Το Python `lambda` περιορίζεται σε ένα expression. Δεν είναι ο γενικός αντικαταστάτης ενός πολυγραμμικού Ruby block. Όταν χρειάζεσαι περισσότερα βήματα ή ένα όνομα που εξηγεί τη δουλειά, χρησιμοποίησε `def`.

Ασκήσεις

Λύσεις κεφαλαίου 6 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

6.1 · Διάβασε τη signature

Για τη `scale` του κεφαλαίου, ποιες κλήσεις είναι έγκυρες: `scale(3)`, `scale(value=3)`, `scale(3, 4, 1)`, `scale(3, 4, offset=1)`; Γράψε και τα αποτελέσματα των έγκυρων.

Hint: `/` και `*` ορίζουν διαφορετικά όρια.

6.2 · Tags χωρίς κοινό mutable state

Γράψε `add_tag(tag, tags=None)` που επιστρέφει νέα λίστα με το tag στο τέλος. Δεν πρέπει να αλλάζει λίστα που πέρασε ο caller. Διαδοχικές κλήσεις χωρίς tags πρέπει να είναι ανεξάρτητες.

Hint: νέο default και αντίγραφο του input είναι δύο ξεχωριστές απαιτήσεις.

6.3 · Μια function που φτιάχνει functions

Γράψε `make_multiplier(factor)` που επιστρέφει function. Το `triple = make_multiplier(3)` πρέπει να επιτρέπει `triple(4) == 12`. Φτιάξε και διπλασιαστή και έλεγξε ότι οι δύο functions κρατούν διαφορετικό factor.

Hint: το εσωτερικό `def` μπορεί να διαβάσει το argument της εξωτερικής function.

ΚΕΦΑΛΑΙΟ 7

Από το Enumerable στα comprehensions

Αν γράφεις Ruby, πιθανότατα σκέφτεσαι συχνά με `map`, `select`, `reject`, `group_by` και `each_with_object`. Στην Python, μεγάλο μέρος αυτής της δουλειάς εκφράζεται με comprehensions, built-ins και λίγους στοχευμένους τύπους από το `collections`.

List comprehension: τι παράγω, από πού, πότε

RUBY

```ruby
names = [" Ada ", "", " Matz "]
clean = names.map(&:strip).reject(&:empty?)
```

PYTHON

```python
names = [" Ada ", "", " Matz "]
trimmed = [name.strip() for name in names]
clean = [name for name in trimmed if name]
assert clean == ["Ada", "Matz"]
```

Η βασική μορφή είναι `[expression for item in source if condition]`. Πρώτα σκέψου τη διαδρομή `for` και το φίλτρο `if`, μετά την τιμή που παράγεται στην αρχή. Το φίλτρο στο τέλος αφαιρεί στοιχεία. Το conditional expression στην αρχή διαλέγει μια τιμή για *κάθε* στοιχείο.

PYTHON

```python
values = [-2, 0, 4]
positive = [n for n in values if n > 0]
clamped = [n if n > 0 else 0 for n in values]
assert positive == [4]
assert clamped == [0, 0, 4]
```

Μπορείς να γράψεις όλο τον καθαρισμό ονομάτων σε μία γραμμή με assignment expression, αλλά δεν είναι υποχρεωτικό. Η διπλή γραμμή κάνει σαφές τι μετασχηματίζεται και τι φιλτράρεται. Μην χρησιμοποιείς comprehension μόνο για side effects, όπως `[print(x) for x in values]`: φτιάχνεις άχρηστη λίστα από `None`.

Dict και set comprehensions

PYTHON

```
words = ["ruby", "python", "ruby"]
lengths = {word: len(word) for word in words}
initials = {word[0] for word in words}
assert lengths == {"ruby": 4, "python": 6}
assert initials == {"r", "p"}
```

Στο dict, η τελευταία τιμή ενός επαναλαμβανόμενου key κερδίζει. Στο set, τα διπλότυπα εξαφανίζονται. Τα parentheses αντί για brackets φτιάχνουν generator expression, που δεν υπολογίζει όλα τα αποτελέσματα αμέσως.

Τα built-ins που αξίζει να μάθεις νωρίς

PYTHON

```
scores = [8, 3, 5]
assert sum(scores) == 16
assert min(scores) == 3
assert max([], default=0) == 0
assert any(n > 7 for n in scores)
assert all(n >= 0 for n in scores)
assert all([]) is True
assert any([]) is False
```

Τα `any` και `all` σταματούν μόλις ξέρουν την απάντηση. Η κενή είσοδος του `all` δίνει `True`: δεν βρέθηκε αντιπαράδειγμα. Αν θέλεις να υπάρχει τουλάχιστον ένα στοιχείο, χρειάζεται να ελέγξεις και ότι το input δεν είναι κενό.

Οι `map` και `filter` υπάρχουν, αλλά επιστρέφουν iterators. Το `map(str, numbers)` είναι καθαρή χρήση. Για `map(lambda ...)`, ένα comprehension συνήθως διαβάζεται καλύτερα. Το `filter(None, values)` αφαιρεί *όλες* τις falsey τιμές. Δεν είναι μετάφραση του Ruby `compact`, που αφαιρεί μόνο τα `nil`.

PYTHON

```
values = [None, 0, False, "", 5]
compact = [value for value in values if value is not None]
assert compact == [0, False, "", 5]
assert list(filter(None, values)) == [5]
```

Sorting με key

PYTHON

```
books = [("Ruby", 40), ("Python", 60), ("Go", 40)]
ranked = sorted(books, key=lambda book: (-book[1], book[0]))
assert ranked == [("Python", 60), ("Go", 40), ("Ruby", 40)]
```

Η function που δίνεις στο key υπολογίζει την τιμή με βάση την οποία θα γίνει το sort για κάθε στοιχείο. Αν δίνει tuple, συγκρίνεται πρώτα το πρώτο στοιχείο και μετά το δεύτερο αν χρειάζεται. Εδώ το αρνητικό πλήθος φέρνει τα μεγαλύτερα πρώτα και ο

τίτλος λύνει την ισοπαλία. Το sorting στην Python είναι stable: ίσα keys κρατούν την αρχική σχετική σειρά.

Grouping και Counter

```python
from collections import Counter, defaultdict

tags = ["python", "ruby", "python"]
counts = Counter(tags)
assert counts["python"] == 2
assert counts["go"] == 0

by_initial = defaultdict(list)
for tag in tags:
    by_initial[tag[0]].append(tag)
assert dict(by_initial) == {
    "p": ["python", "python"], "r": ["ruby"]
}
```

Το `defaultdict(list)` καλεί το `list` όταν ένα missing key ζητηθεί με `[]`. Δεν του περνάμε `list()`: θέλει callable που θα δημιουργεί νέα λίστα. Το `.get()` δεν ενεργοποιεί το default factory. Το `Counter` μετρά hashable τιμές και για missing key επιστρέφει μηδέν.

Το `itertools.groupby` δεν είναι το Ruby `group_by`: ομαδοποιεί συνεχόμενα στοιχεία με ίδιο key. Αν το ίδιο key εμφανιστεί ξανά αργότερα, δημιουργείται νέα ομάδα. Για πλήρη συγκέντρωση ανεξάρτητα από σειρά, χρησιμοποίησε dict ή defaultdict.

Ασκήσεις

Λύσεις κεφαλαίου 7 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

7.1 · Το λάθος compact

Μετέφερε το Ruby `[nil, 0, false, "", 3].compact` σε Python κρατώντας ακριβώς τη σημασία του. Εξήγησε γιατί το `filter(None, ...)` χάνει χρήσιμα δεδομένα.

Hint: έλεγξε ταυτότητα με `None`, όχι truthiness.

7.2 · Μικρό leaderboard

Δίνεται λίστα από `(name, score)`. Γράψε `leaderboard(rows)` που ταξινομεί κατά score φθίνουσα και μετά κατά name αύξουσα. Μην αλλάξεις την αρχική λίστα.

Hint: ένα tuple στο key αρκεί.

7.3 · Λέξεις ανά μήκος

Γράψε `group_by_length(words)` που για `["go", "ruby", "py", "java"]` δίνει `{2: ["go", "py"], 4: ["ruby", "java"]}`. Διατήρησε τη σειρά μέσα σε κάθε ομάδα.

Hint: μπορείς να χρησιμοποιήσεις `defaultdict(list)` και να επιστρέψεις απλό dict.

ΚΕΦΑΛΑΙΟ 8

Iterators και generators: ένα στοιχείο κάθε φορά

Μια λίστα περιέχει τα στοιχεία της. Ένας generator περιέχει μια διαδικασία που μπορεί να παράγει επόμενο στοιχείο. Αυτή η διαφορά αλλάζει τη μνήμη που χρειάζεσαι, τον χρόνο εκτέλεσης και το αν μπορείς να ξαναπεράσεις από την αρχή.

Iterable και iterator

Ένα iterable μπορεί να δώσει iterator μέσω `iter()`. Ένας iterator κρατά θέση και δίνει επόμενη τιμή μέσω `next()`. Στο τέλος σηκώνει `StopIteration`. Το `for` κάνει αυτή τη δουλειά για σένα και αντιμετωπίζει το `StopIteration` ως φυσικό τέλος.

PYTHON

```
values = [10, 20]
cursor = iter(values)
assert next(cursor) == 10
assert next(cursor) == 20
assert next(cursor, "end") == "end"
assert list(cursor) == []
assert list(values) == [10, 20]
```

Η λίστα παραμένει διαθέσιμη για νέα διαδρομή. Ο συγκεκριμένος iterator έχει εξαντληθεί. Το `iter(cursor)` επιστρέφει τον ίδιο iterator, δεν τον επαναφέρει στην αρχή. Πολλά bugs προκύπτουν όταν μια function καταναλώνει έναν iterator για logging και μετά τον παραδίδει άδειο στην πραγματική δουλειά.

Yield: διαφορετικό από το Ruby yield

PYTHON

```
def countdown(start):
    while start > 0:
        yield start
        start -= 1

stream = countdown(3)
assert next(stream) == 3
assert list(stream) == [2, 1]
assert list(stream) == []
```

Η παρουσία του `yield` κάνει τη function generator function. Η κλήση δεν εκτελεί αμέσως το σώμα: επιστρέφει generator object. Κάθε `next` συνεχίζει μέχρι το

επόμενο `yield`, που παραδίδει μια τιμή και παγώνει την κατάσταση της function. Το `return` χωρίς τιμή τερματίζει τον generator.

Στη Ruby το `yield` καλεί το block του caller. Εδώ δεν καλεί κάποιο block. Η πιο κοντινή νοητική σύνδεση είναι ένας custom Enumerator που παράγει στοιχεία καθώς του ζητούνται.

```python
events = []

def delayed():
    events.append("started")
    yield 42

g = delayed()
assert events == []
assert next(g) == 42
assert events == ["started"]
```

Αυτό εξηγεί γιατί ένα error μέσα στο σώμα generator μπορεί να εμφανιστεί όταν αρχίσεις να καταναλώνεις, αντί τη στιγμή που κάλεσες τη generator function.

Generator expressions και μικρά pipelines

```python
lines = [" 10 ", "", " 20 ", " 5 "]
cleaned = (line.strip() for line in lines)
numbers = (int(line) for line in cleaned if line)
assert sum(numbers) == 35
assert list(numbers) == []
```

Δεν αποθηκεύσαμε ενδιάμεσες λίστες. Κάθε αίτημα του `sum` τραβά την επόμενη τιμή από την αλυσίδα. Αυτό δεν σημαίνει ότι κάθε generator είναι πιο γρήγορος. Σημαίνει ότι μπορείς να επεξεργάζεσαι στοιχεία χωρίς να κρατάς όλα τα ενδιάμεσα αποτελέσματα στη μνήμη.

Μην επιστρέφεις generator που διαβάζει από αρχείο το οποίο έχει ήδη κλείσει. Το resource πρέπει να παραμένει ανοικτό όσο γίνεται η κατανάλωση. Θα το συνδέσουμε με το `with` στο επόμενο κεφάλαιο.

Yield from και itertools

PYTHON

```
def flatten(groups):
    for group in groups:
        yield from group

assert list(flatten([[1, 2], [], [3]])) == [1, 2, 3]

from itertools import islice, pairwise, batched

assert list(islice(countdown(10), 3)) == [10, 9, 8]
assert list(pairwise([2, 5, 9])) == [(2, 5), (5, 9)]
assert list(batched([1, 2, 3, 4, 5], 2)) == [
    (1, 2), (3, 4), (5,)
]
```

Το `yield from` δίνει τις τιμές ενός άλλου iterable. Το `islice` επιλέγει ένα κομμάτι από iterable χωρίς να απαιτεί indexing. Το `pairwise` δίνει γειτονικά ζεύγη. Το `batched`, διαθέσιμο από την 3.12, χωρίζει σε tuples μέχρι το ζητούμενο μέγεθος. Αν έχεις υπόλοιπο, η τελευταία είναι μικρότερη, εκτός αν ζητήσεις `strict=True` σε έκδοση που το υποστηρίζει, όπως η 3.14.

> **Πότε θέλεις list;** Όταν χρειάζεσαι indexing, μήκος, πολλά περάσματα από τα ίδια στοιχεία ή σταθερό snapshot. **Πότε θέλεις generator;** Όταν αρκεί ένα πέρασμα και θέλεις να επεξεργάζεσαι τα στοιχεία καθώς παράγονται.

Ασκήσεις

Λύσεις κεφαλαίου 8 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

8.1 · Η δεύτερη διαδρομή

Φτιάξε `numbers = (n * 2 for n in range(3))`. Τι δίνουν διαδοχικά τα `list(numbers)` και `list(numbers)`; Πώς θα άλλαζες τον κώδικα αν χρειαζόσουν δύο ίδιες διαδρομές;

Hint: αποθήκευσε το αποτέλεσμα ή δημιούργησε καινούργιο generator.

8.2 · Τρέχον άθροισμα

Γράψε generator `running_total(values)` που για `[3, 2, 5]` παράγει `3, 5, 10`. Για κενή είσοδο δεν παράγει τίποτα. Δοκίμασε και αρνητικές τιμές.

Hint: κράτα έναν accumulator και κάνε yield μετά την πρόσθεση.

8.3 · Τα πρώτα χρήσιμα στοιχεία

Γράψε `first_nonempty(lines, count)` που επιστρέφει λίστα με τα πρώτα `count` μη κενά strings αφού γίνει strip. Θεώρησε `count >= 0`. Για count 0 δεν πρέπει να καταναλώσει κανένα στοιχείο της εισόδου.

Hint: συνδύασε generator pipeline και `islice`.

ΚΕΦΑΛΑΙΟ 9

Exceptions και resource cleanup

Ένα exception μεταφέρει αποτυχία στον caller. Δεν είναι ειδική τιμή που χρειάζεται να ελέγχει κάθε ενδιάμεσο βήμα. Η βασική αντιστοίχιση από Ruby είναι `begin/rescue/ensure` προς `try/except/finally`, αλλά υπάρχουν μερικές χρήσιμες λεπτομέρειες.

Πιάσε την αποτυχία που περιμένεις

PYTHON

```
def parse_count(text):
    try:
        return int(text)
    except ValueError:
        return None

assert parse_count("12") == 12
assert parse_count("twelve") is None
```

Εδώ η `None` είναι συνειδητή απόφαση του API. Δεν λέμε ότι κάθε λάθος πρέπει να μετατρέπεται σε `None`. Αν ο caller πρέπει να ξεχωρίσει «δεν δόθηκε τιμή» από «δόθηκε κακή τιμή», ένα exception είναι συχνά πιο σαφές.

Το `except Exception` πιάνει πάρα πολλά και μπορεί να κρύψει δικό σου bug. Το σκέτο `except:` πιάνει ακόμη και καταστάσεις όπως `KeyboardInterrupt` και `SystemExit`. Διάλεξε συγκεκριμένους τύπους και κράτα το `try` όσο πιο μικρό γίνεται, ώστε να ξέρεις ποια πράξη απέτυχε.

Raise και η αιτία της αποτυχίας

PYTHON

```
def required_minutes(text):
    try:
        minutes = int(text)
    except ValueError as error:
        raise ValueError("minutes must be an integer") from error
    if minutes < 0:
        raise ValueError("minutes must be non-negative")
    return minutes

assert required_minutes("15") == 15
try:
    required_minutes("many")
except ValueError as error:
    assert isinstance(error.__cause__, ValueError)
else:
    raise AssertionError("expected ValueError")
```

Το `raise ... from error` κρατά ρητή σχέση με το αρχικό exception. Ένα σκέτο `raise` μέσα σε handler ξανασηκώνει το τρέχον exception. Αν η αποτυχία έχει ειδικό νόημα για το domain σου, μπορείς να ορίσεις δικό σου subtype του `Exception` ή ενός κατάλληλου subtype του, όπως `ValueError`.

Else και finally

PYTHON

```
events = []
try:
    number = int("42")
except ValueError:
    events.append("invalid")
else:
    events.append(f"parsed:{number}")
finally:
    events.append("finished")

assert events == ["parsed:42", "finished"]
```

Το `else` εκτελείται όταν το σώμα του `try` τελειώσει φυσιολογικά. Exceptions μέσα στο `else` δεν πιάνονται από τα προηγούμενα `except`. Το `finally` εκτελείται κατά την έξοδο, είτε υπήρξε exception είτε όχι, συμπεριλαμβανομένου ενός `return` μέσα στο `try`.

Μην βάζεις `return` στο `finally`: μπορεί να καλύψει το προηγούμενο αποτέλεσμα ή ακόμη και exception. Το `finally` είναι κατάλληλο για cleanup. Για resources όπως αρχεία, το `with` το κάνει πιο καθαρά.

With: το lifetime φαίνεται στον κώδικα

PYTHON

```
from io import StringIO

with StringIO("ruby\npython\n") as stream:
    languages = [line.strip() for line in stream]

assert languages == ["ruby", "python"]
assert stream.closed
```

Το `StringIO` είναι αρχείο κειμένου στη μνήμη. Το `with` χρησιμοποιεί το context manager protocol για την είσοδο στο block και την έξοδο από αυτό. Στη συνηθισμένη χρήση αρχείου, η έξοδος κλείνει το resource και σε περίπτωση exception. Η Ruby αναλογία είναι το `File.open(path) { |file| ... }`, χωρίς η Python να χρησιμοποιεί block argument.

Ένας context manager μπορεί σκόπιμα να καταπιεί exception. Δεν το κάνουν όλοι. Σε custom context manager, αν το `__exit__` επιστρέψει truthy τιμή, το exception δεν φτάνει στον caller. Αυτό λέγεται exception suppression. Αν δεν θέλεις να γράψεις class, το `contextlib.contextmanager` δίνει πιο σύντομη μορφή.

PYTHON

```
from contextlib import contextmanager

@contextmanager
def recorded(events):
    events.append("enter")
    try:
        yield
    finally:
        events.append("exit")

trace = []
with recorded(trace):
    trace.append("body")
assert trace == ["enter", "body", "exit"]
```

Το `@contextmanager` μετατρέπει αυτή τη generator function σε factory context managers. Το μοναδικό `yield` χωρίζει είσοδο και έξοδο. Αν προκύψει exception στο σώμα του `with`, εμφανίζεται στο σημείο του `yield`, γι' αυτό το cleanup μπαίνει στο `finally`. Στο κεφάλαιο 14 θα αποσυναρμολογήσουμε τη σύνταξη `@`.

Ασκήσεις

Λύσεις κεφαλαίου 9 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

9.1 · Η σειρά του cleanup

Χρησιμοποίησε το `recorded` και σήκωσε `ValueError` μέσα στο `with`, μετά το append `"body"`. Πιάσε το exception έξω από το `with` και πρόσθεσε `"caught"`. Ποια είναι η σειρά των τεσσάρων events;

Hint: η έξοδος από τον context manager προηγείται του εξωτερικού handler.

9.2 · Έγκυρο πλήθος

Γράψε `positive_count(text)` που επιστρέφει integer μεγαλύτερο από μηδέν. Κακό integer text, μηδέν και αρνητικοί αριθμοί πρέπει να σηκώνουν `ValueError`. Μην χρησιμοποιήσεις `assert` ως validation του input.

Hint: ξεχώρισε τη μετατροπή από τον κανόνα του domain.

9.3 · Κλείνει και στο error;

Άνοιξε ένα `StringIO` με `with`, σήκωσε `RuntimeError` μέσα και πιάσε το απ' έξω. Γράψε έλεγχο που αποδεικνύει ότι το stream έχει κλείσει.

Hint: το attribute `.closed` παραμένει διαθέσιμο μετά το κλείσιμο.

ΚΕΦΑΛΑΙΟ 10

Modules, packages και μικρά προγράμματα

Στη Ruby, το `require` φορτώνει κώδικα που συχνά ορίζει constants στο κοινό περιβάλλον. Στην Python, κάθε module έχει δικό του namespace. Το import σου δίνει ένα όνομα μέσα στο δικό σου namespace, όχι όλες τις functions του module αυτόματα.

Τρεις χρήσιμοι τρόποι import

PYTHON

```
import math
from pathlib import Path
from collections import Counter as Frequencies

assert math.sqrt(81) == 9
assert Path("notes.txt").suffix == ".txt"
assert Frequencies("abba")["a"] == 2
```

Το `import math` κρατά ορατή την προέλευση με `math.sqrt`. Το `from pathlib import Path` εισάγει συγκεκριμένο όνομα. Το `as` αλλάζει το τοπικό όνομα. Το `from module import *` κάνει δυσκολότερο να δεις από πού προέρχεται κάθε identifier, οπότε δεν το χρησιμοποιούμε στον κώδικα του βιβλίου.

Ένα συνηθισμένο eager import εκτελεί τον top-level κώδικα του module την πρώτη φορά που φορτώνεται. Στη συνέχεια το module επαναχρησιμοποιείται από το `sys.modules`. Αυτό δεν σημαίνει ότι επανεκτελείται κάθε φορά που γράφεις import. Επίσης δεν σημαίνει ότι διαφορετικά processes μοιράζονται τα ίδια module globals.

Το όνομα του αρχείου έχει σημασία

Μην ονομάσεις το πρώτο σου script `json.py`, `csv.py` ή `typing.py`. Το δικό σου αρχείο μπορεί να προηγηθεί της standard library στην αναζήτηση και να φορτωθεί εκείνο. Ένα μυστηριώδες «partially initialized module» συχνά προέρχεται από τέτοιο shadowing ή από circular import.

Για να ελέγξεις ποιο module φορτώθηκε, δες το `module.__file__` όταν υπάρχει. Για τον interpreter, δες `sys.executable`. Για το περιβάλλον, δες `python -m pip --version`. Πριν αλλάξεις dependencies, έλεγξε ότι παρατηρείς το σωστό περιβάλλον.

Ένα package που τρέχει

Φτιάχνουμε τον παρακάτω φάκελο. Το `__init__.py` δηλώνει ένα συνηθισμένο package. Υπάρχουν και namespace packages χωρίς αυτό, αλλά δεν τα χρειαζόμαστε εδώ.

OUTPUT

```
project/
  tinybook/
    __init__.py
    core.py
    __main__.py
```

Το `tinybook/__init__.py` μπορεί να περιέχει μόνο ένα docstring:

tinybook/__init__.py

```
"""A tiny package for the book."""
```

Ο χρήσιμος υπολογισμός βρίσκεται στο `core.py`:

tinybook/core.py

```
def reading_hours(minutes):
    return minutes / 60
```

Το entry point συνδέει τον υπολογισμό με το terminal:

tinybook/__main__.py

```
from .core import reading_hours

def main():
    print(f"{reading_hours(90):.1f} hours")

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

Από τον φάκελο `project`, εκτελείς:

TERMINAL

```
python -m tinybook
```

Το αποτέλεσμα είναι `1.5 hours`. Το `-m` ζητά από την Python να βρει το package όπως σε import και να εκτελέσει το `__main__.py`. Το relative import `.core` αποκτά έτσι γνωστό package context. Μην τρέξεις το `tinybook/__main__.py` απευθείας ως ξεκομμένο αρχείο και περιμένεις να λειτουργήσει το ίδιο relative import.

Το main guard

Όταν ένα module εκτελείται ως το κύριο πρόγραμμα, το `__name__` είναι `"__main__"`. Όταν εισάγεται κανονικά, είναι το όνομα του module. Ο έλεγχος στο τέλος κρατά το CLI έξω από το import path των tests. Το να εισάγεις ένα module δεν πρέπει να ξεκινά κατά λάθος τη δική σου διαδραστική συνεδρία.

Ένα module μπορεί να περιέχει functions χωρίς class. Δεν χρειάζεσαι ένα `Utils` class μόνο για να ομαδοποιήσεις functions: το ίδιο το module παρέχει namespace. Μοίραζε αρχεία όταν προκύπτει ξεκάθαρη ευθύνη, όχι για να αποκτήσει το project περισσότερους φακέλους.

Πού μπαίνει το pyproject.toml;

Το `pyproject.toml` συγκεντρώνει metadata του project και configuration πολλών εργαλείων. Δεν είναι ακριβές ισοδύναμο ενός `Gemfile` ούτε εξασφαλίζει μόνο του ότι δύο εγκαταστάσεις θα έχουν τις ίδιες εκδόσεις packages. Τα dependencies, το build backend και το lockfile λύνουν διαφορετικά προβλήματα.

TOML

```toml
[project]
name = "tinybook-example"
version = "0.1.0"
requires-python = ">=3.14"
dependencies = []
```

Αυτό δείχνει project metadata. Δεν το παρουσιάζουμε ως πλήρη οδηγό packaging για δημοσίευση. Το τοπικό package παραπάνω εκτελείται από τον γονικό φάκελο χωρίς εγκατάσταση. Σε μεγαλύτερο project μπορείς να χρησιμοποιήσεις ένα εργαλείο όπως το uv για environments και lockfiles, αλλά η βασική διάκριση interpreter, module και package παραμένει η ίδια.

Ασκήσεις

Λύσεις κεφαλαίου 10 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

10.1 · Import χωρίς εκπλήξεις

Ποια γραμμή του `tinybook/__main__.py` εμποδίζει το `main()` να τρέξει κατά το import; Γράψε ένα μικρό import test που ελέγχει τη `reading_hours(30)` χωρίς να καλεί το CLI.

Hint: κάνε import τη function από το `tinybook.core`.

10.2 · Ξεχώρισε τη μετατροπή

Πρόσθεσε μια function `minutes_from_hours(hours)` που πολλαπλασιάζει επί 60. Βάλ' τη σε module χωρίς εκτέλεση στο top level. Δοκίμασε 1.5 και 0. Θεώρησε ότι το input είναι αριθμητικό.

Hint: ένα αρχείο με έναν ορισμό function είναι ήδη χρήσιμο module.

10.3 · Το λάθος όνομα

Ένα project έχει αρχείο `csv.py` και μέσα γράφει `import csv`. Μετά αποτυγχάνει το `csv.DictReader(...)`. Εξήγησε την πιθανή αιτία και δώσε συγκεκριμένα βήματα διάγνωσης και διόρθωσης.

Hint: ξεκίνα από το `csv.__file__` και το όνομα του δικού σου αρχείου.

ΚΕΦΑΛΑΙΟ 11

Classes, dataclasses και protocols

Δεν χρειάζεται να γράφεις class για κάθε δουλειά. Όταν όμως θέλεις να κρατήσεις state μαζί με τις methods που το χειρίζονται ή να δώσεις όνομα σε ένα record, η Python σου δίνει αρκετές επιλογές.

Το self είναι ρητό

PYTHON

```
class ReadingSession:
    def __init__(self, title, minutes=0):
        self.title = title
        self.minutes = minutes

    def add_minutes(self, amount):
        self.minutes += amount

    def summary(self):
        return f"{self.title}: {self.minutes} min"

session = ReadingSession("Python", 15)
session.add_minutes(10)
assert session.summary() == "Python: 25 min"
```

Το `__init__` αρχικοποιεί ένα ήδη δημιουργημένο instance. Η κατασκευή ξεκινά πριν από αυτό, μέσω `__new__`, αλλά σπάνια χρειάζεται να το αγγίξεις σε καθημερινό κώδικα. Το `__init__` δεν επιστρέφει το instance και πρέπει να επιστρέφει `None`.

Το `self` είναι συμβατικό όνομα του πρώτου parameter μιας instance method. Στο `session.summary()`, η Python περνά το instance αυτόματα. Μέσα στη method γράφεις `self.title`. Δεν υπάρχει σιωπηρή αναζήτηση ενός `@title` ή άλλου instance variable.

Instance state και class state

PYTHON

```
class Shelf:
    category = "books"

    def __init__(self):
        self.titles = []

first = Shelf()
second = Shelf()
first.titles.append("Python")
assert second.titles == []
assert first.category == "books"
```

Το `category` είναι class attribute. Το `titles` δημιουργείται για κάθε instance. Αν έβαζες `titles = []` στο σώμα της class, όλα τα instances που δεν έχουν δικό τους attribute με το ίδιο όνομα θα έβλεπαν την ίδια λίστα. Είναι το ίδιο πρόβλημα με το shared mutable state που είδαμε στα defaults.

Το `_internal` δηλώνει «εσωτερικό» κατά σύμβαση, δεν απαγορεύει πρόσβαση. Το `__internal` ενεργοποιεί name mangling, κυρίως για αποφυγή συγκρούσεων σε subclasses. Δεν αποτελεί όριο ασφάλειας. Οι double-underscore methods όπως `__len__` συμμετέχουν στα protocols της γλώσσας.

Properties και εναλλακτικοί constructors

PYTHON

```
class Duration:
    def __init__(self, minutes):
        self.minutes = minutes

    @property
    def hours(self):
        return self.minutes / 60

    @classmethod
    def from_hours(cls, hours):
        return cls(hours * 60)

duration = Duration.from_hours(1.5)
assert duration.minutes == 90
assert duration.hours == 1.5
```

Η property χρησιμοποιείται σαν attribute, αλλά μπορεί να υπολογίζεται. Η classmethod παίρνει την class ως `cls`, οπότε ο constructor μπορεί να λειτουργεί και με κατάλληλα subclasses. Η `staticmethod` δεν λαμβάνει ούτε instance ούτε class. Συχνά μια απλή module function είναι πιο φυσική επιλογή για τέτοια δουλειά.

Για inheritance γράφεις `class Child(Parent):`. Το `super()` ακολουθεί το method resolution order, ή MRO: τη σειρά με την οποία η Python ψάχνει methods στις classes. Δεν σημαίνει απλώς «κάλεσε την άμεση parent class». Για μικρά προγράμματα, κράτα το inheritance απλό ή χρησιμοποίησε composition, με κάθε object να έχει σαφή δουλειά.

Dataclass: records με λιγότερο boilerplate

PYTHON

```
from dataclasses import dataclass, field, replace

@dataclass
class Book:
    title: str
    pages: int
    tags: list[str] = field(default_factory=list)

book = Book("Python", 180)
other = Book("Ruby", 140)
book.tags.append("language")
assert other.tags == []
assert Book("Go", 90) == Book("Go", 90)
revised = replace(book, pages=200)
assert revised.pages == 200
assert book.pages == 180
assert revised.tags is book.tags
```

Το decorator παράγει μεταξύ άλλων `__init__`, `__repr__` και σύγκριση ισότητας με βάση τα fields. Τα annotations δηλώνουν τα fields, αλλά δεν κάνουν runtime type checking. Το `default_factory=list` δημιουργεί νέα λίστα ανά instance. Το `replace` δημιουργεί νέο instance, αλλά δεν κάνει deep copy στα fields.

Το `@dataclass(frozen=True)` εμποδίζει το συνηθισμένο assignment στα fields. Δεν κάνει immutable τις τιμές που περιέχουν. Το `slots=True` αλλάζει την αποθήκευση των attributes και συνήθως δεν επιτρέπει αυθαίρετα νέα instance attributes. Χρησιμοποίησε αυτές τις επιλογές όταν χρειάζεσαι τη συγκεκριμένη συμπεριφορά.

Πώς η σύνταξη χρησιμοποιεί τα protocols

PYTHON

```
class ReadingQueue:
    def __init__(self, titles):
        self._titles = list(titles)

    def __len__(self):
        return len(self._titles)

    def __iter__(self):
        return iter(self._titles)

    def __repr__(self):
        return f"ReadingQueue({self._titles!r})"

queue = ReadingQueue(["Ruby", "Python"])
assert len(queue) == 2
assert list(queue) == ["Ruby", "Python"]
assert bool(ReadingQueue([])) is False
```

Χωρίς `__bool__`, το `__len__` καθορίζει και το truthiness. Ένα κενό `ReadingQueue` είναι falsey.

Ασκήσεις

Λύσεις κεφαλαίου 11 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

11.1 · Η κοινή λίστα

Μια class έχει `items = []` στο σώμα της και η method add κάνει `self.items.append(item)`. Γιατί δύο instances βλέπουν μαζί τις προσθήκες; Διόρθωσε την class ώστε η κάθε λίστα να είναι ανεξάρτητη.

Hint: η δημιουργία της λίστας πρέπει να συμβαίνει στην αρχικοποίηση του instance.

11.2 · Μια dataclass για σημειώσεις

Φτιάξε `Note` με `title: str` και `tags: list[str]`. Τα tags να ξεκινούν ως διαφορετική κενή λίστα για κάθε Note. Πρόσθεσε property `tag_count`.

Hint: συνδύασε `field(default_factory=list)` και `@property`.

11.3 · Ένα μικρό iterable

Φτιάξε `Playlist` που αντιγράφει τα titles της εισόδου και υποστηρίζει `len()` και iteration. Αν αλλάξει η αρχική εξωτερική λίστα μετά τη δημιουργία, η Playlist να παραμένει ίδια.

Hint: η αντιγραφή στο `__init__` είναι μέρος του contract της άσκησης.

ΚΕΦΑΛΑΙΟ 12

Type hints που βοηθούν χωρίς να λένε ψέματα

Τα type hints περιγράφουν τι περιμένει ο κώδικας. Ο interpreter δεν τα μετατρέπει αυτόματα σε validation. Ένας editor ή static type checker μπορεί να τα χρησιμοποιήσει για να εντοπίσει λάθη πριν τρέξεις το πρόγραμμα. Αυτά είναι δύο διαφορετικά επίπεδα.

Η σύγχρονη καθημερινή μορφή

PYTHON

```
def average(values: list[float]) -> float | None:
    if not values:
        return None
    return sum(values) / len(values)

assert average([2.0, 4.0]) == 3.0
assert average([]) is None
```

Γράφουμε `list[str]`, `dict[str, int]`, `tuple[int, str]` και `set[str]`. Το `str | None` δηλώνει ότι μπορεί να επιστραφεί και απουσία τιμής. Το `tuple[int, ...]` δηλώνει μεταβλητού μήκους tuple από integers. Το `-> None` δηλώνει ότι η function δεν δίνει χρήσιμο αποτέλεσμα.

Το παλαιότερο `typing.List` εμφανίζεται σε υπάρχοντα projects, αλλά δεν είναι η προτεινόμενη μορφή για νέο κώδικα που στοχεύει τις εκδόσεις του βιβλίου. Το `Optional[str]` σημαίνει `str | None`, όχι «argument που μπορείς να παραλείψεις». Το αν παραλείπεται ένα argument καθορίζεται από το default στη signature.

Δέξου ό,τι χρειάζεσαι, όχι έναν τυχαίο συγκεκριμένο τύπο

PYTHON

```
from collections.abc import Iterable, Sequence, Callable

def total(values: Iterable[int]) -> int:
    return sum(values)

def first(values: Sequence[str]) -> str | None:
    return values[0] if values else None

def transform(value: int, fn: Callable[[int], int]) -> int:
    return fn(value)

assert total(n for n in range(4)) == 6
assert first(("Ruby", "Python")) == "Ruby"
assert transform(3, lambda n: n + 1) == 4
```

Το `total` χρειάζεται iteration, άρα δεν απαιτεί list. Το `first` χρειάζεται indexing, άρα μια Sequence περιγράφει καλύτερα τη δουλειά. Η υπερβολικά συγκεκριμένη annotation μπορεί να περιορίζει άδικα τους callers στα μάτια του type checker. Η υπερβολικά γενική μπορεί να σου επιτρέπει πράξεις που δεν δικαιολογούνται από το contract.

Το `Any` ουσιαστικά χαλαρώνει τον στατικό έλεγχο στις αντίστοιχες πράξεις. Το `object` λέει «κάποιο Python object» και σε αναγκάζει να ελέγξεις ή να περιορίσεις τον τύπο πριν χρησιμοποιήσεις ειδικές methods. Δεν είναι ταυτόσημα.

Narrowing και εξωτερικά δεδομένα

PYTHON

```
def title_from(value: object) -> str:
    if not isinstance(value, str):
        raise TypeError("title must be text")
    return value.strip()

assert title_from(" Python ") == "Python"
```

Ο πραγματικός έλεγχος είναι το `isinstance`, όχι η annotation. Μετά τον έλεγχο, ένας type checker συνήθως γνωρίζει ότι το value είναι str. Η `typing.cast()` δεν κάνει αντίστοιχο runtime έλεγχο. Είναι δήλωση προς τον checker, όχι απόδειξη ότι ένα JSON payload έχει τη δομή που λες.

TypedDict και Protocol

```python
from typing import TypedDict, NotRequired, Protocol

class BookRow(TypedDict):
    title: str
    pages: int
    note: NotRequired[str]

class HasTitle(Protocol):
    @property
    def title(self) -> str: ...

def label(item: HasTitle) -> str:
    return item.title.upper()

from dataclasses import dataclass

@dataclass
class Article:
    title: str

row: BookRow = {"title": "Python", "pages": 100}
assert isinstance(row, dict)
assert label(Article("Python")) == "PYTHON"
```

Το TypedDict περιγράφει τα keys ενός dict για τον static checker. Στο runtime παραμένει συνηθισμένο dict και δεν γίνεται validation. Το `NotRequired` σημαίνει ότι το key μπορεί να απουσιάζει, όχι ότι η τιμή του μπορεί να είναι `None`.

Το Protocol περιγράφει συμπεριφορά ή attributes που χρειάζεται ο caller. Η `Article` δεν είναι subclass του `HasTitle`, αλλά έχει συμβατό public interface. Αυτό είναι structural typing: ο checker κοιτάζει τι υποστηρίζει η class. Ταιριάζει στο duck typing που ήδη γνωρίζεις από Ruby.

Type parameters και aliases από την Python 3.12

```python
type Scores = dict[str, int]

def head[T](items: Sequence[T]) -> T | None:
    return items[0] if items else None

assert head([10, 20]) == 10
assert head(["ruby"]) == "ruby"
```

Το T συνδέει τον τύπο του input με τον τύπο του output. Δεν είναι απλώς «οτιδήποτε». Αν περάσεις Sequence από strings, ο checker μπορεί να συμπεράνει `str | None`. Η σύνταξη `type` φτιάχνει type alias, όχι νέο runtime τύπο κατάλληλο για κάθε χρήση του `isinstance`.

Για τις αλλαγές αξιολόγησης annotations στην 3.14, δες το κεφάλαιο 18.

Ασκήσεις

Λύσεις κεφαλαίου 12 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

12.1 · Annotation ή validation;

Η `def identity(value: int) -> int: return value` καλείται με `"hello"`. Θα το σταματήσει μόνη της η Python; Γράψε την πρόβλεψη και μια δεύτερη function που ελέγχει πραγματικά ότι δόθηκε ακριβώς int, αποκλείοντας bool.

Hint: για την απαίτηση της άσκησης, το `type(value) is int` είναι πιο ακριβές από `isinstance(value, int)`.

12.2 · Generic τελευταίο στοιχείο

Γράψε `last[T]` που δέχεται `Sequence[T]` και επιστρέφει `T | None`. Δοκίμασε λίστα integers, tuple strings και κενή sequence.

Hint: η generic σύνταξη μπαίνει μετά το όνομα της function.

12.3 · Contract χωρίς inheritance

Όρισε Protocol `Named` με read-only property `name: str` και function `greet(person: Named) -> str`. Φτιάξε dataclass `Person` που ικανοποιεί το interface χωρίς να είναι subclass του Protocol.

Hint: ένα κανονικό instance attribute μπορεί να καλύπτει το read-only property που ζητά ένα Protocol.

ΚΕΦΑΛΑΙΟ 13

Pattern matching: διάβασε τη δομή των δεδομένων

Το `match/case`, διαθέσιμο από την Python 3.10, δεν είναι απλώς switch με άλλο όνομα. Μπορεί να ελέγξει τη δομή μιας τιμής και να κρατήσει τα κομμάτια που σε ενδιαφέρουν σε νέα ονόματα.

Sequences και captures

PYTHON

```
def command(parts):
    match parts:
        case ["help"]:
            return "show help"
        case ["open", title]:
            return f"opening {title}"
        case ["tag", title, *tags] if tags:
            return f"{title}: {len(tags)} tags"
        case _:
            return "unknown command"

assert command(["open", "Python"]) == "opening Python"
assert command(["tag", "Python", "code"]) == "Python: 1 tags"
assert command(["tag", "Python"]) == "unknown command"
```

Τα strings μέσα σε εισαγωγικά είναι literal patterns. Τα σκέτα `title` και `tags` είναι captures: δένουν ονόματα σε τιμές. Το _ είναι wildcard που δεν αποθηκεύει τιμή. Τα cases εξετάζονται με τη σειρά και εκτελείται το πρώτο που ταιριάζει και περνά το guard.

Η sequence μορφή μπορεί να ταιριάξει list και tuple. Δεν σημαίνει ότι ταιριάζει οποιοδήποτε iterable: generators δεν αντιμετωπίζονται ως sequences εδώ. Strings, bytes και bytearray επίσης εξαιρούνται από sequence patterns. Το `case [a, b]` δεν είναι τρόπος να σπάσεις ένα string δύο χαρακτήρων.

Dict patterns αφήνουν χώρο για επιπλέον keys

PYTHON

```
def event_label(event):
    match event:
        case {"kind": "read", "minutes": int(n)} if n > 0:
            return f"read:{n}"
        case {"kind": "pause"}:
            return "paused"
        case _:
            return "unsupported"

assert event_label({"kind": "read", "minutes": 15}) == "read:15"
assert event_label({"kind": "pause", "reason": "tea"}) == "paused"
```

Ένα mapping pattern απαιτεί τα συγκεκριμένα keys αλλά επιτρέπει και επιπλέον keys. Για να τα κρατήσεις, γράψε `**rest`. Το `int(n)` είναι class pattern: ταιριάζει σε int και συνδέει την τιμή στο `n`. Δεν καλεί μετατροπή από string.

Πρόσεξε ότι το bool είναι subtype του int. Αν δεν θέλεις `True` ως διάρκεια, βάλε guard `type(n) is int` μαζί με τον έλεγχο του ορίου. Το pattern matching δεν αντικαθιστά αυτόματα όλους τους κανόνες validation του domain σου.

Η μεγάλη παγίδα: σκέτο όνομα σημαίνει capture

Αν έχεις μεταβλητή `expected = "ready"`, το `case expected:` δεν συγκρίνει με την τιμή της. Αποθηκεύει εκεί ό,τι ταιριάζει. Αυτό μπορεί να καλύψει όλα τα επόμενα cases και ο compiler να απορρίψει τον κώδικα ως unreachable.

Για σύγκριση με μεταβλητή χρησιμοποίησε guard. Για ονοματισμένες constants, ένα dotted name, όπως `Status.READY`, είναι value pattern. Τα `None`, `True` και `False` έχουν δικούς τους singleton patterns.

PYTHON

```
def matches_expected(value, expected):
    match value:
        case candidate if candidate == expected:
            return True
        case _:
            return False

assert matches_expected("ready", "ready")
assert not matches_expected("waiting", "ready")
```

Alternatives, as και class patterns

PYTHON

```
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int

def locate(point):
    match point:
        case Point(x=0, y=0):
            return "origin"
        case Point(x=x, y=0) | Point(x=0, y=x):
            return f"axis:{x}"
        case Point() as other:
            return f"plane:{other.x},{other.y}"
        case _:
            return "not a point"

assert locate(Point(0, 0)) == "origin"
assert locate(Point(0, 5)) == "axis:5"
assert locate(Point(2, 3)) == "plane:2,3"
```

Το | ενώνει εναλλακτικά patterns. Πρέπει να κάνουν bind τα ίδια ονόματα, ώστε το σώμα να γνωρίζει τι είναι διαθέσιμο. Το as κρατά ολόκληρο το object. Τα keyword class patterns κάνουν εμφανές ποια attributes ελέγχεις. Οι dataclasses υποστηρίζουν συνήθως και positional patterns, αλλά η keyword μορφή είναι πιο ανθεκτική σε αλλαγές της σειράς των fields.

Το `match` δεν επιστρέφει τιμή ως expression. Μην βασίζεσαι σε captures που προήλθαν από αποτυχημένο pattern για κώδικα μετά το match. Κράτα τη χρήση τους μέσα στο case που τα ορίζει με επιτυχία.

Ασκήσεις

Λύσεις κεφαλαίου 13 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

13.1 · Capture ή σύγκριση;

Θέλεις να ελέγξεις αν το input είναι ίσο με μια μεταβλητή `wanted`. Γιατί δεν γράφεις `case wanted:`; Γράψε λειτουργική εκδοχή με guard.

Hint: ένα νέο capture και μετά σύγκριση κρατούν τις δύο δουλειές χωριστές.

13.2 · Μικρός parser εντολών

Γράψε `parse_command(parts)` που επιστρέφει `("list", None)` για `["list"]`, `("show", name)` για `["show", name]` και `None` αλλιώς. Το `["show"]` δεν είναι έγκυρο.

Hint: κάθε case περιγράφει το ακριβές πλήθος που χρειάζεσαι.

13.3 · Διάρκεια με σωστό τύπο

Γράψε `read_minutes(event)` που για mapping με `kind="read"` και μη αρνητικό ακριβώς int `minutes` επιστρέφει τη διάρκεια. Για οτιδήποτε άλλο επιστρέφει `None`. Το 0 είναι έγκυρο, το `True` όχι. Επιπλέον keys επιτρέπονται.

Hint: mapping pattern μαζί με guard για τύπο και όριο.

ΚΕΦΑΛΑΙΟ 14

Closures και decorators χωρίς μυστήριο

Ένα decorator δεν είναι διαφορετικό είδος function. Είναι κάτι που δέχεται το object που ορίστηκε και επιστρέφει αυτό που θα συνδεθεί τελικά με το όνομά του. Για τα συνηθισμένα function decorators, η είσοδος και η έξοδος είναι functions.

Αφαίρεσε το @ για να δεις τι συμβαίνει

PYTHON

```
from functools import wraps

def uppercase_result(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        return result.upper()
    return wrapper

@uppercase_result
def greeting(name):
    """Return a short greeting."""
    return f"Hello, {name}"

assert greeting("Ada") == "HELLO, ADA"
assert greeting.__name__ == "greeting"
```

Το `@uppercase_result` αντιστοιχεί ουσιαστικά σε `greeting = uppercase_result(greeting)` αμέσως μετά τον ορισμό. Το decorator τρέχει τότε. Το σώμα του wrapper τρέχει αργότερα, κάθε φορά που καλείται η τελική `greeting`.

Το `*args, **kwargs` προωθεί arguments. Το `wraps` διατηρεί χρήσιμα metadata όπως όνομα και docstring και προσθέτει `__wrapped__`, ώστε εργαλεία introspection να βρίσκουν την αρχική function. Δεν εγγυάται ότι ο wrapper διατηρεί κάθε χαρακτηριστικό του αρχικού callable.

Decorator με δικά του arguments

PYTHON

```
def prefixed(prefix):
    def decorate(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            return prefix + function(*args, **kwargs)
        return wrapper
    return decorate

@prefixed("[book] ")
def title():
    return "Python"

assert title() == "[book] Python"
```

Η εξωτερική function παίρνει το configuration. Η μεσαία είναι ο decorator και παίρνει την αρχική function. Η εσωτερική είναι ο wrapper που διαχειρίζεται κάθε κλήση. Δεν χρειάζεται να αποστηθίσεις τρία nested `def`: σκέψου ποια τιμή γνωρίζεις σε κάθε χρονική στιγμή.

Σε stacked decorators, ο πιο κοντινός στο `def` εφαρμόζεται πρώτος. Τα `@outer` πάνω από `@inner` δίνουν `outer(inner(function))`. Η σειρά έχει σημασία αν ο ένας αλλάζει το αποτέλεσμα που βλέπει ο άλλος.

Το closure κρατά το binding

PYTHON

```
functions = [lambda: number for number in range(3)]
assert [function() for function in functions] == [2, 2, 2]

fixed = [lambda number=number: number for number in range(3)]
assert [function() for function in fixed] == [0, 1, 2]
```

Οι πρώτες functions διαβάζουν το ίδιο binding όταν καλούνται. Όταν ολοκληρωθεί το comprehension, η τελευταία τιμή είναι 2. Στη δεύτερη εκδοχή, το default argument αποθηκεύει την τρέχουσα τιμή κατά τη δημιουργία κάθε function. Εδώ αξιοποιούμε σκόπιμα τη στιγμή αξιολόγησης των defaults που πριν προκάλεσε το bug με τη λίστα.

Μια factory function, όπως το `make_multiplier` του κεφαλαίου 6, είναι συχνά πιο καθαρή λύση από ένα πολύπλοκο lambda. Ένα comprehension έχει δικό του scope για το loop variable, αλλά οι closures *μέσα στο ίδιο comprehension* εξακολουθούν να μπορούν να μοιράζονται το binding του.

Partial και cache ως έτοιμα εργαλεία

PYTHON

```
from functools import partial, lru_cache

def add(a, b):
    return a + b

add_ten = partial(add, 10)
assert add_ten(5) == 15

@lru_cache(maxsize=128)
def normalize_tag(tag):
    return tag.strip().casefold()

assert normalize_tag(" Python ") == "python"
assert normalize_tag(" Python ") == "python"
assert normalize_tag.cache_info().hits == 1
```

Το `partial` αποθηκεύει κάποια arguments και δίνει νέο callable για τα υπόλοιπα. Το `lru_cache` αποθηκεύει αποτελέσματα ανά key κλήσης και χρειάζεται hashable arguments. Δεν είναι σωστό για κάθε function: αποτελέσματα που εξαρτώνται από τον χρόνο, από εξωτερικό mutable state ή από επιθυμητά side effects μπορεί να μη γίνεται να επαναχρησιμοποιηθούν.

Η cache κρατά references στα arguments και στα αποτελέσματα μέχρι να απομακρυνθούν. Αν το αποτέλεσμα είναι mutable και ο caller το αλλάξει, μελλοντική κλήση μπορεί να δει την αλλαγή. Ξεκίνα με απλές, προβλέψιμες functions και πρόσθεσε cache όταν υπάρχει συγκεκριμένη ανάγκη.

> **Πριν γράψεις decorator:** δες αν μια κανονική function call λέει καθαρότερα τι γίνεται. Τα decorators είναι χρήσιμα για επαναλαμβανόμενη συμπεριφορά γύρω από πολλές functions, όχι για να κρυφτεί η κύρια ροή του προγράμματος.

Ασκήσεις

Λύσεις κεφαλαίου 14 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

14.1 · Τρεις ίδιες απαντήσεις

Δημιούργησε τρεις functions σε loop που να επιστρέφουν 10, 20 και 30 αντίστοιχα. Δείξε πρώτα το late-binding bug και μετά διόρθωσέ το με factory ή default argument.

Hint: το πότε διαβάζεται το binding είναι το θέμα.

14.2 · Καθάρισε το αποτέλεσμα

Γράψε decorator `stripped` για functions που επιστρέφουν str. Να κάνει strip στο αποτέλεσμα, να προωθεί positional και keyword arguments και να διατηρεί το `__name__` της αρχικής function.

Hint: άλλαξε μόνο τη γραμμή επεξεργασίας του αποτελέσματος στο πρώτο παράδειγμα.

14.3 · Prefix με configuration

Χρησιμοποίησε decorator factory για να προσθέτεις `"INFO: "` μπροστά από αποτέλεσμα function. Έλεγξε ότι το prefix μπαίνει μία φορά ανά κλήση και ότι μπορείς να εφαρμόσεις τον decorator σε δύο functions με διαφορετικό prefix για καθεμία.

Hint: κάθε κλήση της decorator factory δημιουργεί closure με το δικό του configuration.

ΚΕΦΑΛΑΙΟ 15

Αρχεία και δεδομένα με τη standard library

Ένα χρήσιμο script πρέπει συνήθως να διαβάσει κάτι έξω από τον κώδικά του. Το πρώτο βήμα είναι να ξεχωρίσεις τρία επίπεδα: bytes ή κείμενο, μορφή δεδομένων και κανόνες του δικού σου προγράμματος.

Pathlib: paths ως objects

PYTHON

```
from pathlib import Path

path = Path("notes") / "python.txt"
assert path.name == "python.txt"
assert path.stem == "python"
assert path.suffix == ".txt"
assert path.with_suffix(".md").name == "python.md"
```

Το `/` ενώνει path components εδώ, δεν κάνει αριθμητική. Το `Path` δεν ανοίγει αυτόματα το αρχείο. Ένα relative path αναφέρεται στο current working directory, όχι απαραίτητα στον φάκελο του source file. Για δεδομένα που πρέπει να βρίσκονται δίπλα στο module, μπορεί να χρειαστεί βάση από `Path(__file__).resolve().parent`.

Για μικρά αρχεία υπάρχουν `read_text` και `write_text`. Για σταδιακή επεξεργασία, άνοιξε stream με `with`. Το επόμενο παράδειγμα χρησιμοποιεί δικό του προσωρινό φάκελο, ώστε να μπορείς να το επαναλάβεις χωρίς να αλλάξεις δικά σου αρχεία.

PYTHON

```
from tempfile import TemporaryDirectory

with TemporaryDirectory() as folder:
    target = Path(folder) / "notes.txt"
    target.write_text("Ruby\nPython\n", encoding="utf-8")
    with target.open(encoding="utf-8") as stream:
        titles = [line.strip() for line in stream]
    assert titles == ["Ruby", "Python"]
```

Το explicit encoding δηλώνει πώς γράφτηκε το κείμενο και κάνει τον κώδικα σαφή και σε παλιότερες εκδόσεις. Η 3.15 χρησιμοποιεί UTF-8 mode από προεπιλογή, αλλά τα legacy αρχεία δεν μετατρέπονται μαγικά σε UTF-8.

Το `write_text` αντικαθιστά υπάρχον περιεχόμενο. Αν θες προσθήκη, άνοιξε με mode `"a"`. Αν θες δημιουργία που αποτυγχάνει όταν υπάρχει ήδη αρχείο, χρησιμοποίησε `"x"`. Για την εφαρμογή σου διάλεξε σκόπιμα ποια από αυτές τις συμπεριφορές υπόσχεσαι.

JSON: parser, όχι schema validator

PYTHON

```
import json

payload = '{"title": "Python", "minutes": 25, "done": false}'
record = json.loads(payload)
assert record["minutes"] == 25
assert record["done"] is False
encoded = json.dumps(record, ensure_ascii=False, indent=2)
assert json.loads(encoded) == record
```

Τα `loads` και `dumps` δουλεύουν με strings. Τα `load` και `dump` με file-like objects. Το JSON `null` γίνεται `None`, τα booleans γίνονται `True` και `False`. Τα object keys είναι strings. Μια tuple γράφεται ως JSON array και διαβάζεται πίσω ως list, όχι ως tuple.

Το επιτυχές parsing δεν λέει ότι υπάρχουν τα σωστά keys ή ότι το `minutes` είναι έγκυρη διάρκεια. Κάνε αυτούς τους ελέγχους ξεχωριστά. Μην χρησιμοποιείς `eval` για να διαβάζεις δεδομένα. Δεν χρειάζεται εκτέλεση Python για να διαβαστεί JSON.

CSV: άφησε τον parser να χειριστεί τα κόμματα

PYTHON

```
import csv
from io import StringIO

source = StringIO('title,minutes\n"Python, daily",25\nRuby,15\n')
rows = list(csv.DictReader(source))
assert rows[0]["title"] == "Python, daily"
assert rows[0]["minutes"] == "25"
assert sum(int(row["minutes"]) for row in rows) == 40
```

Το `.split(",")` δεν είναι CSV parser: αγνοεί quoting, escaped quotes και newlines μέσα σε πεδία. Το `DictReader` δίνει ένα dict ανά γραμμή με βάση τα headers. Οι τιμές του παραδείγματος παραμένουν strings και χρειάζονται μετατροπή.

Στα πραγματικά CSV αρχεία, άνοιξε με `newline=""`, όπως `path.open(encoding="utf-8", newline="")`, και άφησε το module `csv` να χειριστεί τα line endings. Εξέτασε headers, missing fields και γραμμές που έχουν περισσότερα πεδία από όσα περιμένεις. Ο parser και οι κανόνες του δικού σου format δεν είναι το ίδιο πράγμα.

Ημερομηνίες και ακριβείς δεκαδικοί

PYTHON

```
from datetime import date, datetime, UTC, timedelta
from decimal import Decimal

day = date.fromisoformat("2026-10-16")
assert day + timedelta(days=2) == date(2026, 10, 18)
moment = datetime(2026, 10, 16, 9, 0, tzinfo=UTC)
assert moment.isoformat() == "2026-10-16T09:00:00+00:00"
assert Decimal("0.1") + Decimal("0.2") == Decimal("0.3")
```

Το `date` είναι ημερολογιακή ημέρα. Το `datetime` μπορεί να έχει ή να μην έχει πληροφορία timezone. Μην αναμειγνύεις naive και aware datetimes σε συγκρίσεις. Για πραγματικό τρέχον UTC timestamp χρησιμοποίησε `datetime.now(UTC)`. Για ζώνες με κανόνες θερινής ώρας υπάρχει το `zoneinfo.ZoneInfo` και τα διαθέσιμα timezone δεδομένα του συστήματος ή του αντίστοιχου package.

Το `Decimal` από string κρατά τη δεκαδική τιμή που έγραψες. Αν ξεκινήσεις από `Decimal(0.1)`, εισάγεις ήδη την προσέγγιση του float. Η ακρίβεια και το rounding είναι επιλογές του domain, όχι θέμα ωραίου formatting.

Για TOML configuration, η standard library έχει `tomllib` από την 3.11 για ανάγνωση. Δεν περιλαμβάνει αντίστοιχο TOML writer. Δεν χρειάζεται να προσθέσεις dependency πριν δεις αν η υπάρχουσα standard library λύνει ήδη το πρόβλημα.

Ασκήσεις

Λύσεις κεφαλαίου 15 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

15.1 · JSON round trip

Τι συμβαίνει στην tuple `("ruby", "python")` μετά από `json.loads(json.dumps(...))`; Γράψε τον έλεγχο και εξήγησε γιατί δεν είναι ίδιος τύπος.

Hint: το JSON έχει array, όχι ξεχωριστή tuple.

15.2 · Άθροισμα από CSV

Γράψε `csv_minutes(text)` που διαβάζει CSV string με headers `title,minutes` και επιστρέφει το άθροισμα. Θεώρησε έγκυρα headers και integer πεδία. Βάλε στο test τίτλο με κόμμα και αρχείο με μόνο headers.

Hint: `StringIO`, `DictReader`, μετά μετατροπή του σωστού πεδίου.

15.3 · Αρχεία σε προσωρινό φάκελο

Γράψε UTF-8 κείμενο `"Σημειώσεις Python"` σε αρχείο μέσα σε `TemporaryDirectory`. Διάβασέ το και έλεγξε ότι έμεινε ίδιο. Μην χρησιμοποιήσεις φάκελο με προσωπικά δεδομένα.

Hint: το context manager καθαρίζει τον δικό του προσωρινό φάκελο.

ΚΕΦΑΛΑΙΟ 16

Tests και debugging που ελέγχουν τη σωστή υπόθεση

Το να μεταφράζεις Ruby σε Python απαιτεί έλεγχο της συμπεριφοράς, όχι μόνο αποδοχή από τον parser. Ένα πρόγραμμα μπορεί να είναι έγκυρο συντακτικά και να επιστρέφει `None`, να χάνει το μηδέν ή να καταναλώνει ένα generator δύο φορές.

Από το assert στο test suite

Τα μικρά `assert` του βιβλίου ελέγχουν ότι ο κώδικας δίνει το αποτέλεσμα που περιμένουμε. Σε κανονικό input validation, όμως, δεν χρησιμοποιούμε assert: με `python -O` μπορούν να αφαιρεθούν. Για πραγματικό κανόνα του API χρησιμοποίησε `if` και κατάλληλο exception.

Το `unittest` είναι μέρος της standard library. Θα χρησιμοποιήσουμε μια function που αφαιρεί διπλότυπα κρατώντας σειρά, ώστε το test να έχει σαφές παρατηρήσιμο αποτέλεσμα.

PYTHON

```
def unique_in_order(values):
    return list(dict.fromkeys(values))

import unittest

class UniqueTests(unittest.TestCase):
    def test_keeps_first_occurrence(self):
        result = unique_in_order(["py", "rb", "py"])
        self.assertEqual(result, ["py", "rb"])

    def test_empty_input(self):
        self.assertEqual(unique_in_order([]), [])

    def test_does_not_change_input(self):
        original = [3, 1, 3]
        unique_in_order(original)
        self.assertEqual(original, [3, 1, 3])
```

Σε αρχείο `test_unique.py`, μπορείς να τρέξεις `python -m unittest -v test_unique`. Για φάκελο tests, χρησιμοποίησε `python -m unittest discover -s tests -v` από τον σωστό project root. Τα ονόματα test methods αρχίζουν με `test_`.

Για να είναι και το παράδειγμα του βιβλίου αυτοτελές, το εκτελούμε προγραμματιστικά:

PYTHON

```
from io import StringIO

suite = unittest.defaultTestLoader.loadTestsFromTestCase(
    UniqueTests
)
result = unittest.TextTestRunner(stream=StringIO()).run(suite)
assert result.wasSuccessful()
assert result.testsRun == 3
```

Edge cases που αξίζει να ελέγξεις

Μην γράφεις δέκα παραλλαγές του ίδιου συνηθισμένου input αν καμία δεν ελέγχει το contract. Για την `first_at_least`, χρήσιμα tests είναι: υπάρχει κατάλληλη τιμή, δεν υπάρχει, η είσοδος είναι κενή, η πρώτη τιμή ισούται ακριβώς με το όριο και η πρώτη κατάλληλη δεν είναι η μικρότερη κατάλληλη.

Για function που παίρνει λίστα, έλεγξε mutation αν το contract λέει ότι το input πρέπει να μείνει ίδιο. Για generator, έλεγξε και κατανάλωση. Για parsing, έλεγξε κακή μορφή και μη αποδεκτή τιμή χωριστά. Για sorting, έλεγξε ισοπαλία αν έχεις δεύτερο κριτήριο.

Exceptions και επαναλαμβανόμενα cases

PYTHON

```
def positive(value):
    if value <= 0:
        raise ValueError("must be positive")
    return value

class PositiveTests(unittest.TestCase):
    def test_rejects_nonpositive(self):
        for value in [0, -1, -100]:
            with self.subTest(value=value):
                with self.assertRaises(ValueError):
                    positive(value)

suite = unittest.defaultTestLoader.loadTestsFromTestCase(
    PositiveTests
)
result = unittest.TextTestRunner(stream=StringIO()).run(suite)
assert result.wasSuccessful()
```

Το `subTest` δείχνει ποιο case απέτυχε χωρίς να χρειάζεται νέα method για κάθε τιμή. Το `assertRaises` ελέγχει το exception ως αναμενόμενο αποτέλεσμα. Μην γράφεις `try/except: pass` στο test: μπορεί να περάσει ακόμη κι όταν δεν συμβεί τίποτα.

Το pytest είναι δημοφιλής εναλλακτική με απλά asserts και fixtures. Δεν είναι προϋπόθεση για το βιβλίο. Αν το project σου το χρησιμοποιεί ήδη, οι ίδιες ιδέες μεταφέρονται. Το framework δεν αποφασίζει ποιες συμπεριφορές αξίζει να ελέγξεις.

Διάβασε το traceback από το τέλος

Ξεκίνα από τον τύπο και το μήνυμα του exception, μετά ακολούθησε τα frames προς την κλήση που το προκάλεσε. Μην αλλάζεις τυχαία τον κώδικα μέχρι να σταματήσει το error. Μίκρυνε το input, τύπωσε `repr(value)` και `type(value)` ή χρησιμοποίησε `breakpoint()` για διαδραστική εξέταση.

Στον debugger, τα `p expression`, `n`, `s` και `c` σε βοηθούν να δεις τιμές, να προχωρήσεις στην επόμενη γραμμή, να μπεις σε κλήση ή να συνεχίσεις. Το `breakpoint()` σταματά την εκτέλεση, οπότε δεν πρέπει να μένει σε αυτοματοποιημένα tests που περιμένεις να ολοκληρωθούν μόνα τους.

Μια καλή υπόθεση έχει ελέγξιμη πρόβλεψη: «το dict περιέχει το key με τιμή None» αντί για «η Python δεν διαβάζει σωστά το αρχείο». Ο έλεγχος `key in data` θα ξεχωρίσει αμέσως το missing από το present-but-None.

Ασκήσεις

Λύσεις κεφαλαίου 16 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

16.1 · Ποιο test λείπει;

Η `default_limit(value)` επιστρέφει `value or 20`. Τα tests για `None` και `5` περνούν. Πρόσθεσε το test που αποδεικνύει ότι παραβιάζει την απαίτηση «default μόνο στο None» και διόρθωσε τη function.

Hint: το 0 δεν είναι απουσία τιμής.

16.2 · Έλεγξε το contract της first

Γράψε `unittest.TestCase` για τη `first_at_least` της άσκησης 5.2. Χρησιμοποίησε τουλάχιστον τέσσερα cases, μαζί με κενή είσοδο και τιμή ακριβώς στο όριο.

Hint: κάθε case πρέπει να αποκλείει μια διαφορετική πιθανή λανθασμένη υλοποίηση.

16.3 · Error που πρέπει να συμβεί

Γράψε test ότι το `positive(0)` σηκώνει `ValueError`. Μετά πρόσθεσε επιτυχές test για το `positive(1)`. Εξήγησε γιατί χρειάζονται και τα δύο.

Hint: μια function που σηκώνει exception για κάθε input θα περνούσε μόνο το πρώτο test.

ΚΕΦΑΛΑΙΟ 17

Async και await: να ξέρεις τι περιμένει τι

Δεν χρειάζεται async για ένα απλό script που διαβάζει μικρό CSV. Χρειάζεται όμως να μπορείς να το διαβάσεις όταν συναντήσεις σύγχρονη Python που περιμένει δίκτυο, queues ή άλλες ασύγχρονες λειτουργίες.

Coroutine function, coroutine object, αποτέλεσμα

PYTHON

```
import asyncio

async def read_label(name):
    await asyncio.sleep(0)
    return name.upper()

async def main():
    coroutine = read_label("python")
    result = await coroutine
    assert result == "PYTHON"

asyncio.run(main())
```

Το `async def` ορίζει coroutine function. Η κλήση της δημιουργεί coroutine object. Το σώμα δεν έχει ολοκληρωθεί απλώς επειδή έγινε η κλήση. Το `await` περιμένει το αποτέλεσμα ενός awaitable και μπορεί να παραδώσει έλεγχο στο event loop όταν η λειτουργία πρέπει να περιμένει.

Το `asyncio.run()` ξεκινά και διαχειρίζεται event loop για το κύριο coroutine. Σε απλό script είναι το entry point. Σε περιβάλλον που έχει ήδη ενεργό event loop, όπως αρκετά notebooks ή frameworks, δεν καλείς δεύτερο `asyncio.run()` μέσα στο ίδιο thread. Εκεί χρησιμοποιείς το async entry point που παρέχει το περιβάλλον.

Δύο await στη σειρά παραμένουν διαδοχικά

PYTHON

```
async def sequential():
    first = await read_label("ruby")
    second = await read_label("python")
    return [first, second]

assert asyncio.run(sequential()) == ["RUBY", "PYTHON"]
```

Το δεύτερο coroutine ξεκινά αφού τελειώσει το πρώτο. Το `async` δεν δημιουργεί αυτόματα concurrency. Για να προχωρούν περισσότερα coroutines όσο τα άλλα περιμένουν, τα ξεκινάς ως tasks.

TaskGroup: τα tasks έχουν κοινό lifetime

PYTHON

```
async def concurrent_labels(names):
    async with asyncio.TaskGroup() as group:
        tasks = [
            group.create_task(read_label(name))
            for name in names
        ]
    return [task.result() for task in tasks]

assert asyncio.run(concurrent_labels(["ruby", "python"])) == [
    "RUBY", "PYTHON"
]
```

Το `TaskGroup`, διαθέσιμο από την 3.11, περιμένει τα tasks κατά την έξοδο. Η λίστα αποτελεσμάτων κρατά τη σειρά που εμείς κρατήσαμε τα task references, όχι μια υποτιθέμενη σειρά ολοκλήρωσης. Αν ένα task αποτύχει με συνηθισμένο exception, τα υπόλοιπα γίνονται cancel και το group περιμένει να ολοκληρωθεί το cleanup τους.

Οι αποτυχίες μπορούν να φτάσουν ως `ExceptionGroup`. Η σύνταξη `except*` χειρίζεται τις αντίστοιχες υποομάδες. Δεν θα γράφεις παντού αυτό το επίπεδο error handling, αλλά πρέπει να μπορείς να το αναγνωρίζεις.

PYTHON

```
async def broken():
    raise ValueError("bad item")

async def handle_group():
    messages = []
    try:
        async with asyncio.TaskGroup() as group:
            group.create_task(broken())
    except* ValueError as errors:
        messages.extend(str(error) for error in errors.exceptions)
    return messages

assert asyncio.run(handle_group()) == ["bad item"]
```

Cleanup και blocking code

Το cancellation εμφανίζεται συνήθως ως `asyncio.CancelledError` όταν η coroutine ξαναπάρει έλεγχο σε κατάλληλο σημείο. Χρησιμοποίησε `try/finally` για cleanup και μην καταπίνεις το `CancelledError` χωρίς συγκεκριμένο λόγο. Μηχανισμοί όπως το TaskGroup βασίζονται στο cancellation.

Μια συνηθισμένη synchronous κλήση δεν γίνεται non-blocking επειδή βρίσκεται μέσα σε `async def`. Το `time.sleep()` μπλοκάρει το thread, ενώ το `await asyncio.sleep()` επιτρέπει σε άλλα tasks να προχωρήσουν. Για υπάρχουσα blocking I/O function μπορεί να βοηθήσει το `await asyncio.to_thread(function, ...)`.

Αυτό δεν αποτελεί γενική λύση για CPU-heavy υπολογισμούς. Η συνηθισμένη CPython έχει GIL. Οι προαιρετικές free-threaded εκδόσεις και τα native extensions αλλάζουν τα δεδομένα, αλλά δεν κάνουν αυτόματα σωστό έναν κώδικα που τρέχει παράλληλα. Με async, τα tasks δίνουν ευκαιρία στα υπόλοιπα να τρέξουν όταν φτάνουν σε await που περιμένει.

Async iteration

PYTHON

```
async def labels(names):
    for name in names:
        yield await read_label(name)

async def consume():
    result = []
    async for label in labels(["ruby", "python"]):
        result.append(label)
    return result

assert asyncio.run(consume()) == ["RUBY", "PYTHON"]
```

Το `async for` ζητά το επόμενο στοιχείο ασύγχρονα. Το `async with` περιμένει την είσοδο και την έξοδο από async context manager. Δεν αντικαθιστούν τα `for` και `with` για κάθε object. Χρειάζονται το αντίστοιχο async protocol.

Ασκήσεις

Λύσεις κεφαλαίου 17 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

17.1 · Δημιουργήθηκε ή εκτελέστηκε;

Γράψε coroutine που προσθέτει `"ran"` σε μια λίστα και επιστρέφει 7. Δημιούργησέ τη χωρίς await, έλεγξε ότι η λίστα είναι ακόμη κενή και μετά περίμενέ τη. Μην αφήσεις το coroutine ξεχασμένο χωρίς εκτέλεση.

Hint: κάνε τον έλεγχο μέσα σε ένα async `main` και μετά await το ίδιο object.

17.2 · Map με TaskGroup

Γράψε async `double(n)` που κάνει `await asyncio.sleep(0)` και επιστρέφει `n * 2`. Με TaskGroup φτιάξε `double_all(values)` που κρατά τη σειρά εισόδου. Δοκίμασε `[3, 1, 2]` και `[]`.

Hint: κράτα τα task objects σε λίστα πριν βγεις από το group.

17.3 · Άθροισμα από async generator

Γράψε async generator που παράγει τις τιμές 1, 2, 3, με ένα `await asyncio.sleep(0)` πριν από καθεμία. Κατανάλωσέ τον με `async for` και επέστρεψε 6.

Hint: το σώμα του consumer πρέπει επίσης να είναι async.

ΚΕΦΑΛΑΙΟ 18

Η σημερινή Python: 3.14 και 3.15

Το «σύγχρονο» δεν σημαίνει να χρησιμοποιείς κάθε νέα σύνταξη μόλις εμφανιστεί. Σημαίνει να γνωρίζεις τι υπάρχει, από ποια έκδοση υποστηρίζεται και πότε αξίζει να το χρησιμοποιήσεις. Ο πίνακας αυτός κρατά τα όρια ορατά.

Δυνατότητα	Από	Χρήση στο βιβλίο
`match/case`, `X | None`	3.10	Βασική ύλη
`TaskGroup`, `except*`, `tomllib`	3.11	Βασική ύλη
`def f[T]`, `type Alias = ...`	3.12	Σύγχρονα type hints
T-strings, lazy annotations	3.14	Αυτό το κεφάλαιο
Unpacking σε comprehensions	3.15	Νέα σύνταξη, ξεχωριστά
`sentinel`, `frozendict`, `lazy import`	3.15	Νέα εργαλεία, ξεχωριστά

Τα παραδείγματα με ετικέτα **MONO PYTHON 3.15** χρειάζονται interpreter 3.15. Η αναφορά τους έχει διασταυρωθεί με την επίσημη τεκμηρίωση. Οι τοπικοί runtime έλεγχοι του βιβλίου καλύπτουν την 3.14.7, όχι αυτά τα snippets.

3.14: το t-string δεν είναι str

PYTHON

```
from string.templatelib import Template

name = "Ada"
template = t"Hello, {name}!"
assert isinstance(template, Template)
assert template.strings == ("Hello, ", "!")
assert template.values == ("Ada",)
assert template.interpolations[0].expression == "name"
```

Ένα f-string συνδυάζει τα κομμάτια του σε str. Ένα t-string δίνει Template με χωριστά σταθερά κομμάτια και interpolations. Οι expressions μέσα στα braces αξιολογούνται κατά τη δημιουργία του template. Αναβάλλεται ο τελικός τρόπος επεξεργασίας τους, όχι αυθαίρετα όλη η εκτέλεση.

Αυτό επιτρέπει σε μια library να επεξεργαστεί τις τιμές με συγκεκριμένους κανόνες. Δεν κάνει ένα τυχαίο template αυτόματα ασφαλές SQL ή HTML. Χρειάζεται ο κατάλληλος consumer. Για το απλό `print` ενός μηνύματος, το f-string παραμένει η φυσική επιλογή. Τεκμηρίωση t-strings.

3.14: annotations που αξιολογούνται αργότερα

PYTHON

```
def keep_node(node: Node) -> Node:
    return node

class Node:
    pass

from typing import get_type_hints

hints = get_type_hints(keep_node)
assert hints["node"] is Node
```

Το forward reference μπορεί εδώ να οριστεί μετά τη function. Η ανάκτηση resolved hints γίνεται αφού υπάρχει το `Node`. Αυτό δεν αναιρεί τον δυναμικό χαρακτήρα της Python και δεν προσθέτει αυτόματο runtime validation. Η αξιολόγηση annotations μπορεί να εκτελέσει expressions. Αν γράφεις library που τα εξετάζει, διάβασε το `annotationlib` και το migration guide αντί να υποθέτεις παλαιά eager semantics. Annotationlib.

Η 3.14 επιτρέπει επίσης πολλαπλούς τύπους στο `except` χωρίς parentheses όταν δεν χρησιμοποιείται `as`. Η παλιότερη μορφή `except (ValueError, TypeError):` παραμένει σωστή και είναι ξεκάθαρη, οπότε τη διατηρούμε στα περισσότερα παραδείγματα.

3.15: unpacking μέσα σε comprehension

MONO PYTHON 3.15

```
groups = [[1, 2], [], [3]]
flat = [*group for group in groups]
assert flat == [1, 2, 3]

parts = [{"a": 1}, {"b": 2}, {"a": 3}]
combined = {**part for part in parts}
assert combined == {"a": 3, "b": 2}
```

Στην 3.14, το flatten γράφεται `[item for group in groups for item in group]`. Η νέα μορφή ξεδιπλώνει κάθε παραγόμενο iterable ή mapping. Στα mappings, οι μεταγενέστερες τιμές του ίδιου key υπερισχύουν. Η σύντομη γραφή δεν είναι

συμβατή με παλιότερους parsers, ακόμη κι αν δεν εκτελεστεί ο συγκεκριμένος κλάδος. PEP 798.

3.15: missing που δεν είναι None

MONO PYTHON 3.15

```
MISSING = sentinel("MISSING")

def choose(value=MISSING):
    if value is MISSING:
        return "not supplied"
    return value

assert choose() == "not supplied"
assert choose(None) is None
assert choose(0) == 0
```

Ένα sentinel δίνει μοναδική τιμή για το «δεν δόθηκε argument», ώστε το None να μπορεί να παραμένει έγκυρο input. Η σύγκριση γίνεται με `is`. Στον κορμό 3.14 μπορείς να χρησιμοποιήσεις `MISSING = object()` για το ίδιο απλό identity check, χωρίς όλα τα χαρακτηριστικά του νέου built-in τύπου. Sentinel values.

3.15: immutable mapping και lazy imports

MONO PYTHON 3.15

```
options = frozendict({"mode": "read", "limit": 10})
assert options["limit"] == 10
assert isinstance(hash(options), int)
```

Το `frozendict` δεν επιτρέπει αλλαγή των δικών του entries. Είναι hashable όταν όλα τα keys και values είναι hashable. Δεν κάνει immutable τα objects που περιέχει και δεν είναι subclass του dict. Αν το API χρειάζεται γενικά mapping, η `collections.abc.Mapping` περιγράφει καλύτερα την απαίτηση από ένα αποκλειστικό `dict` check. Built-in types 3.15.

MONO PYTHON 3.15

```
lazy import json

def decode_title(payload):
    return json.loads(payload)["title"]

assert decode_title('{"title": "Python"}') == "Python"
```

Το explicit lazy import μεταθέτει το φόρτωμα μέχρι να χρησιμοποιηθεί το όνομα. Αυτό αλλάζει και το πότε μπορεί να εμφανιστούν import errors ή side effects. Δεν είναι αυτόματη βελτίωση κάθε project. Η σύνταξη έχει περιορισμούς, όπως χρήση σε module scope, και χρειάζεται να ελέγξεις την επίσημη αναφορά πριν αλλάξεις τον τρόπο φόρτωσης μιας εφαρμογής. Python 3.15: lazy imports.

Η 3.15 ενεργοποιεί UTF-8 mode από προεπιλογή. Κράτησε ρητό encoding όταν ορίζει το format των αρχείων σου. UTF-8 mode.

Ασκήσεις

Λύσεις κεφαλαίου 18 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

Οι ασκήσεις του κεφαλαίου λύνονται στην 3.14, ώστε να μη χρειάζεται δεύτερος interpreter για να ολοκληρώσεις το βιβλίο.

18.1 · Αναγνώρισε το template

Φτιάξε `t"Book: {title}"` για `title = "Python"`. Έλεγξε τα `.strings` και `.values`. Είναι το αποτέλεσμα str; Πότε αξιολογήθηκε το title;

Hint: κράτα χωριστά δημιουργία template και τελικό rendering.

18.2 · Flatten που δουλεύει στην 3.14

Μετάφρασε το νέο `[*group for group in groups]` σε συμβατό comprehension για 3.14. Δοκίμασε `[[], [1], [2, 3]]` και `[]`.

Hint: χρειάζεσαι δύο `for`, με την ίδια σειρά που θα είχαν σε nested loop.

18.3 · Missing, None και μηδέν

Με `MISSING = object()` γράψε `pick(value=MISSING)` που επιστρέφει `"missing"` μόνο όταν δεν δόθηκε argument. Τα `None`, `0` και κενό string πρέπει να διατηρούνται.

Hint: κανένα truthiness check δεν μπορεί να εκφράσει αυτή την απαίτηση.

ΚΕΦΑΛΑΙΟ 19

Τελικό project: ένα μικρό readlog

Θα φτιάξουμε εργαλείο που διαβάζει συνεδρίες ανάγνωσης από CSV και εμφανίζει τα συνολικά λεπτά ανά βιβλίο. Το project είναι αρκετά μικρό για να το καταλάβεις ολόκληρο και αρκετά πραγματικό ώστε να ενώσει όσα έμαθες.

Το contract πριν από τον κώδικα

Το αρχείο έχει ακριβώς τα headers `title`, `minutes`, `day`, με οποιαδήποτε σειρά. Κάθε τίτλος καθαρίζεται με strip και πρέπει να μην είναι κενός. Τα λεπτά είναι μη αρνητικός integer. Η ημερομηνία πρέπει να γίνεται δεκτή από το `date.fromisoformat`, με τα παραδείγματά μας σε μορφή `YYYY-MM-DD`.

Οι τίτλοι με διαφορετική γραφή παραμένουν διαφορετικοί. Δεν συγχωνεύουμε σιωπηρά `Python` και `PYTHON`. Επαναλαμβανόμενοι ίδιοι τίτλοι αθροίζονται. Η αναφορά ταξινομείται πρώτα κατά συνολικά λεπτά φθίνουσα και μετά αλφαβητικά κατά τίτλο. Αρχείο με μόνο headers δίνει σύνολο 0. Άκυρη γραμμή απορρίπτει το αρχείο με μήνυμα, ώστε να μη βγάζουμε φαινομενικά πλήρη αναφορά από μέρος των δεδομένων.

OUTPUT

```
project/
  sessions.csv
  readlog/
    __init__.py
    core.py
    cli.py
    __main__.py
  tests/
    test_readlog.py
```

Το sample `sessions.csv`:

CSV

```
title,minutes,day
Python basics,25,2026-10-14
Ruby patterns,15,2026-10-14
"Python, daily",35,2026-10-15
Python basics,20,2026-10-16
```

Βήμα 1: το μοντέλο και το parsing

Το `__init__.py` κρατά μόνο το docstring του package:

readlog/__init__.py

```
"""Summarize reading sessions from CSV."""
```

Το `core.py` περιέχει το μοντέλο, το parsing και την καθαρή function συγκέντρωσης. Παίρνουμε stream αντί για filename, ώστε τα tests να μπορούν να δίνουν `StringIO`.

readlog/core.py

```
import csv
from collections import Counter
from collections.abc import Iterable
from dataclasses import dataclass
from datetime import date
from typing import TextIO

@dataclass(frozen=True)
class Session:
    title: str
    minutes: int
    day: date

def parse_sessions(stream: TextIO) -> list[Session]:
    reader = csv.DictReader(stream, strict=True)
    expected = {"title", "minutes", "day"}
    headers = reader.fieldnames
    if (headers is None or len(headers) != 3
            or set(headers) != expected):
        raise ValueError("expected headers: title,minutes,day")

    sessions = []
    for row in reader:
        line = reader.line_num
        if None in row or any(v is None for v in row.values()):
            raise ValueError(f"line {line}: wrong field count")
        title = row["title"].strip()
        try:
            minutes = int(row["minutes"])
            day = date.fromisoformat(row["day"].strip())
        except ValueError as error:
            raise ValueError(
                f"line {line}: invalid minutes or day"
            ) from error
        if not title or minutes < 0:
            raise ValueError(
                f"line {line}: empty title or negative minutes"
            )
        sessions.append(Session(title, minutes, day))
    return sessions

def summarize(sessions: Iterable[Session]) -> list[tuple[str, int]]:
    totals: Counter[str] = Counter()
    for session in sessions:
        totals[session.title] += session.minutes
    return sorted(totals.items(), key=lambda row: (-row[1], row[0]))
```

Το `Counter` κάνει τη συσσώρευση σαφή. Το `frozen=True` ταιριάζει σε μια εγγραφή που παράχθηκε από το parsing και μετά διαβάζεται. Τα annotations βοηθούν τον αναγνώστη, αλλά οι έλεγχοι του parser είναι αυτοί που επιβάλλουν το contract.

Η `reader.line_num` αναφέρεται στις φυσικές γραμμές που διάβασε ο CSV parser. Σε εγγραφή με quoted newline μπορεί να δείχνει την τελευταία γραμμή της εγγραφής. Αυτό είναι ακριβέστερο από το να υποθέσουμε ότι κάθε CSV record καταλαμβάνει μία γραμμή κειμένου.

Βήμα 2: το όριο του CLI

Το `cli.py` είναι υπεύθυνο για arguments, άνοιγμα αρχείου και εμφάνιση. Τα imports από το core δεν κάνουν I/O. Η standard library `argparse` παρέχει usage και `--help`.

readlog/cli.py

```
import argparse
import csv
import sys
from pathlib import Path
from .core import parse_sessions, summarize

def main(argv: list[str] | None = None) -> int:
    parser = argparse.ArgumentParser(
        description="Summarize reading minutes by book."
    )
    parser.add_argument("path", type=Path, help="CSV input file")
    args = parser.parse_args(argv)
    try:
        with args.path.open(
            encoding="utf-8-sig", newline=""
        ) as stream:
            sessions = parse_sessions(stream)
    except (OSError, UnicodeError, ValueError, csv.Error) as error:
        print(f"error: {error}", file=sys.stderr)
        return 2

    rows = summarize(sessions)
    for title, minutes in rows:
        print(f"{minutes:>4} min  {title}")
    print(f"TOTAL {sum(minutes for _, minutes in rows)} min")
    return 0
```

Το `utf-8-sig` διαβάζει UTF-8 με ή χωρίς αρχικό BOM, που μπορεί να βρεις σε CSV export. Ο handler καλύπτει συγκεκριμένες αποτυχίες του ορίου εισόδου. Δεν πιάνει οποιοδήποτε `Exception` του προγράμματος και δεν μετατρέπει κάθε bug σε «κακό CSV».

Το `__main__.py` μετατρέπει το return value σε exit status:

readlog/__main__.py

```
from .cli import main

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

Από τον φάκελο `project`, εκτελείς:

TERMINAL

```
python -m readlog sessions.csv
python -m readlog --help
```

Η αναφορά του sample είναι:

OUTPUT

```
  45 min  Python basics
  35 min  Python, daily
  15 min  Ruby patterns
TOTAL 95 min
```

Βήμα 3: tests με χρήσιμες διακρίσεις

Ελέγχουμε συγκέντρωση, ισοπαλία, κενή είσοδο και αποτυχίες. Δεν εξαρτόμαστε από δίκτυο ή εξωτερική υπηρεσία. Το παράδειγμα κρατά τα tests αρκετά μικρά για να βλέπεις ποια απαίτηση καλύπτει το καθένα.

tests/test_readlog.py

```
import unittest
from io import StringIO
from readlog.core import parse_sessions, summarize

HEADER = "title,minutes,day\n"

class ReadlogTests(unittest.TestCase):
    def test_groups_and_sorts(self):
        text = HEADER + (
            "Ruby,15,2026-10-14\n"
            "Python,25,2026-10-15\n"
            "Ruby,20,2026-10-16\n"
        )
        rows = summarize(parse_sessions(StringIO(text)))
        self.assertEqual(rows, [("Ruby", 35), ("Python", 25)])

    def test_tie_uses_title(self):
        text = HEADER + (
            "Ruby,20,2026-10-14\n"
            "Python,20,2026-10-15\n"
        )
        rows = summarize(parse_sessions(StringIO(text)))
        self.assertEqual(rows, [("Python", 20), ("Ruby", 20)])

    def test_headers_only(self):
        self.assertEqual(parse_sessions(StringIO(HEADER)), [])

    def test_zero_and_quoted_comma(self):
        text = HEADER + '"Python, daily",0,2026-10-16\n'
        session, = parse_sessions(StringIO(text))
        self.assertEqual(session.title, "Python, daily")
        self.assertEqual(session.minutes, 0)

    def test_rejects_invalid_rows(self):
        invalid = [
            "Python,-1,2026-10-16\n",
            "Python,many,2026-10-16\n",
```

```
            "Python,10,2026-02-30\n",
            " ,10,2026-10-16\n",
            "Python,10\n",
            "Python,10,2026-10-16,extra\n",
        ]
        for row in invalid:
            with self.subTest(row=row):
                with self.assertRaises(ValueError):
                    parse_sessions(StringIO(HEADER + row))

    def test_rejects_wrong_headers(self):
        for text in ["", "title,minutes\n", "title,title,day\n"]:
            with self.subTest(text=text):
                with self.assertRaises(ValueError):
                    parse_sessions(StringIO(text))
```

Η εντολή εκτέλεσης είναι `python -m unittest discover -s tests -v`. Στο συνοδευτικό project υπάρχουν επίσης tests του CLI για επιτυχημένο αρχείο, missing αρχείο, κακό encoding και άκυρο CSV. Αυτά ελέγχουν ότι το πρόγραμμα επιστρέφει το σωστό exit status και δεν τυπώνει μερική αναφορά μετά από αποτυχία.

Ασκήσεις

Λύσεις κεφαλαίου 19 · Δοκίμασε πρώτα χωρίς να τις ανοίξεις.

19.1 · Φίλτρο μετά τη συγκέντρωση

Γράψε `at_least(rows, minimum)` που δέχεται το αποτέλεσμα της `summarize` και κρατά τίτλους με συνολικά λεπτά τουλάχιστον `minimum`, διατηρώντας την υπάρχουσα σειρά. Δοκίμασε όριο 35 στο sample: μένουν δύο τίτλοι.

Hint: το φίλτρο μπαίνει μετά τη συγκέντρωση, όχι στις ξεχωριστές συνεδρίες.

19.2 · Ημέρες με ανάγνωση

Γράψε `active_days(sessions)` που μετρά διαφορετικές ημερομηνίες με τουλάχιστον μία συνεδρία θετικών λεπτών. Οι εγγραφές μηδενικών λεπτών δεν προσθέτουν ενεργή ημέρα. Δοκίμασε δύο sessions στην ίδια μέρα.

Hint: set comprehension πάνω στο σωστό φίλτρο.

19.3 · JSON αναφορά

Γράψε `report_json(rows)` που μετατρέπει τα συνολικά rows σε JSON array με objects `{"title": ..., "minutes": ...}`. Διατήρησε τη σειρά. Έλεγξε το περιεχόμενο με `json.loads`, όχι συγκρίνοντας τυχαία spaces του serialized string.

Hint: πρώτα φτιάξε κανονικά Python δεδομένα, μετά κάνε serialization.

ΣΥΓΚΡΙΝΕ ΤΗ ΣΚΕΨΗ ΣΟΥ

Λύσεις και τρόπος σκέψης

Οι παρακάτω λύσεις είναι ενδεικτικές. Μια διαφορετική υλοποίηση είναι σωστή όταν ικανοποιεί το ίδιο contract και παραμένει κατανοητή. Πριν συγκρίνεις κώδικα, σύγκρινε τις περιπτώσεις που δοκίμασες. Κάθε λύση με κώδικα συνοδεύεται από ελέγχους και διατίθεται ως ξεχωριστό αρχείο στο `code/solutions`.

Κεφάλαιο 1 · Το πρώτο script

1.1 · Το χαμένο αποτέλεσμα

Η αρχική function επιστρέφει `None`. Το expression του σώματος δεν γίνεται αυτόματα return.

PYTHON

```
def double(number):
    return number * 2

assert double(6) == 12
assert double(0) == 0
```

1.2 · Μια κάρτα ονόματος

Το return κάνει τη function χρήσιμη και για tests και για μελλοντική σύνθεση. Το πού θα εμφανιστεί το αποτέλεσμα είναι ευθύνη του caller.

PYTHON

```
def badge(name, language):
    return f"{name} · {language}"

assert badge("Ada", "Python") == "Ada · Python"
assert badge("Matz", "Ruby") == "Matz · Ruby"
```

1.3 · Διάβασε το indentation

Το όριο 18 ανήκει στην πρώτη περίπτωση. Το σημαντικό test είναι ακριβώς στο όριο, όχι μόνο πολύ πάνω ή πολύ κάτω από αυτό.

PYTHON

```
def access_label(age):
    return "adult" if age >= 18 else "minor"

assert access_label(17) == "minor"
assert access_label(18) == "adult"
assert access_label(19) == "adult"
```

Κεφάλαιο 2 · Τιμές και truthiness

2.1 · Μηδέν ή απουσία;

Το `value or 20` δίνει διαδοχικά 20, 20, 5. Η απαίτηση όμως ζητά 20, 0, 5. Ελέγχουμε συγκεκριμένα την απουσία.

PYTHON

```
def default_limit(value):
    return 20 if value is None else value

assert default_limit(None) == 20
assert default_limit(0) == 0
assert default_limit(5) == 5
```

2.2 · Χρόνος ανάγνωσης

Το quotient είναι οι ολόκληρες ώρες και το remainder τα λεπτά που περισσεύουν. Δεν χρειάζεται float διαίρεση και μεταγενέστερη στρογγυλοποίηση.

PYTHON

```
def split_minutes(total):
    return divmod(total, 60)

assert split_minutes(135) == (2, 15)
assert split_minutes(0) == (0, 0)
assert split_minutes(60) == (1, 0)
```

2.3 · Δύο ονόματα, μία λίστα

Το append αλλάζει το κοινό object. Το δεύτερο assignment αλλάζει μόνο το binding του `b`. Το `c` είχε από την αρχή διαφορετική εξωτερική λίστα.

PYTHON

```
a = [1, 2]
b = a
c = a.copy()
b.append(3)
b = [9]
assert a == [1, 2, 3]
assert b == [9]
assert c == [1, 2]
assert a is not b
```

Κεφάλαιο 3 · Strings και bytes

3.1 · Ποιο split;

Χωρίς separator, παίρνουμε λέξεις ανάμεσα σε whitespace. Με ρητό space, κάθε space είναι συγκεκριμένο όριο, άρα εμφανίζονται κενά πεδία.

PYTHON

```
text = " red  blue "
assert text.split() == ["red", "blue"]
assert text.split(" ") == ["", "red", "", "blue", ""]
```

3.2 · Καθαρά tags

Το loop κάνει ορατή τη σειρά: split, strip, casefold, έλεγχος, append. Τα διπλότυπα διατηρούνται επειδή αυτό ζητά η άσκηση.

PYTHON

```
def clean_tags(text):
    tags = []
    for piece in text.split(","):
        tag = piece.strip().casefold()
        if tag:
            tags.append(tag)
    return tags

assert clean_tags(" Ruby, python , , RUBY ") == [
    "ruby", "python", "ruby"
]
assert clean_tags(" , ") == []
```

3.3 · Μικρή προεπισκόπηση

Οι τελείες προστίθενται μόνο όταν κόπηκε κείμενο. Το limit 0 δίνει `"..."` για μη κενό text και κενό string για κενό text.

PYTHON

```
def preview(text, limit):
    if len(text) <= limit:
        return text
    return text[:limit] + "..."

assert preview("python", 3) == "pyt..."
assert preview("hi", 3) == "hi"
assert preview("hi", 0) == "..."
assert preview("", 0) == ""
assert preview("hey", 3) == "hey"
```

Κεφάλαιο 4 · Collections

4.1 · Διόρθωσε το sorting

Αν θέλουμε mutation, καλούμε `.sort()` χωρίς assignment. Αν θέλουμε νέο αποτέλεσμα, χρησιμοποιούμε `sorted`.

PYTHON

```
names = ["Matz", "Ada"]
ordered = sorted(names)
assert ordered == ["Ada", "Matz"]
assert names == ["Matz", "Ada"]
names.sort()
assert names == ["Ada", "Matz"]
```

4.2 · Μοναδικά με σειρά

Το `dict.fromkeys` φτιάχνει ένα key για κάθε διαφορετική τιμή. Η επανεμφάνιση υπάρχοντος key δεν αλλάζει τη θέση του.

PYTHON

```
def unique_in_order(values):
    return list(dict.fromkeys(values))

assert unique_in_order(["py", "rb", "py", "go", "rb"]) == [
    "py", "rb", "go"
]
assert unique_in_order([]) == []
```

4.3 · Ανεξάρτητες σειρές

Η δημιουργία του εσωτερικού list συμβαίνει τρεις διαφορετικές φορές. Η μορφή με comprehension, `[[0, 0] for _ in range(3)]`, θα ήταν επίσης σωστή.

PYTHON

```
grid = []
for _ in range(3):
    grid.append([0, 0])
grid[0][0] = 7
assert grid == [[7, 0], [0, 0], [0, 0]]
assert grid[0] is not grid[1]
```

Κεφάλαιο 5 · Control flow και loops

5.1 · Το else που τρέχει

Το `else` καλύπτει την εξάντληση χωρίς εύρεση, συμπεριλαμβανομένης της κενής εισόδου.

PYTHON

```
def find_four(values):
    for value in values:
        if value == 4:
            message = "found"
            break
    else:
        message = "missing"
    return message

assert find_four([1, 2, 3]) == "missing"
assert find_four([]) == "missing"
assert find_four([1, 4, 3]) == "found"
```

5.2 · Η πρώτη αρκετά μεγάλη τιμή

Το early return διατηρεί τη σημασία «πρώτη στη σειρά εισόδου». Δεν κάνουμε sort και δεν ψάχνουμε minimum.

PYTHON

```
def first_at_least(values, limit):
    for value in values:
        if value >= limit:
            return value
    return None

assert first_at_least([2, 8, 5], 5) == 8
assert first_at_least([5, 8], 5) == 5
assert first_at_least([1, 2], 5) is None
assert first_at_least([], 5) is None
```

5.3 · Δύο στήλες που πρέπει να ταιριάζουν

Το `dict` καταναλώνει το zip, άρα ενεργοποιεί και τον έλεγχο του strict μήκους. Χωρίς κατανάλωση, η απλή δημιουργία zip δεν θα αρκούσε για να εμφανιστεί το error.

PYTHON

```
def make_scores(names, scores):
    return dict(zip(names, scores, strict=True))

assert make_scores(["Ada", "Matz"], [8, 9]) == {
    "Ada": 8, "Matz": 9
}
assert make_scores([], []) == {}
try:
    make_scores(["Ada"], [])
except ValueError:
    pass
else:
    raise AssertionError("different lengths were accepted")
```

Κεφάλαιο 6 · Functions

6.1 · Διάβασε τη signature

Οι `scale(3)` και `scale(3, 4, offset=1)` είναι έγκυρες και δίνουν 6 και 13. Οι άλλες δύο παραβιάζουν positional-only και keyword-only όρια αντίστοιχα.

PYTHON

```
def scale(value, /, factor=2, *, offset=0):
    return value * factor + offset

assert scale(3) == 6
assert scale(3, 4, offset=1) == 13
for invalid_call in [
    lambda: scale(value=3),
    lambda: scale(3, 4, 1),
]:
    try:
        invalid_call()
    except TypeError:
        pass
    else:
        raise AssertionError("invalid signature was accepted")
```

6.2 · Tags χωρίς κοινό mutable state

Το `None` λύνει το κοινό default. Η αντιγραφή λύνει το ξεχωριστό θέμα mutation της λίστας του caller.

PYTHON

```
def add_tag(tag, tags=None):
    result = [] if tags is None else list(tags)
    result.append(tag)
    return result

existing = ["ruby"]
assert add_tag("python", existing) == ["ruby", "python"]
assert existing == ["ruby"]
assert add_tag("a") == ["a"]
assert add_tag("b") == ["b"]
```

6.3 · Μια function που φτιάχνει functions

Κάθε κλήση της factory δημιουργεί ξεχωριστό περιβάλλον με το δικό της factor. Δεν χρειαζόμαστε `nonlocal` επειδή δεν αναθέτουμε νέα τιμή στο factor.

PYTHON

```
def make_multiplier(factor):
    def multiply(value):
        return value * factor
    return multiply

triple = make_multiplier(3)
double = make_multiplier(2)
assert triple(4) == 12
assert double(4) == 8
assert triple(0) == 0
```

Κεφάλαιο 7 · Comprehensions

7.1 · Το λάθος compact

Το `None` είναι η μόνη τιμή που αφαιρείται. Δεν χάνουμε 0, False ή κενό string.

PYTHON

```
values = [None, 0, False, "", 3]
result = [value for value in values if value is not None]
assert result == [0, False, "", 3]
assert len(result) == 4
assert result[1] is False
```

7.2 · Μικρό leaderboard

Το αρνητικό score αντιστρέφει μόνο το sorting με βάση το πρώτο στοιχείο. Ένα γενικό `reverse=True` στο tuple key θα αντέστρεφε και την αλφαβητική σειρά.

PYTHON

```
def leaderboard(rows):
    return sorted(rows, key=lambda row: (-row[1], row[0]))

rows = [("Matz", 8), ("Ada", 8), ("Grace", 10)]
assert leaderboard(rows) == [
    ("Grace", 10), ("Ada", 8), ("Matz", 8)
]
assert rows[0] == ("Matz", 8)
assert leaderboard([]) == []
```

7.3 · Λέξεις ανά μήκος

Κάθε μήκος παίρνει δική του λίστα. Το iteration στην αρχική σειρά μαζί με append διατηρεί τη σειρά μέσα στην ομάδα.

PYTHON

```
from collections import defaultdict

def group_by_length(words):
    groups = defaultdict(list)
    for word in words:
        groups[len(word)].append(word)
    return dict(groups)

assert group_by_length(["go", "ruby", "py", "java"]) == {
    2: ["go", "py"], 4: ["ruby", "java"]
}
assert group_by_length([]) == {}
```

Κεφάλαιο 8 · Iterators και generators

8.1 · Η δεύτερη διαδρομή

Η πρώτη κατανάλωση δίνει `[0, 2, 4]`, η δεύτερη `[]`. Αν χρειάζεσαι επανάληψη πάνω στο ίδιο snapshot, αποθήκευσε το αποτέλεσμα σε list.

PYTHON

```
numbers = (n * 2 for n in range(3))
assert list(numbers) == [0, 2, 4]
assert list(numbers) == []

snapshot = [n * 2 for n in range(3)]
assert list(snapshot) == [0, 2, 4]
assert list(snapshot) == [0, 2, 4]
```

8.2 · Τρέχον άθροισμα

Το state ανήκει στη συγκεκριμένη εκτέλεση του generator. Κάθε yield δίνει το άθροισμα μετά την τρέχουσα τιμή.

PYTHON

```
def running_total(values):
    total = 0
    for value in values:
        total += value
        yield total

assert list(running_total([3, 2, 5])) == [3, 5, 10]
assert list(running_total([3, -5, 2])) == [3, -2, 0]
assert list(running_total([])) == []
```

8.3 · Τα πρώτα χρήσιμα στοιχεία

Το `islice` ζητά μόνο όσα στοιχεία χρειάζεται. Το test με count 0 ελέγχει και ότι δεν αγγίξαμε την πηγή, όχι μόνο ότι επιστρέψαμε κενή λίστα.

PYTHON

```
from itertools import islice

def first_nonempty(lines, count):
    cleaned = (line.strip() for line in lines)
    useful = (line for line in cleaned if line)
    return list(islice(useful, count))

assert first_nonempty([" ", " a ", "b", "c"], 2) == ["a", "b"]
assert first_nonempty([], 3) == []
source = iter(["untouched"])
assert first_nonempty(source, 0) == []
assert next(source) == "untouched"
```

Κεφάλαιο 9 · Exceptions

9.1 · Η σειρά του cleanup

Η σειρά είναι enter, body, exit, caught. Το exception περνά από το `yield` και ενεργοποιεί το finally πριν φτάσει στον εξωτερικό handler.

PYTHON

```
from contextlib import contextmanager

@contextmanager
def recorded(events):
    events.append("enter")
    try:
        yield
    finally:
        events.append("exit")

events = []
try:
    with recorded(events):
        events.append("body")
        raise ValueError("example")
except ValueError:
    events.append("caught")
assert events == ["enter", "body", "exit", "caught"]
```

9.2 · Έγκυρο πλήθος

Η `int` αναλαμβάνει το invalid text. Ο δικός μας έλεγχος αναλαμβάνει το θετικό όριο. Και οι δύο αποτυχίες έχουν τον συμφωνημένο τύπο `ValueError`.

PYTHON

```
def positive_count(text):
    count = int(text)
    if count <= 0:
        raise ValueError("count must be positive")
    return count

assert positive_count("3") == 3
for value in ["zero", "0", "-1"]:
    try:
        positive_count(value)
    except ValueError:
        pass
    else:
        raise AssertionError(f"accepted {value!r}")
```

9.3 · Κλείνει και στο error;

Το resource cleanup δεν εξαρτάται από την επιτυχή ολοκλήρωση του σώματος.

```python
from io import StringIO

try:
    with StringIO("notes") as stream:
        assert not stream.closed
        raise RuntimeError("interrupted")
except RuntimeError:
    pass
assert stream.closed
```

Κεφάλαιο 10 · Modules

10.1 · Import χωρίς εκπλήξεις

Το `if __name__ == "__main__":` προστατεύει την κλήση. Το παρακάτω test χρησιμοποιεί το package του κεφαλαίου 10. Τρέχει με τον φάκελο `project` διαθέσιμο στο import path, όπως περιγράφεται στο συνοδευτικό README.

```python
from tinybook.core import reading_hours

assert reading_hours(30) == 0.5
assert reading_hours(0) == 0
```

10.2 · Ξεχώρισε τη μετατροπή

Ο ορισμός μπορεί να μπει στο `core.py`. Τα δύο asserts είναι ο έλεγχος του caller, όχι εντολές που χρειάζεται να τρέχουν στο import του παραγωγικού module.

```python
def minutes_from_hours(hours):
    return hours * 60

assert minutes_from_hours(1.5) == 90
assert minutes_from_hours(0) == 0
```

10.3 · Το λάθος όνομα

Το πιθανότερο είναι ότι το δικό σου `csv.py` κάνει shadowing στο module της standard library. Δες το `csv.__file__` από την ίδια εκτέλεση που παρουσιάζει το πρόβλημα. Μετονόμασε το δικό σου module σε κάτι όπως `reading_csv.py`, διόρθωσε τα δικά του imports και ξεκίνα νέο interpreter ή τρέξε ξανά τα tests σε νέο process. Ο ήδη φορτωμένος κώδικας μπορεί να παραμένει στο `sys.modules`, οπότε η μετονομασία δεν αλλάζει όσα έχει ήδη φορτώσει ένα ανοιχτό session. Αν το path είναι σωστό, συνέχισε τη διάγνωση για circular imports αντί να θεωρείς βέβαιη την πρώτη υπόθεση.

Κεφάλαιο 11 · Classes και dataclasses

11.1 · Η κοινή λίστα

Η λίστα ήταν class state. Μεταφέροντας τη δημιουργία της στο `__init__`, αποκτά κάθε instance τη δική του.

PYTHON

```
class Basket:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

a, b = Basket(), Basket()
a.add("Python")
assert a.items == ["Python"]
assert b.items == []
assert a.items is not b.items
```

11.2 · Μια dataclass για σημειώσεις

Το factory παράγει state. Η property υπολογίζει μια τιμή από το τρέχον state, χωρίς να κρατά δεύτερο counter που μπορεί να ξεφύγει.

PYTHON

```
from dataclasses import dataclass, field

@dataclass
class Note:
    title: str
    tags: list[str] = field(default_factory=list)

    @property
    def tag_count(self):
        return len(self.tags)

a, b = Note("A"), Note("B")
a.tags.append("python")
assert a.tag_count == 1
assert b.tag_count == 0
```

11.3 · Ένα μικρό iterable

Η `list(titles)` δημιουργεί νέα λίστα για το εσωτερικό snapshot. Το `__iter__` επιστρέφει νέο iterator κάθε φορά.

PYTHON

```
class Playlist:
    def __init__(self, titles):
        self._titles = list(titles)

    def __len__(self):
        return len(self._titles)

    def __iter__(self):
        return iter(self._titles)

source = ["A", "B"]
playlist = Playlist(source)
source.append("C")
assert len(playlist) == 2
assert list(playlist) == ["A", "B"]
assert list(playlist) == ["A", "B"]
```

Κεφάλαιο 12 · Types

12.1 · Annotation ή validation;

Η πρώτη function επιστρέφει κανονικά το string. Η δεύτερη κάνει τον ρητό runtime έλεγχο που ζητήθηκε. Ο αποκλεισμός subclasses εδώ είναι συγκεκριμένη απαίτηση της άσκησης, όχι γενικός κανόνας για όλα τα APIs.

PYTHON

```
def identity(value: int) -> int:
    return value

assert identity("hello") == "hello"

def strict_identity(value: object) -> int:
    if type(value) is not int:
        raise TypeError("expected an int, excluding bool")
    return value

assert strict_identity(0) == 0
for value in [True, "3"]:
    try:
        strict_identity(value)
    except TypeError:
        pass
    else:
        raise AssertionError("wrong type accepted")
```

12.2 · Generic τελευταίο στοιχείο

Το T κρατά τη σχέση ανάμεσα στα στοιχεία και στο αποτέλεσμα. Το None καλύπτει την κενή sequence.

PYTHON

```
from collections.abc import Sequence

def last[T](items: Sequence[T]) -> T | None:
    return items[-1] if items else None

assert last([1, 2, 3]) == 3
assert last(("ruby", "python")) == "python"
assert last([]) is None
```

12.3 · Contract χωρίς inheritance

Το Protocol περιγράφει ό,τι διαβάζει η greet. Η Person το ικανοποιεί μέσω του attribute της. Το runtime assert ελέγχει το αποτέλεσμα, όχι το αν ένας static checker συμφωνεί με κάθε annotation του project.

PYTHON

```
from typing import Protocol
from dataclasses import dataclass

class Named(Protocol):
    @property
    def name(self) -> str: ...

@dataclass
class Person:
    name: str

def greet(person: Named) -> str:
    return f"Hello, {person.name}"

assert greet(Person("Ada")) == "Hello, Ada"
```

Κεφάλαιο 13 · Patterns

13.1 · Capture ή σύγκριση;

Το wanted παραμένει εξωτερική τιμή. Το candidate δέχεται το input και το guard κάνει τη σύγκριση.

PYTHON

```
def is_wanted(value, wanted):
    match value:
        case candidate if candidate == wanted:
            return True
        case _:
            return False

assert is_wanted("ready", "ready")
assert not is_wanted("waiting", "ready")
```

13.2 · Μικρός parser εντολών

Το sequence pattern ελέγχει ταυτόχρονα το command και το πλήθος πεδίων. Δεν χρειάζονται διαδοχικά indexing και length checks.

PYTHON

```
def parse_command(parts):
    match parts:
        case ["list"]:
            return "list", None
        case ["show", name]:
            return "show", name
        case _:
            return None

assert parse_command(["list"]) == ("list", None)
assert parse_command(["show", "Python"]) == ("show", "Python")
assert parse_command(["show"]) is None
assert parse_command(["list", "extra"]) is None
```

13.3 · Διάρκεια με σωστό τύπο

Το guard κάνει ρητές δύο απαιτήσεις που δεν εκφράζει μόνο του το int pattern: να αποκλείεται το bool και να επιτρέπονται μόνο μη αρνητικές τιμές.

PYTHON

```
def read_minutes(event):
    match event:
        case {"kind": "read", "minutes": n} if (
            type(n) is int and n >= 0
        ):
            return n
        case _:
            return None

assert read_minutes({"kind": "read", "minutes": 0}) == 0
assert read_minutes({"kind": "read", "minutes": True}) is None
assert read_minutes({"kind": "read", "minutes": -1}) is None
assert read_minutes({"kind": "read", "minutes": "3"}) is None
assert read_minutes({"kind": "read", "minutes": 3, "id": 8}) == 3
assert read_minutes({}) is None
```

Κεφάλαιο 14 · Decorators

14.1 · Τρεις ίδιες απαντήσεις

Το default κρατά την τιμή που υπάρχει όταν δημιουργείται η function. Το πρώτο test αναπαράγει το bug, ώστε να φαίνεται καθαρά τι διόρθωσε η δεύτερη εκδοχή.

PYTHON

```
broken = [lambda: n for n in [10, 20, 30]]
assert [fn() for fn in broken] == [30, 30, 30]

fixed = [lambda n=n: n for n in [10, 20, 30]]
assert [fn() for fn in fixed] == [10, 20, 30]
```

14.2 · Καθάρισε το αποτέλεσμα

Ο wrapper δεν χρειάζεται να ξέρει τη signature της αρχικής function. Προωθεί τα arguments και επεξεργάζεται μόνο το str που επιστρέφεται.

PYTHON

```
from functools import wraps

def stripped(function):
    @wraps(function)
    def wrapper(*args, **kwargs):
        return function(*args, **kwargs).strip()
    return wrapper

@stripped
def label(name, *, prefix=""):
    return f"  {prefix}{name}  "

assert label("Ada", prefix="Dr. ") == "Dr. Ada"
assert label.__name__ == "label"
```

14.3 · Prefix με configuration

Το prefix δεν αποθηκεύεται σε κοινό global state. Κάθε decorator invocation κρατά το δικό του.

PYTHON

```
from functools import wraps

def prefixed(prefix):
    def decorate(function):
        @wraps(function)
        def wrapper(*args, **kwargs):
            return prefix + function(*args, **kwargs)
        return wrapper
    return decorate

@prefixed("INFO: ")
def status():
    return "ready"

@prefixed("NOTE: ")
def note():
    return "saved"

assert status() == "INFO: ready"
assert status() == "INFO: ready"
assert note() == "NOTE: saved"
```

Κεφάλαιο 15 · Αρχεία και μορφές

15.1 · JSON round trip

Η array μορφή του JSON αντιστοιχεί σε list κατά την ανάγνωση. Το format δεν διατηρεί την πληροφορία ότι αρχικά δόθηκε tuple.

PYTHON

```
import json

value = ("ruby", "python")
restored = json.loads(json.dumps(value))
assert restored == ["ruby", "python"]
assert isinstance(restored, list)
assert not isinstance(restored, tuple)
```

15.2 · Άθροισμα από CSV

Ο CSV parser κρατά τον τίτλο με κόμμα ως ένα πεδίο. Η function μετατρέπει μόνο τα λεπτά, σύμφωνα με το contract.

PYTHON

```
import csv
from io import StringIO

def csv_minutes(text):
    rows = csv.DictReader(StringIO(text))
    return sum(int(row["minutes"]) for row in rows)

sample = 'title,minutes\n"Python, daily",25\nRuby,15\n'
assert csv_minutes(sample) == 40
assert csv_minutes("title,minutes\n") == 0
```

15.3 · Αρχεία σε προσωρινό φάκελο

Ο φάκελος υπάρχει μόνο για το lifetime του context. Κάνουμε την ανάγνωση πριν βγούμε από αυτόν.

PYTHON

```
from pathlib import Path
from tempfile import TemporaryDirectory

with TemporaryDirectory() as folder:
    path = Path(folder) / "notes.txt"
    original = "Σημειώσεις Python"
    path.write_text(original, encoding="utf-8")
    assert path.read_text(encoding="utf-8") == original
```

Κεφάλαιο 16 · Tests

16.1 · Ποιο test λείπει;

Ο έλεγχος του μηδενός χωρίζει την επιθυμητή συμπεριφορά από την truthiness εκδοχή. Δείχνουμε πρώτα γιατί η λανθασμένη function δεν συμφωνεί με το αναμενόμενο.

PYTHON

```
def broken_default(value):
    return value or 20

assert broken_default(0) != 0

def default_limit(value):
    return 20 if value is None else value

assert default_limit(0) == 0
assert default_limit(None) == 20
assert default_limit(5) == 5
```

16.2 · Έλεγξε το contract της first

Το case `[2, 8, 5]` αποκλείει μια λανθασμένη υλοποίηση που επιστρέφει τη μικρότερη κατάλληλη τιμή. Το `[5]` αποκλείει λάθος αυστηρό `>` αντί για `>=`.

PYTHON

```
import unittest

def first_at_least(values, limit):
    for value in values:
        if value >= limit:
            return value
    return None

class FirstTests(unittest.TestCase):
    def test_contract(self):
        cases = [
            ([2, 8, 5], 5, 8),
            ([5], 5, 5),
            ([1, 2], 5, None),
            ([], 5, None),
        ]
        for values, limit, expected in cases:
            with self.subTest(values=values, limit=limit):
                self.assertEqual(
                    first_at_least(values, limit), expected
                )

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

16.3 · Error που πρέπει να συμβεί

Το επιτυχές test αποδεικνύει ότι ο έλεγχος δεν απορρίπτει τα πάντα. Το αρνητικό test αποδεικνύει ότι το μηδέν δεν περνά.

PYTHON

```
import unittest

def positive(value):
    if value <= 0:
        raise ValueError("must be positive")
    return value

class PositiveTests(unittest.TestCase):
    def test_rejects_zero(self):
        with self.assertRaises(ValueError):
            positive(0)

    def test_accepts_one(self):
        self.assertEqual(positive(1), 1)

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

Κεφάλαιο 17 · Async

17.1 · Δημιουργήθηκε ή εκτελέστηκε;

Το σώμα εκτελείται όταν περιμένουμε το coroutine. Η αναφορά στο coroutine διατηρείται και καταναλώνεται σωστά, οπότε δεν αφήνουμε unawaited coroutine.

PYTHON

```
import asyncio

async def work(events):
    events.append("ran")
    return 7

async def main():
    events = []
    pending = work(events)
    assert events == []
    assert await pending == 7
    assert events == ["ran"]

asyncio.run(main())
```

17.2 · Map με TaskGroup

Περιμένουμε να βγούμε από το group πριν ζητήσουμε τα task results. Η σειρά της λίστας tasks ακολουθεί την είσοδο.

PYTHON

```
import asyncio

async def double(n):
    await asyncio.sleep(0)
    return n * 2

async def double_all(values):
    async with asyncio.TaskGroup() as group:
        tasks = [group.create_task(double(n)) for n in values]
    return [task.result() for task in tasks]

assert asyncio.run(double_all([3, 1, 2])) == [6, 2, 4]
assert asyncio.run(double_all([])) == []
```

17.3 · Άθροισμα από async generator

Η generator function συνδυάζει async αναμονή και παραγωγή. Ο consumer αναλαμβάνει την τελική συσσώρευση.

PYTHON

```
import asyncio

async def numbers():
    for number in [1, 2, 3]:
        await asyncio.sleep(0)
        yield number

async def total():
    result = 0
    async for number in numbers():
        result += number
    return result

assert asyncio.run(total()) == 6
```

Κεφάλαιο 18 · Σύγχρονες εκδόσεις

18.1 · Αναγνώρισε το template

Η αλλαγή του ονόματος title μετά τη δημιουργία δεν αλλάζει την τιμή που αποθηκεύτηκε στο interpolation. Το αποτέλεσμα είναι Template, όχι str.

PYTHON

```
from string.templatelib import Template

title = "Python"
template = t"Book: {title}"
title = "Ruby"
assert isinstance(template, Template)
assert not isinstance(template, str)
assert template.strings == ("Book: ", "")
assert template.values == ("Python",)
```

18.2 · Flatten που δουλεύει στην 3.14

Το πρώτο for παίρνει ομάδα. Το δεύτερο παίρνει στοιχείο από αυτή την ομάδα. Οι κενές ομάδες απλώς δεν παράγουν στοιχεία.

PYTHON

```
def flatten(groups):
    return [item for group in groups for item in group]

assert flatten([[], [1], [2, 3]]) == [1, 2, 3]
assert flatten([]) == []
```

18.3 · Missing, None και μηδέν

Χρειάζεται ένα μοναδικό object που δημιουργείται μία φορά και χρησιμοποιείται ως το default. Ένα καινούργιο `object()` μέσα στη σύγκριση δεν θα είχε την ίδια ταυτότητα.

PYTHON

```
MISSING = object()

def pick(value=MISSING):
    return "missing" if value is MISSING else value

assert pick() == "missing"
assert pick(None) is None
assert pick(0) == 0
assert pick("") == ""
```

Κεφάλαιο 19 · Το project

19.1 · Φίλτρο μετά τη συγκέντρωση

Τα 25 και 20 λεπτά του ίδιου τίτλου έχουν ήδη γίνει 45. Αν φιλτράραμε συνεδρίες πριν αθροίσουμε, θα χάναμε τον τίτλο στο όριο 35.

PYTHON

```
def at_least(rows, minimum):
    return [(title, minutes) for title, minutes in rows
            if minutes >= minimum]

rows = [("Python basics", 45), ("Python, daily", 35),
        ("Ruby patterns", 15)]
assert at_least(rows, 35) == rows[:2]
assert at_least(rows, 100) == []
assert at_least([], 0) == []
```

19.2 · Ημέρες με ανάγνωση

Το set συγκεντρώνει ημέρες, όχι sessions. Το φίλτρο αποκλείει μόνο μηδενική διάρκεια, σύμφωνα με την απαίτηση. Το test χρησιμοποιεί το `readlog` του συνοδευτικού project.

PYTHON

```
from datetime import date
from readlog.core import Session

def active_days(sessions):
    return len({s.day for s in sessions if s.minutes > 0})

day = date(2026, 10, 16)
sessions = [
    Session("Python", 20, day),
    Session("Ruby", 10, day),
    Session("Go", 0, date(2026, 10, 17)),
]
assert active_days(sessions) == 1
assert active_days([]) == 0
```

19.3 · JSON αναφορά

Φτιάχνουμε πρώτα τη δομή και αφήνουμε τον serializer να χειριστεί quotes και escaping. Το round-trip test ελέγχει το νόημα της αναφοράς.

PYTHON

```
import json

def report_json(rows):
    records = [{"title": title, "minutes": minutes}
               for title, minutes in rows]
    return json.dumps(records, ensure_ascii=False)

result = report_json([("Σημειώσεις", 25), ("Ruby", 10)])
assert json.loads(result) == [
    {"title": "Σημειώσεις", "minutes": 25},
    {"title": "Ruby", "minutes": 10},
]
assert json.loads(report_json([])) == []
```

ΠΑΡΑΡΤΗΜΑ

Ruby → Python: γρήγορη αναφορά

Χρησιμοποίησε τον πίνακα για να ξαναβρείς κάτι που έχεις ήδη δουλέψει. Οι αντιστοιχίσεις περιγράφουν συνηθισμένες περιπτώσεις, όχι υπόσχεση ότι οι δύο γλώσσες έχουν ίδια semantics.

Τιμές και σύνταξη

Από τη Ruby	Στην Python	Τι να θυμάσαι
`nil`	`None`	Έλεγχος με `is None`
`true`, `false`	`True`, `False`	Το αρχικό κεφαλαίο μετρά
`elsif`	`elif`	Το block ορίζεται με indentation
`unless x`	`if not x`	Διαφορετικές falsey τιμές
`a ? b : c`	`b if a else c`	Το αποτέλεσμα γράφεται πρώτο
`&&`, `||`, `!`	`and`, `or`, `not`	Τα and/or επιστρέφουν operands
`obj.equal?(x)`	`obj is x`	Ταυτότητα, όχι ισότητα τιμών
`value.nil?`	`value is None`	Δεν απορρίπτει 0 ή κενά
Τελευταίο expression	Ρητό `return`	Χωρίς return παίρνεις None
`"#{name}"`	`f"{name}"`	Formatting με `:...`
`"12".to_i`	`int("12")`	Το κακό text προκαλεί error
`a / b` με integers	`/` ή `//`	Διάλεξε πραγματική ή floor διαίρεση
`0...n`	`range(n)`	Το τελικό όριο αποκλείεται

Collections και iteration

Από τη Ruby	Στην Python	Τι να θυμάσαι
`Array`	`list`	Mutable sequence
`Hash`	`dict`	Το missing `[]` σηκώνει KeyError
`array << x`	`items.append(x)`	Επιστρέφει None
`array.concat(xs)`	`items.extend(xs)`	Ξεδιπλώνει τα στοιχεία
`array.size`	`len(items)`	Function μέσω protocol
`array.empty?`	`not items`	Για γνωστό collection
`array.include?(x)`	`x in items`	Το κόστος εξαρτάται από τον τύπο
`array.uniq`	`list(dict.fromkeys(xs))`	Hashable στοιχεία, σειρά πρώτης εμφάνισης
`array.compact`	Φίλτρο `is not None`	Το filter(None, ...) αφαιρεί και 0
`array.join(",")`	`",".join(items)`	Το separator έχει τη method
`each_with_index`	`enumerate(items)`	Η θέση προηγείται της τιμής
`map`, `select`	Comprehensions	Ξεχώρισε παραγωγή από φίλτρο
`sort_by`	`sorted(..., key=...)`	Νέα λίστα, stable sort
`group_by`	`defaultdict(list)`	Όχι ίδιο με itertools.groupby
`each_slice(n)`	`itertools.batched`	Παράγει tuples, τελευταία ίσως μικρότερη
`Hash.new` με block	`defaultdict(factory)`	Factory στο missing `[]`, όχι στο get

Functions και objects

Από τη Ruby	Στην Python	Τι να θυμάσαι
`*args`, `**kwargs`	Ίδια σύμβολα	Tuple και dict αντίστοιχα
Keyword arguments	Keyword-only με `*`	Το όριο φαίνεται στη signature
`&block` / `Proc`	Callable argument	Περνάς function χωρίς να την καλέσεις
`yield` σε block	Κλήση callable	Το Python yield παράγει generator
`initialize`	`__init__`	Αρχικοποίηση, χωρίς επιστροφή instance
`@name`	`self.name`	Ρητό self μέσα στη method
`attr_accessor`	Κανονικά attributes	Property όταν χρειάζεται υπολογισμός
`def self.build`	`@classmethod`	Η method παίρνει cls
`to_s`, `inspect`	`__str__`, `__repr__`	Για χρήστη και για διάγνωση
`Struct` / `Data`	`dataclass`	Αντίστοιχη ανάγκη, όχι ίδια semantics
`rescue`	`except`	Πιάσε συγκεκριμένο exception
`ensure`	`finally`	Cleanup, χωρίς return που κρύβει error
`File.open` με block	`with ... as ...`	Lifetime του resource
`require`	`import`	Module namespace και cache

Πέντε ερωτήσεις όταν κάτι φαίνεται περίεργο

1. Έχω τιμή `None` ή απλώς falsey τιμή; Έλεγξε `repr(value)` και `type(value)`.
2. Άλλαξα ένα object ή έδεσα ξανά ένα όνομα; Έλεγξε ποιοι callers κρατούν το ίδιο reference.
3. Έχω collection ή iterator που ήδη καταναλώθηκε; Δες πού έγινε το πρώτο iteration.
4. Το error εμφανίζεται στην κλήση ή στην κατανάλωση; Generators και async μεταθέτουν εκτέλεση.
5. Ποιος interpreter και ποιο module τρέχουν; Έλεγξε `sys.executable` και `module.__file__` πριν αλλάξεις τον κώδικα.

Έχεις φτάσει σε intermediate συντακτικό;

Η ανάγνωση δεν είναι από μόνη της απόδειξη δεξιότητας. Χρησιμοποίησε τα παρακάτω ως πρακτικό έλεγχο, χωρίς να απαιτείς αποστήθιση της standard library.

Μπορείς να εξηγήσεις γιατί ένα mutable default κρατά αλλαγές ανάμεσα σε κλήσεις και να το διορθώσεις χωρίς να αλλάξεις το contract. Μπορείς να γράψεις μια signature με positional-only και keyword-only arguments και να προβλέψεις ποιες κλήσεις είναι έγκυρες. Μπορείς να διαλέξεις comprehension, loop ή generator ανάλογα με το πώς θα χρησιμοποιήσεις τα δεδομένα.

Μπορείς να φτιάξεις dataclass με ανεξάρτητα mutable fields, να διαβάσεις ένα Protocol και να ξεχωρίσεις type hint από πραγματικό validation. Μπορείς να εξηγήσεις τι κάνει ένας decorator ως κανονική function call και να αναγνωρίσεις capture pattern που δεν συγκρίνει με constant.

Μπορείς να χωρίσεις ένα πρόγραμμα σε importable core και CLI, να πιάσεις το συγκεκριμένο error που περιμένεις και να κλείσεις σωστά τα resources. Μπορείς να εξηγήσεις γιατί δύο διαδοχικά awaits δεν εκτελούνται αυτόματα concurrently.

Αν μπορείς να ξαναγράψεις το readlog από το contract, να ολοκληρώσεις τις τρεις επεκτάσεις και να ελέγξεις τα edge cases χωρίς να αντιγράψεις τη λύση, έχεις καλύψει τον στόχο αυτού του βιβλίου. Όταν κολλάς, επέστρεψε στο συγκεκριμένο κεφάλαιο. Δεν χρειάζεται να ξαναδιαβάσεις τα πάντα από την αρχή.

ΠΑΡΑΡΤΗΜΑ

Πηγές και επόμενη ανάγνωση

Οι πηγές είναι η επίσημη τεκμηρίωση και τα σχετικά PEPs. Τα παραδείγματα και οι ασκήσεις του βιβλίου έχουν γραφτεί για τη συγκεκριμένη διαδρομή Ruby → Python. Οι σύνδεσμοι ελέγχθηκαν στις 16 Οκτωβρίου 2026. Τα PEPs εξηγούν τη σχεδίαση, αλλά για την τελική συμπεριφορά μιας έκδοσης προτίμησε την τεκμηρίωσή της.

Εκδόσεις

- Python downloads και ενεργές εκδόσεις: επιβεβαίωση της σταθερής σειράς 3.15 και της υποστήριξης των παλαιότερων σειρών.
- What's new in Python 3.14: t-strings, annotations και αλλαγές στον interpreter.
- What's new in Python 3.15: νέες δυνατότητες και οδηγίες μετάβασης.
- PEP 790: χρονοδιάγραμμα της σειράς 3.15.

Κεφάλαια 1-8: ο πυρήνας της γλώσσας

- Control flow και function definitions: loops, arguments, annotations και pattern matching.
- Built-in types: truthiness, αριθμοί, strings, sequences, mappings και sets.
- Data structures: list methods, comprehensions και iteration.
- Collections: Counter και defaultdict.
- Itertools: islice, pairwise, batched και groupby.
- Data model: objects, protocols και special methods.

Κεφάλαια 9-14: οργάνωση και intermediate σύνταξη

- Errors and exceptions: exception handling, chaining και cleanup.
- Contextlib: generator-based context managers.
- Modules: namespaces, imports και package execution.
- Classes: scopes, instances, inheritance και iteration.
- Dataclasses: fields, defaults, frozen instances και replace.
- Typing: unions, TypedDict, Protocol, generics και runtime όρια.
- Functools: wraps, partial και caches.

Κεφάλαια 15-19: δεδομένα, tests και νέα features

- Pathlib, CSV και JSON: file APIs και formats του project.
- Unittest: test cases, discovery, subtests και exception assertions.
- Asyncio tasks: coroutines, TaskGroup, cancellation και to_thread.
- Argparse: arguments και help του CLI.

- String templatelib: Template και Interpolation.
- Annotationlib: lazy evaluation των annotations.
- PEP 798: unpacking μέσα σε comprehensions.
- PEP 661: sentinel values.
- Built-in functions 3.15 και built-in types 3.15: νέοι built-in τύποι.
- PEP 686: UTF-8 mode ως default.

Περιβάλλον και Ruby αντιστοιχίσεις

- Python Packaging: installing packages και pyproject.toml: περιβάλλοντα, εγκατάσταση και project metadata.
- Uv και pytest: προαιρετικά εργαλεία για την επόμενη φάση. Δεν απαιτούνται για τον κώδικα του βιβλίου.
- Ruby methods και Ruby Enumerable: οι αντίστοιχες Ruby έννοιες που χρησιμοποιούνται ως αφετηρία.

Η σωστή επόμενη κίνηση είναι ένα μικρό δικό σου εργαλείο: parsing ενός export, οργάνωση ενός συνόλου αρχείων ή μεταφορά ενός Ruby script που ήδη καταλαβαίνεις. Κράτησε το input μικρό και γνωστό. Γράψε πρώτα το contract, μετά τον πιο απλό κώδικα που το ικανοποιεί και τέλος τα tests που θα εντόπιζαν μια πειστική αλλά λάθος υλοποίηση.