

# Ανάλυση δεδομένων με τη γλώσσα προγραμματισμού Python ΕΚΔΔΑ

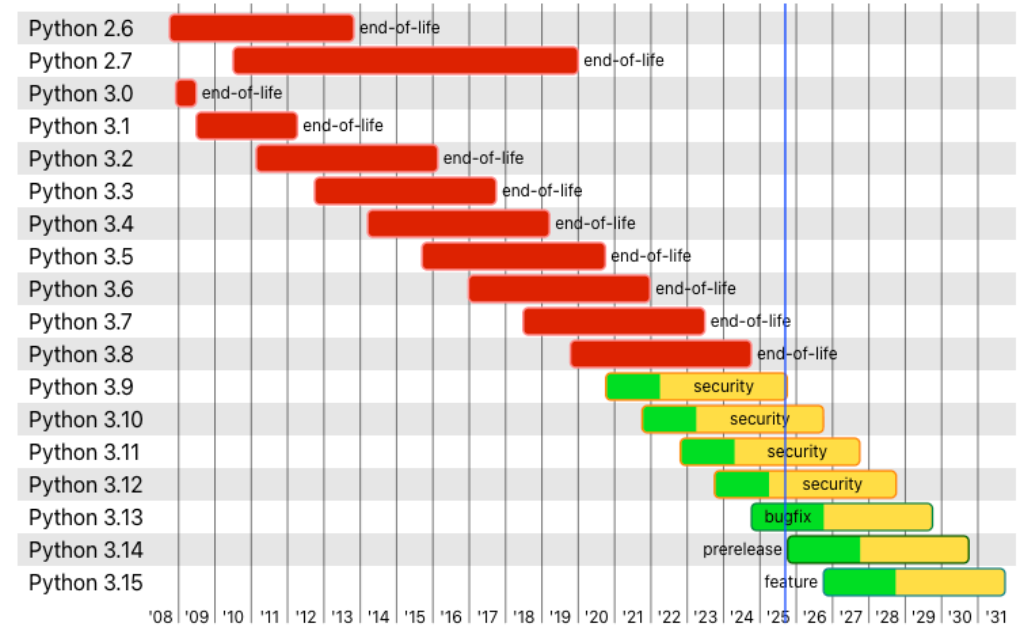
1.2 Εισαγωγή στην Python (α' μέρος)

Εβδομάδα 1/9

Προετοιμασία διαφανειών: Γκόγκος Χρήστος

# Τι είναι η Python;

- Η Python είναι μια γλώσσα προγραμματισμού, δηλαδή μπορεί να χρησιμοποιηθεί για τη συγγραφή προγραμμάτων με τρόπο που είναι «κατανοητός» από ανθρώπους και που μπορούν να εκτελεστούν στη συνέχεια από ένα υπολογιστικό σύστημα
- Η Python δημιουργήθηκε το 1990 από τον [Guido Van Rossum](#) ως μια γλώσσα προγραμματισμού υψηλού επιπέδου, γενικού σκοπού δίνοντας έμφαση στη γρήγορη ανάπτυξη κώδικα και στην υψηλή αναγνωσιμότητα



<https://devguide.python.org/versions/>

# Πόσο δημοφιλής είναι η γλώσσα Python;

Αύγουστος 2025

Aug 2025	Aug 2024	Change	Programming Language		Ratings	Change
1	1			Python	26.14%	+8.10%
2	2			C++	9.18%	-0.86%
3	3			C	9.03%	-0.15%
4	4			Java	8.59%	-0.58%
5	5			C#	5.52%	-0.87%
6	6			JavaScript	3.15%	-0.76%
7	8	^		Visual Basic	2.33%	+0.15%
8	9	^		Go	2.11%	+0.08%
9	25	^^		Perl	2.08%	+1.17%
10	12	^		Delphi/Object Pascal	1.82%	+0.19%

<https://www.tiobe.com/tiobe-index/>

# Γιατί η Python είναι τόσο δημοφιλής;

- Είναι δωρεάν και εύκολα διαθέσιμη σε όλους (π.χ., Windows, MacOS, Linux, κ.λπ.)
- Διαθέτει διερμηνευτή εντολών και σημειωματάρια (jupyter notebooks) για άμεση εκτέλεση τμημάτων κώδικα
- Αρκετή λειτουργικότητα περιλαμβάνεται στην ίδια την γλώσσα - batteries included!
- Είναι ανοικτού κώδικα (open source)
- Διαθέτει δωρεάν μεγάλο πλήθος βιβλιοθηκών ( π.χ., numpy, pandas, scikit-learn, pytorch κ.α.)
- Το προγράμματα σε Python είναι σύντομα και εύκολα αναγνώσιμα, «εκτελέσιμος ψευδοκώδικας»

# Ο κώδικας Python ως ψευδοκώδικας

## Ψευδοκώδικας

```
PROMPT for ph_value  
GET the ph_value and make it a number  
IF ph_value is greater than or equal to 7 THEN  
    DISPLAY "Neutral or alkaline"  
ELSE  
    DISPLAY "Acidic"  
ENDIF
```

## Python

```
ph_value = float(input("Enter pH value:"))  
if ph_value >= 7:  
    print("Neutral or alkaline")  
else:  
    print("Acidic")
```

<https://interactivetextbooks.tudelft.nl/programming-foundations/content/chapter6/pseudocode.html>

# To Zen της Python (1/2)

- Το Zen της Python είναι ένα σύνολο 19 βασικών αρχών για τη συγγραφή κομψού, αναγνώσιμου και λειτουργικού κώδικα Python
- Δημιουργήθηκε από τον [Tim Peters](#) και έχει αποτελέσει το λεγόμενο PEP 20
- PEP (Python Enhancement Proposal) είναι ένα έγγραφο που είτε παρέχει κάποια χρήσιμη πληροφορία για την κοινότητα της Python είτε περιγράφει ένα νέο χαρακτηριστικό της Python και του περιβάλλοντός της
- Δείτε σχετικά και το PEP 8 - <https://peps.python.org/pep-0008/>

# To Zen της Python (2/2)

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.  
Errors should never pass silently.  
Unless explicitly silenced.  
In the face of ambiguity, refuse the temptation to guess.  
There should be one-- and preferably only one --obvious way to do it.  
Although that way may not be obvious at first unless you're Dutch.  
Now is better than never.  
Although never is often better than *right* now.  
If the implementation is hard to explain, it's a bad idea.  
If the implementation is easy to explain, it may be a good idea.  
Namespaces are one honking great idea -- let's do more of those!
```

1. Το όμορφο είναι καλύτερο από το άσχημο.
2. Το ρητό είναι καλύτερο από το υπονοούμενο.
3. Το απλό είναι καλύτερο από το σύνθετο.
4. Το σύνθετο είναι καλύτερο από το περίπλοκο.
5. Το επίπεδο είναι καλύτερο από το ένθετο.
6. Το αραιό είναι καλύτερο από το πυκνό.
7. Η αναγνωσιμότητα μετράει.
8. Οι ειδικές περιπτώσεις δεν είναι αρκετά ειδικές για να παραβιάσουν τους κανόνες.
9. Ωστόσο η πρακτικότητα νικά την απόλυτη καθαρότητα.
10. Τα λάθη δεν πρέπει ποτέ να περνούν σιωπηλά.
11. Εκτός αν σιωπηθούν ρητά.
12. Μπροστά στην αμφιβολία, αρνηθείτε τον πειρασμό να μαντέψετε.
13. Θα πρέπει να υπάρχει ένας – και κατά προτίμηση μόνο ένας – προφανής τρόπος για να το κάνετε.
14. Αν και αυτός ο τρόπος μπορεί να μην είναι προφανής στην αρχή, εκτός αν είστε Ολλανδός.
15. Το τώρα είναι καλύτερο από το ποτέ.
16. Αν και το ποτέ είναι συχνά καλύτερο από το **αμέσως** τώρα.
17. Αν η υλοποίηση είναι δύσκολο να εξηγηθεί, είναι κακή ιδέα.
18. Αν η υλοποίηση είναι εύκολο να εξηγηθεί, μπορεί να είναι καλή ιδέα.
19. Οι χώροι ονομάτων είναι μια ηχηρά καλή ιδέα – ας κάνουμε και άλλες τέτοιες!

# Pythonic (πυθωνικός) κώδικας

- Ο πυθωνικός κώδικας χρησιμοποιεί ιδιώματα της γλώσσας Python και είναι εύκολα αναγνώσιμος
- Ιδίωμα μιας γλώσσας είναι ένας κοινά αποδεκτός και ενδεδειγμένος τρόπος επίλυσης ενός προβλήματος που χρησιμοποιούν οι έμπειροι προγραμματιστές

## Παράδειγμα 2 τρόπων αντιμετάθεσης 2 μεταβλητών

```
a, b = 7, 42  
print(a, b) # Έξοδος: 7 42
```

```
# Μη πυθωνικός κώδικας αντιμετάθεσης 2 μεταβλητών  
temp = a  
a = b  
b = temp  
print(a, b) # Έξοδος: 42 7
```

```
#####
```

```
a, b = 7, 42  
print(a, b) # Έξοδος: 7 42
```

```
# Πυθωνικός κώδικας αντιμετάθεσης 2 μεταβλητών  
a, b = b, a  
print(a, b) # Έξοδος: 7 42
```



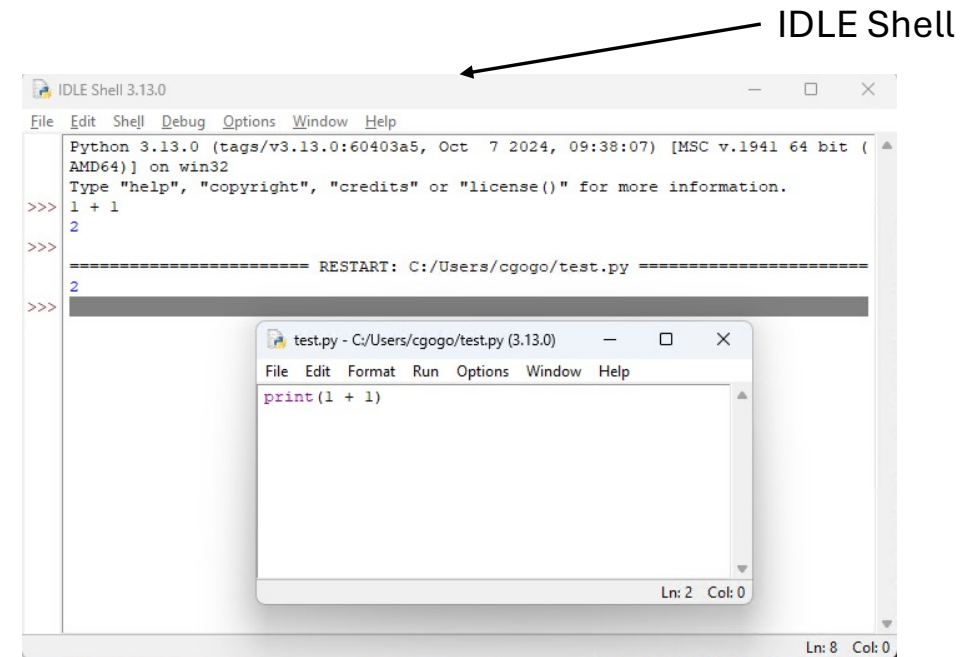
# Εγκατάσταση της Python

- Επίσκεψη στο <https://www.python.org/>, μεταφόρτωση και εγκατάσταση μιας πρόσφατης έκδοσης της Python 3.major\_version.minor\_version που είναι διαθέσιμη, π.χ. Python 3.13.7 (για Σεπτέμβριο 2025)
- Στα Windows η εγκατάσταση γίνεται εύκολα κατεβάζοντας και εκτελώντας έναν “Windows installer” ανάλογα με τον τύπο υπολογιστή που διαθέτετε (π.χ., 64-bit για επεξεργαστές Intel ή ARM64 για επεξεργαστές ARM)
- Υπάρχουν και άλλοι περισσότερο προχωρημένοι τρόποι εγκατάστασης της Python (π.χ., uv, conda, poetry)

Δείτε για την εγκατάσταση της Python στα Windows  
[https://www.youtube.com/watch?v=Pi\\_SMAktdXk](https://www.youtube.com/watch?v=Pi_SMAktdXk)  
από το 11:33 μέχρι το 16:00

# Μετά την εγκατάσταση της Python

- Μετά την εγκατάσταση της Python μπορούν να χρησιμοποιηθούν:
  - το **Python IDLE shell** που επιτρέπει τη αλληλοεπιδραστική εκτέλεση εντολών Python, την εμφάνιση αποτελεσμάτων, αλλά και τη συγγραφή εκτέλεση και αποσφαλμάτωση προγραμμάτων σε Python (στα Windows δημιουργείται με την εγκατάσταση της Python εικονίδιο σχετικής εφαρμογής)
  - Το **REPL (Read Evaluate Print Loop)** της Python που μπορεί να κληθεί από τη γραμμή εντολών πληκτρολογώντας την εντολή `python` (ή `python3` ή `py` ανάλογα με την εγκατάσταση)
- Ωστόσο, υπάρχουν και άλλοι τρόποι συγγραφής και εκτέλεσης κώδικα σε Python που μπορεί να είναι βολικότεροι για συγγραφή και εκτέλεση αποσπασμάτων κώδικα ή μεγαλύτερων εφαρμογών



REPL

```
C:\Users\cgogo>python
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07)
[MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more i
nformation.
>>> 1 + 1
2
>>>
```

# REPL (1/2) – Τι είναι;

- Εφόσον έχει εγκατασταθεί η Python μπορεί να χρησιμοποιηθεί το REPL
- Ενεργοποιείται απλά γράφοντας `python` στη γραμμή εντολών
- REPL σημαίνει:
  - **R**ead – λήψη εισόδου από τον χρήστη (`python` κώδικας)
  - **E**val – αποτίμηση κώδικα
  - **P**rint – εκτύπωση αποτελεσμάτων
  - **L**oop – επανάληψη της διαδικασίας μέχρι να γίνει έξοδος από τον χρήστη
- Χρήσιμο για:
  - γρήγορο πειραματισμό με κώδικα
  - αποσφαλμάτωση κώδικα
  - δοκιμή συναρτήσεων βιβλιοθηκών
  - κλήση συστήματος βοήθειας, π.χ. `>>> help(str)`

# REPL (2/2) – Παραδείγματα κώδικα Python

- Άμεση εκτέλεση κώδικα
- Χωρίς ανάγκη να γραφεί πλήρες πρόγραμμα (script)
- Πλοήγηση στο ιστορικό των εντολών με τα βελάκια πάνω και κάτω
- Επιτρέπεται η εισαγωγή εντολών πολλαπλών γραμμών
- Έξοδος με `exit()` ή με `Ctrl+D`
- Από την Python 3.13 το REPL διαθέτει **syntax highlighting**

```
>>> 2 + 3**2
11
>>> "hello".upper()
'HELLO'
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2025, 9, 14, 8, 48, 13, 697875)
```

# Άσκηση #1 (REPL): Εκφώνηση

- Υπολογίστε τις ακόλουθες εκφράσεις στο REPL και παρατηρήστε τα αποτελέσματα που λαμβάνετε:

```
>>> 2 + 3 * 4
>>> "Python"[:-1]
>>> [x**2 for x in range(6)]
>>> import random; random.randint(1,6)
>>> random.choice(["πέτρα", "ψαλίδι", "χαρτί"])
>>> sorted([7, 2, 3, 1, 6])
```

- Δοκιμάστε στο REPL τις ακόλουθες 4 εντολές μια προς μια:

```
>>> import this
>>> import __hello__; __hello__.main()
>>> from __future__ import braces
>>> import antigravity
```

# Άσκηση #1: Αποτελέσματα εντολών στο REPL

```
$ python
Python 3.13.7 | packaged by conda-forge | (main, Sep 3 2025, 14:24:46) [Clang 19.1.7 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3 * 4
14
>>> "Python"[::-1]
'nohtyP'
>>> [x**2 for x in range(6)]
[0, 1, 4, 9, 16, 25]
>>> import random; random.randint(1,6)
4
>>> import random; random.randint(1,6)
3
>>> random.choice(["πέτρα", "ψαλίδι", "χαρτί"])
'χαρτί'
>>> random.choice(["πέτρα", "ψαλίδι", "χαρτί"])
'χαρτί'
>>> sorted([7, 2, 3, 1, 6])
[1, 2, 3, 6, 7]
```

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the
```

```
>>> import __hello__
>>> __hello__.main()
Hello world!
>>> from __future__ import braces
File "<python-input-10>", line 1
    from __future__ import braces
                                ^^^^^^
SyntaxError: not a chance
>>> import antigravity
```

<https://xkcd.com/353/>

# Μεταβλητές (1/2)

- Στην Python οι μεταβλητές (variables) είναι ονοματισμένες αναφορές προς αντικείμενα που βρίσκονται στη μνήμη του υπολογιστή
- Λόγω του dynamic typing, δεν δηλώνονται τύποι δεδομένων, ενώ ο τύπος των μεταβλητών μπορεί να αλλάζει καθώς εκτελείται ο κώδικας
- Η ανάθεση τιμών σε μεταβλητές γίνεται με τον τελεστή =, ο δε τύπος της μεταβλητής προκύπτει από τον τύπο του αντικειμένου στο δεξί μέρος της ανάθεσης
- Οι μεταβλητές έχουν όνομα (name), τύπο (type) και αναγνωριστικό (id):
  - Με τη χρήση της συνάρτησης type() βλέπουμε τον τύπο μιας μεταβλητής
  - Με τη χρήση της συνάρτησης id() βλέπουμε το αναγνωριστικό «θέσης» μνήμης μιας μεταβλητής

```
>>> x = 42
>>> type(x)
<class 'int'>
>>> id(x)
4306464400
>>> x = "python"
>>> type(x)
<class 'str'>
>>> id(x)
4310246208
```

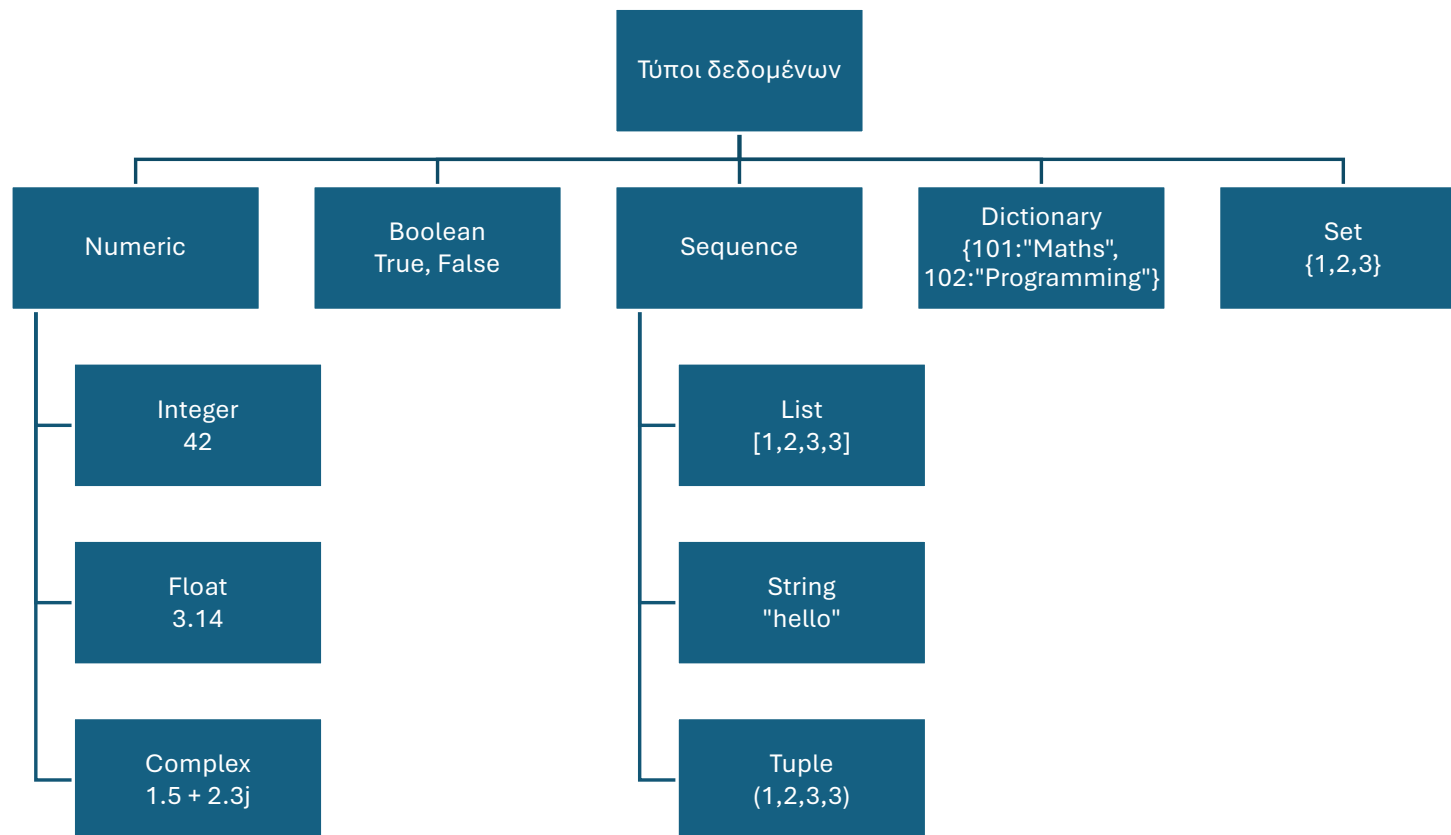
## Μεταβλητές (2/2)

- Τα ονόματα των μεταβλητών ξεκινούν με γράμμα ή με κάτω παύλα \_ (underscore) και μπορούν να περιέχουν γράμματα ψηφία και την κάτω παύλα
- Στις μεταβλητές, αλλά και γενικότερα, υπάρχει διάκριση πεζών κεφαλαίων
- Μπορούν να χρησιμοποιηθούν ελληνικοί χαρακτήρες στα ονόματα των μεταβλητών, αλλά αυτό δεν συνίσταται

```
>>> a_valid_variable_name = 42
>>> a_valid_variable_name
42
>>> ένα_έγκυρο_όνομα_μεταβλητής = 7
>>> ένα_έγκυρο_όνομα_μεταβλητής
7
>>> x = 1729
>>> X
Traceback (most recent call last):
  File "<python-input-37>", line 1, in <module>
    X
NameError: name 'X' is not defined. Did you mean: 'x'?
```



# Τύποι δεδομένων της Python



# Αριθμητικοί τελεστές

Σύμβολο	Πράξη	Παράδειγμα στο REPL
+	Πρόσθεση	>>> 1 + 2 3
-	Αφαίρεση	>>> 1 - 2 -1
*	Πολλαπλασιασμός	>>> 2 * 3 6
/	Διαίρεση	>>> 10 / 4 2.5
//	Πηλίκo ακέραιας διαίρεσης	>>> 10 // 3 3
%	Υπόλοιπο ακέραιας διαίρεσης	>>> 10 % 3 1
**	Ύψωση σε δύναμη	>>> 2 ** 10 1024

# Προτεραιότητα αριθμητικών τελεστών

- Ισχύει ότι ισχύει στα μαθηματικά για τις αριθμητικές πράξεις, δηλαδή, πρώτα παρενθέσεις, μετά ύψωση σε δύναμη, μετά πολλαπλασιασμός και διαίρεση και τέλος πρόσθεση και αφαίρεση
- Σε πράξεις με τελεστές ίδιας προτεραιότητας, στην πρόσθεση/αφαίρεση, καθώς και στο πολλαπλασιασμό/διαίρεση οι πράξεις γίνονται από αριστερά προς τα δεξιά, ενώ στην ύψωση σε δύναμη οι πράξεις γίνονται από δεξιά προς τα αριστερά

```
>>> 10/4/2
1.25
>>> (10/4)/2
1.25
>>> 10/(4/2)
5.0
>>> 2**1**2
2
>>> 2**(1**2)
2
>>> (2**1)**2
4
```

# Συγκριτικοί τελεστές

Συγκριτικός τελεστής	Ερμηνεία
<code>a == b</code>	Έλεγχος αν οι τιμές των μεταβλητών <code>a</code> και <code>b</code> είναι ίσες
<code>a != b</code>	Έλεγχος αν οι τιμές των μεταβλητών <code>a</code> και <code>b</code> είναι διαφορετικές
<code>a &lt; b</code>	Έλεγχος αν η τιμή της <code>a</code> είναι μικρότερη από την τιμή της <code>b</code>
<code>a &gt; b</code>	Έλεγχος αν η τιμή της <code>a</code> είναι μεγαλύτερη από την τιμή της <code>b</code>
<code>a &lt;= b</code>	Έλεγχος αν η τιμή της <code>a</code> είναι μικρότερη ή ίση από την τιμή της <code>b</code>
<code>a &gt;= b</code>	Έλεγχος αν η τιμή της <code>a</code> είναι μεγαλύτερη ή ίση από την τιμή της <code>b</code>
<code>a is b</code>	Έλεγχος αν η <code>a</code> και η <code>b</code> αναφέρονται στο ίδιο αντικείμενο
<code>a in b</code>	Έλεγχος αν η <code>a</code> υπάρχει στη <code>b</code> (π.χ. <code>'a' in 'banana'</code> επιστρέφει <code>True</code> )

# Λογικοί τελεστές (boolean)

Λογικός τελεστής	Ερμηνεία	Πίνακας αληθείας
and	λογικό ΚΑΙ (σύζευξη)	False and False $\rightarrow$ False False and True $\rightarrow$ False True and False $\rightarrow$ False True and True $\rightarrow$ True
or	λογικό Ή (διάζευξη)	False or False $\rightarrow$ False False or True $\rightarrow$ True True or False $\rightarrow$ True True or True $\rightarrow$ True
not	λογική άρνηση	not False $\rightarrow$ True not True $\rightarrow$ False

# Τετραγωνική ρίζα, κυβική ρίζα κ.ο.κ.

- Για να υπολογιστεί η τετραγωνική ρίζα ενός αριθμού γνωρίζουμε ότι ο αριθμός θα πρέπει να υψωθεί στη δύναμη  $1/2$
- Ομοίως για την κυβική ρίζα υψώνεται στη δύναμη  $1/3$  κ.ο.κ. για ρίζες υψηλότερης τάξης

```
>>> 2**0.5
1.4142135623730951
>>> 2**(1/2)
1.4142135623730951
>>> 2**(1/3)
1.2599210498948732
>>> 2**(1/4)
1.189207115002721
>>> 2**(1/5)
1.148698354997035
>>> -2**0.5
-1.4142135623730951
>>> (-2)**0.5
(8.659560562354934e-17+1.4142135623730951j)
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.sqrt(-2)
Traceback (most recent call last):
  File "<python-input-82>", line 1, in <module>
    math.sqrt(-2)
    ~~~~~^~~~~
ValueError: math domain error
>>> import cmath
>>> cmath.sqrt(-2)
1.4142135623730951j
```

# Αυτόματη μετατροπή τύπων

- Ο τύπος δεδομένων μιας μεταβλητής μπορεί να αλλάζει αυτόματα εφόσον αυτό υποδεικνύεται από τις πράξεις που γίνονται
- Στο παράδειγμα η μεταβλητή  $x$  σταδιακά αποκτά τύπο δεδομένων:  
ακέραιο  $\rightarrow$  πραγματικό  $\rightarrow$  μιγαδικό

```
>>> x = 2
>>> type(x)
<class 'int'>
>>> x = x / 3
>>> x
0.6666666666666666
>>> type(x)
<class 'float'>
>>> x = -x
>>> x = x**0.5
>>> x
(4.9995996217394874e-17+0.816496580927726j)
>>> type(x)
<class 'complex'>
```

# Εξαναγκασμένη αλλαγή τύπου μεταβλητής

- Υπάρχουν περιπτώσεις που μπορεί να είναι επιθυμητή η αλλαγή τύπου μιας μεταβλητής (π.χ. από ακέραιο σε χαρακτήρες, από πραγματικό σε ακέραιο κ.α.)
- Αυτό ονομάζεται type casting και γίνεται όπως φαίνεται στα παραδείγματα δεξιά

```
>>> x = 10
>>> type(x)
<class 'int'>
>>> x = str(x)
>>> type(x)
<class 'str'>
>>> y = 10/3
>>> type(y)
<class 'float'>
>>> y = int(y)
>>> type(y)
<class 'int'>
```



# Είσοδος τιμών από τον χρήστη (1/2)

- Με την ενσωματωμένη συνάρτηση `input()` ζητείται από τον χρήστη να εισάγει δεδομένα από το πληκτρολόγιο
- Τα δεδομένα επιστρέφονται από την `input()` ως λεκτικά, οπότε αν χρειάζεται να χρησιμοποιηθούν στη συνέχεια ως ακέραιες τιμές ή πραγματικές τιμές θα πρέπει να γίνει η κατάλληλη μετατροπή τύπων

```
>>> a = input()
42
>>> a, type(a)
('42', <class 'str'>)
>>> a = int(a)
>>> a, type(a)
(42, <class 'int'>)
>>> a
42
>>> a = float(a)
>>> a, type(a)
(42.0, <class 'float'>)
>>> a
42.0
```

## Είσοδος τιμών από τον χρήστη (2/2)

- Η `input()` μπορεί να δέχεται ως όρισμα ένα λεκτικό που χρησιμοποιείται ως μήνυμα προς τον χρήστη
- Με αυτό τον τρόπο μπορεί να γραφεί κώδικας που καθοδηγεί τον χρήστη για τις τιμές που πρέπει να εισάγει

```
>>> x = float(input("Δώσε μια πραγματική τιμή: "))  
Δώσε μια πραγματική τιμή: 3.14159  
>>> x, type(x)  
(3.14159, <class 'float'>)
```

# Εμφάνιση αποτελεσμάτων με την print() (1/2)

- Η ενσωματωμένη συνάρτηση print() χρησιμοποιείται για εμφάνιση τιμών και μηνυμάτων στην οθόνη σε μια γραμμή
- Η print() μπορεί να δέχεται πολλαπλά ορίσματα που εμφανίζονται το ένα δίπλα στο άλλο με ένα κενό χαρακτήρα ως διαχωριστικό
- Μπορούν να χρησιμοποιηθούν οι παράμετροι sep και end για να τροποποιήσουν τη συμπεριφορά της print()

```
>>> a, b = 1, 2
>>> print(a, "message", a + b)
1 message 3
>>> print(a, "message", a + b, sep="-")
1-message-3
>>> print(a, "message", a + b, end="")
1 message 3>>> █
```

# Εμφάνιση αποτελεσμάτων με την print() (2/2)

- Είναι συνηθισμένο να προετοιμάζεται μια συμβολοσειρά εξόδου με κατάλληλη μορφοποίηση της ώστε να περιέχει τιμές μεταβλητών και εκφράσεων σε προκαθορισμένες θέσεις πριν εμφανιστεί στην οθόνη με την print()
- Η μορφοποίηση μπορεί να γίνει:
  - Με το σύμβολο %
  - Με τη μέθοδο των συμβολοσειρών .format()
  - Με f-strings (νεότερος και προτιμότερος τρόπος), δείτε το <https://docs.python.org/3/tutorial/inputoutput.html>

```
>>> name, age = "Μαρία", 31
>>> s1 = "Το όνομα είναι %s και η ηλικία είναι %s" % (name, age)
>>> s1
'Tο όνομα είναι Μαρία και η ηλικία είναι 31'
>>> s2 = "Το όνομα είναι {} και η ηλικία είναι {}".format(name, age)
>>> s2
'Tο όνομα είναι Μαρία και η ηλικία είναι 31'
>>> s3 = f"Το όνομα είναι {name} και η ηλικία είναι {age}"
>>> s3
'Tο όνομα είναι Μαρία και η ηλικία είναι 31'
```

# Σχόλια

- Τα σχόλια (comments) στον κώδικα, αγνοούνται από τον διερμηνευτή και χρησιμοποιούνται για να παρέχουν εξήγηση σε σημεία του κώδικα που ο προγραμματιστής κρίνει ότι απαιτείται
- Η έναρξη σχολίου στην Python γίνεται με το σύμβολο #
- Μια καλή πρακτική είναι το σχόλιο να μην εξηγεί το τι κάνει ο κώδικας αλλά γιατί το κάνει
- Επίσης, τα σχόλια θα πρέπει να είναι καθαρά διατυπωμένα, ενημερωμένα και σύντομα

# Συμβολοσειρές

- Μια ακολουθία χαρακτήρων που οριοθετείται (αρχή και τέλος) με ειδικά σύμβολα ονομάζεται συμβολοσειρά
- Στην Python για την αρχή και το τέλος της συμβολοσειράς μπορούν να χρησιμοποιηθούν, μονά εισαγωγικά (π.χ. 'test'), διπλά εισαγωγικά (π.χ. "test"), τρία μονά εισαγωγικά (π.χ. '''test'''), και τρία διπλά εισαγωγικά (π.χ. """test""")
- Τα τριπλά εισαγωγικά χρησιμοποιούνται για συμβολοσειρές που καταλαμβάνουν πολλές γραμμές
- Μπορεί να χρησιμοποιηθεί συνδυασμός από μονά και διπλά εισαγωγικά για να υπάρχουν μέσα στη συμβολοσειρά είτε μονά είτε διπλά εισαγωγικά
- Οι συμβολοσειρές συχνά αναφέρονται ως αλφαριθμητικά ή λεκτικά

```
>>> s1 = 'this is a test'
>>> print(s1)
this is a test
>>> s2 = "this is a test"
>>> print(s2)
this is a test
>>> s1 == s2
True
>>> s3 = '''this is
... a test'''
>>> print(s3)
this is
a test
>>> s4 = """this is
... a test"""
>>> print(s4)
this is
a test
>>> s5 = 'this is a "test"'
>>> print(s5)
this is a "test"
```

# Δεικτοδότηση συμβολοσειρών

- Οι χαρακτήρες που περιέχονται σε μια συμβολοσειρά έχουν δείκτες με τους οποίους μπορούν να αναφερθούν
- Η δεικτοδότηση (indexing) ξεκινά για τον πρώτο χαρακτήρα από τα αριστερά από το 0 και αυξάνεται κατά ένα για κάθε χαρακτήρα προς τα δεξιά
- Υπάρχει και εναλλακτική δεικτοδότηση από το τέλος προς την αρχή με αρνητικούς δείκτες, ξεκινά με τον πλέον δεξιό χαρακτήρα που έχει δείκτη -1 και ο δείκτης μειώνεται κατά ένα για κάθε χαρακτήρα προς τα αριστερά

'this is a test'

<b>t</b>	<b>h</b>	<b>i</b>	<b>s</b>		<b>i</b>	<b>s</b>		<b>a</b>		<b>t</b>	<b>e</b>	<b>s</b>	<b>t</b>
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'this is a test'
>>> s[0], s[5], s[8], s[13]
('t', 'i', 'a', 't')
>>> s[-1], s[-14]
('t', 't')
```

# Τεμαχισμός συμβολοσειρών

- Ο τεμαχισμός (slicing) συμβολοσειρών επιστρέφει ένα τμήμα μιας συμβολοσειράς με βάση το μοτίβο [**<αρχή>:<τέλος>:<βήμα>**]
  - Η <αρχή>, <τέλος>, <βήμα> πρέπει να είναι ακέραιοι
  - Αν παραλείπεται η <αρχή> υπονοείται η αρχή της συμβολοσειράς
  - Αν παραλείπεται το <τέλος> υπονοείται το τέλος της συμβολοσειράς
  - Αν παραλείπεται το <βήμα> υπονοείται η τιμή 1
- Προσοχή! το τμήμα s[start:end], ξεκινά από το start αλλά δεν περιέχει το χαρακτήρα στη θέση end, σταματά στο end -1, δηλαδή είναι συμπεριληπτικό (inclusive) του start, αλλά όχι του end (exclusive)

s = 'this is a test'

<b>t</b>	<b>h</b>	<b>i</b>	<b>s</b>		<b>i</b>	<b>s</b>		<b>a</b>		<b>t</b>	<b>e</b>	<b>s</b>	<b>t</b>
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Τεμαχισμός	Ερμηνεία	παράδειγμα
s[start:end:step]	Τμήμα της s από τον δείκτη start μέχρι και τον δείκτη end-1	s[5:7] → 'is'
s[start:]	Τμήμα της s από τον δείκτη start μέχρι το τέλος	s[8:] → 'a test'
s[:end]	Τμήμα της s από την αρχή μέχρι μέχρι και τον δείκτη end-1	s[:4] → 'this'
s[:]	Αντιγραφή της συμβολοσειράς	s[:] → 'this is a test'
s[::-1]	Λήψη αντίστροφής συμβολοσειράς	s[::-1] → 'tset a si siht'



# Η συνάρτηση len() και διάφορες μέθοδοι συμβολοσειρών

- Η συνάρτηση len() επιστρέφει το μήκος μιας συμβολοσειράς
- Μέθοδοι συμβολοσειρών (καλούνται σε αντικείμενα συμβολοσειρών, χρησιμοποιώντας την τελεία και το όνομα της μεθόδου)
  - .upper()
  - .lower()
  - .find(str, start, end)
  - .count(str, start, end)
  - .index(value)
  - .replace(old, new, count)
- Δείτε για λεπτομέρειες το <https://docs.python.org/3/library/stdtypes.html#string-methods>

t	h	i	s		i	s		a		t	e	s	t
0	1	2	3	4	5	6	7	8	9	10	11	12	13
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
>>> s = 'this is a test'
>>> len(s)
14
>>> s.upper()
'THIS IS A TEST'
>>> s.find('is')
2
>>> s.find('is',3)
5
>>> s.find('is',8)
-1
>>> s.count('t')
3
>>> s.index('e')
11
```

```
>>> s = 'this is a test'
>>> s.replace("t", "T", 2)
'This is a Test'
>>> s.replace(" ", "")
'thisisatest'
```

# Τελεστές συμβολοσειρών

- Συνένωση συμβολοσειρών (+)
- Επανάληψη (\*)
- Σχέση μέλους (in, not in)
- Σύγκριση συμβολοσειρών (λεξικογραφικά με τους συγκριτικούς τελεστές)
- Δεικτοδότηση και τεμαχισμός ([], [:])
- Επαυξημένη ανάθεση (+=, \*=)

```
>>> 'Hello' + ' World!'
'Hello World!'
>>> 'hi ' * 5
'hi hi hi hi hi '
>>> 'day' in 'birthday'
True
>>> 'apple' < 'appricot'
True
>>> s = 'Hello'
>>> s += ' World!'
>>> s
'Hello World!'
```

# Εντολές επιλογής και επανάληψης

if, while, for

# Εντολές επιλογής και επανάληψης

## Επιλογή (conditional)

- Η εντολή if/elif/else
- Ο τριαδικός τελεστής if (ternary operation)

## Επανάληψη (repetition)

- Η εντολή while
- Η εντολή for

# Η εντολή επιλογής if

- Η εντολή επιλογής if επιτρέπει την εκτέλεση ενός μπλοκ εντολών με βάση την αποτίμηση μιας συνθήκης
- Μπλοκ εντολών είναι μια ή περισσότερες εντολές που βρίσκονται στο ίδιο επίπεδο κατακόρυφης στοίχισης (indentation) και που εκτελούνται μαζί
- Μια εντολή if μπορεί να βρίσκεται μέσα σε ένα μπλοκ μιας άλλης εντολής if (εμφωλευμένα if)

```
if <συνθήκη1>:  
    <μπλοκ εντολών 1>  
elif <συνθήκη2>:  
    <μπλοκ εντολών 2>  
elif <συνθήκη3>:  
    <μπλοκ εντολών 3>  
...  
else:  
    <μπλοκ εντολών n>
```

# Παράδειγμα με την εντολή if

- Πρόγραμμα που δέχεται μια τιμή από τον χρήστη και εμφανίζει το πρόσημο της τιμής

```
x = input("Εισάγετε έναν αριθμό: ")
x = int(x)
if x > 0:
    print('Θετικός')
elif x < 0:
    print('Αρνητικός')
else:
    print('Μηδέν')
```

Εισάγετε έναν αριθμό: 57  
Θετικός

# Ένα ακόμα παράδειγμα με την εντολή if

- Πρόγραμμα που δέχεται δύο βαθμούς από τον χρήστη και υπολογίζει το μέσο όρο τους. Αν οι βαθμοί απέχουν πάνω από 2 μονάδες τότε ο μικρότερος αλλάζει έτσι ώστε να απέχει 2 βαθμούς από τον μεγαλύτερο βαθμό (π.χ. αν οι βαθμοί είναι 16 και 17 τότε ο μέσος όρος είναι 16.5, αν οι βαθμοί είναι 10 και 20, τότε το 10 γίνεται 18 και ο μέσος όρος  $(18+20)/2 = 19$ )

```
a = float(input("Δώσε τον πρώτο βαθμό: "))
b = float(input("Δώσε το δεύτερο βαθμό: "))
if a > b:
    a, b = b, a      # στο a βρίσκεται η
                    # μικρότερη τιμή και
                    # στο b η μεγαλύτερη τιμή

if b - a > 2:
    a = b - 2
    c = (a + b) / 2
    print(f"Ο προσαρμοσμένος Μ.Ο. είναι {c}")
else:
    c = (a + b) / 2
    print(f"Ο Μ.Ο. είναι {c}")
```

```
Δώσε τον πρώτο βαθμό: 10
Δώσε το δεύτερο βαθμό: 20
Ο προσαρμοσμένος μέσος όρος είναι 19.0
```

## Διπλές ανισότητες σε συνθήκες

- Η Python υποστηρίζει έναν συντομότερο τρόπο γραφής διπλών ανισοτήτων, όπως χρησιμοποιούνται και στα μαθηματικά
- Για παράδειγμα η συνθήκη που ελέγχει ότι η θερμοκρασία είναι μεγαλύτερη από 18 αλλά μικρότερη ή ίση του 26 μπορεί να γραφεί ως  **$t > 18 \text{ and } t \leq 26$**  ή ως  **$18 < t \leq 26$**
- Συνήθως χρησιμοποιείται ο δεύτερος τρόπος (διπλή ανισότητα), για συντομία



# Ο τριαδικός τελεστής στην Python

- Ο τριαδικός τελεστής (ternary operator ή conditional expression) είναι ένα σύντομος τρόπος να γραφεί μια εντολή if/else σε μια γραμμή
- Η σύνταξή του είναι η ακόλουθη:  
τιμή\_αν\_\_αληθής if συνθήκη else τιμή\_αν\_\_ψευδής
- Πρόκειται για έναν συνοπτικό τρόπο που καθιστά τον κώδικα ευανάγνωστο για **απλές συνθήκες**, για αποτύπωση συνθετότερης λογικής να αποφεύγεται

- Πρόγραμμα που εμφανίζει μήνυμα «Ενήλικος» ή «Ανήλικος» ανάλογα με τη ηλικία

```
age = 20
status = "Ενήλικος" if age >= 18 else "Ανήλικος"
print(status) # Ενήλικος
```

## Άσκηση #2 (εντολή if): Εκφώνηση

- Γράψτε πρόγραμμα που να δέχεται από το χρήστη έναν ακέραιο αριθμό και να εμφανίζει με κατάλληλο μήνυμα σε ποια από τις ακόλουθες περιπτώσεις βρίσκεται ο αριθμός:
  - Άρτιος και θετικός
  - Άρτιος και αρνητικός
  - Περιττός και θετικός
  - Περιττός και αρνητικός
  - Μηδέν

## Άσκηση #2: Λύση

```
num = int(input("Δώσε έναν ακέραιο αριθμό: "))

if num == 0:
    print("0 αριθμός είναι μηδέν")
elif num > 0 and num % 2 == 0:
    print("0 αριθμός είναι άρτιος και θετικός")
elif num < 0 and num % 2 == 0:
    print("0 αριθμός είναι άρτιος και αρνητικός")
elif num > 0 and num % 2 != 0:
    print("0 αριθμός είναι περιττός και θετικός")
else:
    print("0 αριθμός είναι περιττός και αρνητικός")
```

Δώσε έναν ακέραιο αριθμό: 1789  
Ο αριθμός είναι περιττός και θετικός

# Η εντολή επανάληψης while

- Η εντολή while επιτρέπει την επαναληπτική εκτέλεση ενός μπλοκ κώδικα (το <μπλοκ εντολών 1>) για όσο ισχύει μια συνθήκη, όταν η συνθήκη γίνει ψευδής, τότε η εκτέλεση συνεχίζεται με τις εντολές μετά την while
- Στην Python έχει προστεθεί στην while και το προαιρετικό τμήμα else:
- Αν υπάρχει σε μια εντολή while το else: και το <μπλοκ εντολών 2>, τότε αυτό θα εκτελεστεί αν η συνθήκη γίνει ψευδής και μόνο τότε (δηλαδή αν δεν γίνει έξοδος από την επανάληψη νωρίτερα με την εντολή break που θα δούμε στη συνέχεια)

```
while <συνθήκη>:  
    <μπλοκ εντολών 1>  
else:  
    <μπλοκ εντολών 2>
```

# Παράδειγμα με την εντολή while

- Πρόγραμμα που δέχεται μια ακέραια τιμή και πραγματοποιεί αντίστροφη μέτρηση μέχρι το 0. Στο τέλος να εμφανίζει το μήνυμα «Τέλος εκτέλεσης»

```
x = int(input("Δώσε μια θετική ακέραια τιμή: "))
while x >= 0:
    print(f"{x}")
    x -= 1 # ισοδύναμο με x = x - 1
print("Τέλος εκτέλεσης")
```

```
Δώσε μια θετική ακέραια τιμή: 10
10
9
8
7
6
5
4
3
2
1
0
Τέλος εκτέλεσης
```

# Η εντολή επανάληψης for

- Η εντολή for επιτρέπει την επαναληπτική εκτέλεση του ίδιου μπλοκ κώδικα για κάθε στοιχείο μιας ακολουθίας (ενός iterable όπως για παράδειγμα μια συμβολοσειρά, μια λίστα κ.α.)
- Γενικά, η for χρησιμοποιείται όταν το πλήθος των επαναλήψεων είναι γνωστό εκ των προτέρων
- Το τμήμα else της εντολής λειτουργεί όπως και στη while, δηλαδή το <μπλοκ εντολών 2> εκτελείται όταν η έξοδος από την επανάληψη προκληθεί λόγω εξάντλησης των στοιχείων της ακολουθίας και όχι αν προκληθεί πρόωρη έξοδος από την επανάληψη (με την break)

iterable  
↙  
**for x in S**  
    **<μπλοκ εντολών 1>**  
**else:**  
    **<μπλοκ εντολών 2>**

# Παράδειγμα με την εντολή for

- Πρόγραμμα που εμφανίζει το κάθε ψηφίο του αριθμού 1234567890 σε νέα γραμμή, μετά την εμφάνιση όλων των ψηφίων εμφανίζει το κείμενο «Τέλος»

```
a = 1234567890
for x in str(a):
    print(x)
print("Τέλος")
```

```
1
2
3
4
5
6
7
8
9
0
Τέλος
```

# Οι εντολές break και continue

- Η εντολή **break** διακόπτει την εκτέλεση του βρόχου επανάληψης στον οποίο βρίσκεται και προκαλεί τη συνέχεια εκτέλεσης με την πρώτη εντολή μετά το βρόχο
- Η εντολή **continue** διακόπτει την εκτέλεση της τρέχουσας επανάληψης του βρόχου επανάληψης στον οποίο βρίσκεται και προκαλεί τη συνέχεια εκτέλεσης της επόμενης επανάληψης του βρόχου

```
s = '12345'  
print("Break")  
for c in s:  
    if c == '3':  
        break  
    print(c)
```

```
print("Continue")  
for c in s:  
    if c == '3':  
        continue  
    print(c)
```

```
Break  
1  
2  
Continue  
1  
2  
4  
5
```



# Η συνάρτηση range()

- Η συνάρτηση range(start, end, step) επιτρέπει τη δημιουργία ακολουθιών ακεραίων αριθμών
  - Το όρισμα start συμπεριλαμβάνεται στην ακολουθία, αν παραληφθεί υπονοείται το 0
  - Το όρισμα end δεν μπορεί να παραληφθεί, και η τιμή end δεν συμπεριλαμβάνεται στην ακολουθία
  - Το όρισμα step καθορίζει το βήμα της ακολουθίας (start, start + step, start + 2\* step, ...), αν παραληφθεί υπονοείται το 1

```
>>> range(10)
range(0, 10)
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,10,3)) # από το 1 μέχρι το 9 με βήμα 3
[1, 4, 7]
>>> list(range(10,0,-1)) # από το 10 μέχρι το 1 με βήμα -1
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

# Εμφωλευμένες επαναλήψεις

- Προπαίδια από 1 έως 5 (μια εκτύπωση με for και μια με while)

```
for i in range(1, 6):
    for j in range(1, 6):
        print(f"{i * j:2}", end=" ")
    print()

print("-" * 20)

i = 1
while i <= 5:
    j = 1
    while j <= 5:
        print(f"{i * j:2}", end=" ")
        j += 1
    print()
    i += 1
```

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25
-----				
1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

## Άσκηση #3 (εντολή for): Εκφώνηση

- Γράψτε κώδικα που να επιλύει το πρόβλημα εντοπισμού ενός τυχαίου ακέραιου αριθμού από το 1 μέχρι το 100, με 6 το πολύ προσπάθειες του χρήστη. Για κάθε επιλογή του χρήστη θα λαμβάνει ανατροφοδότηση σχετικά με το αν η επιλογή του πέτυχε τη ζητούμενη τιμή ή αν είναι μικρότερη ή μεγαλύτερη. Αν ο χρήστης εντοπίσει τη ζητούμενη τιμή σε κάποια από τις 6 προσπάθειες θα ανακηρύσσεται νικητής, αλλιώς θα έχει χάσει το παιχνίδι. Για τη δημιουργία ενός τυχαίου αριθμού (lucky\_number) στο διάστημα [1,100] χρησιμοποιήστε τον ακόλουθο κώδικα:

```
import random  
lucky_number = random.randint(1,100)
```

## Άσκηση #3: Λύση

```
import random
lucky_number = random.randint(1,100)
print("The lucky number was generated!")

winner = False
c = 0
for i in range(6):
    c += 1
    x = int(input(f"Guess the number (try {c}): "))
    if x == lucky_number:
        winner = True
        break
    elif x < lucky_number:
        print("Too low")
    else:
        print("Too high")

if winner:
    print(f"Winner at try {c}!")
else:
    print(f"Looser, the lucky number was {lucky_number}!")
```

```
The lucky number was generated!
Guess the number (try 1): 50
Too low
Guess the number (try 2): 75
Too high
Guess the number (try 3): 62
Too low
Guess the number (try 4): 69
Too low
Guess the number (try 5): 73
Too low
Guess the number (try 6): 74
Winner at try 6!
```

# Συναρτήσεις

Functions

# Συναρτήσεις

- Οι συναρτήσεις (functions) είναι τμήματα κώδικα στα οποία έχει αποδοθεί ένα όνομα, ορίζονται σε ένα σημείο του κώδικα και μπορούν να καλούνται με το όνομα που τους έχει αποδοθεί από διάφορα σημεία του κώδικα, πολλές φορές
- Η κλήση και συνεπώς η εκτέλεση μιας συνάρτησης γίνεται χρησιμοποιώντας το όνομα της σε οποιοδήποτε σημείο κώδικα είναι ορατή
- Μια συνάρτηση δέχεται εισόδους (μπορεί και καμία), εκτελεί εντολές και επιστρέφει κάποια αποτελέσματα (μπορεί και κανένα) με τη χρήση της εντολής return

Η γενική μορφή του ορισμού μιας συνάρτησης είναι η ακόλουθη:

```
def όνομα_συνάρτησης(λίστα_παραμέτρων):  
    εντολές  
    return αποτέλεσμα
```

} Σώμα συνάρτησης

# Πλεονεκτήματα χρήσης συναρτήσεων

- Μια διάσημη τακτική επίλυσης σύνθετων προβλημάτων είναι η «Διαίρει και βασίλευε» (Divide & Conquer), όπου ένα πρόβλημα διασπάται σε άλλα μικρότερα υποπροβλήματα και αυτό συνεχίζεται μέχρι τα επιμέρους υποπροβλήματα να είναι απλά να επιλυθούν - με αυτό τον τρόπο καθίσταται εφικτή η επίλυση δύσκολων προβλημάτων
- Ένα παράπλευρο όφελος είναι ότι οι απαιτούμενες διορθώσεις και αλλαγές εντοπίζονται σε μικρότερα τμήματα κώδικα και αυτό διευκολύνει το έργο του προγραμματιστή
- Χρησιμοποιώντας συναρτήσεις η συγγραφή, κατανόηση, ανάγνωση και διόρθωση του κώδικα που επιλύει ένα πρόβλημα καθίσταται ευκολότερη

# Παραδείγματα ορισμών και κλήσεων συναρτήσεων

## Συνάρτηση που δεν επιστρέφει τιμή

```
# Συνάρτηση που δεν επιστρέφει τιμή
def print_greeting(name):
    print(f"Γεια σου {name}!")

# Κλήση συνάρτησης που δεν επιστρέφει τιμή
print_greeting("Μαρία")

result = print_greeting("Ελένη")
print("Αποτέλεσμα:", result)
```

```
Γεια σου Μαρία!
Γεια σου Ελένη!
Αποτέλεσμα: None
```

## Συνάρτηση που επιστρέφει τιμή

```
# Συνάρτηση που επιστρέφει μια τιμή
def calculate_area(length, width):
    area = length * width
    return area

# Κλήση συνάρτησης calculate_area()
room_area = calculate_area(4.5, 3.2)
print(f"Το εμβαδόν είναι: {room_area} τ.μ.")
```

```
Το εμβαδόν είναι: 14.4 τ.μ.
```



# Παράμετροι και ορίσματα συναρτήσεων

- Μια συνάρτηση στον ορισμό της μπορεί να δέχεται καμία, μια ή περισσότερες παραμέτρους (parameters) που ο σκοπός τους είναι να λειτουργήσουν ως είσοδοι τιμών που θα χρησιμοποιηθούν στο σώμα της συνάρτησης για την υλοποίηση της απαιτούμενης λειτουργικότητας
- Κατά την κλήση μιας συνάρτησης οι τιμές που δίνονται στις παραμέτρους ονομάζονται ορίσματα (arguments)
- Συνεπώς, οι παράμετροι μιας συνάρτησης είναι κατά τον ορισμό της συνάρτησης, ενώ τα ορίσματα κατά την κλήση της

```
# Ορισμός συνάρτησης με 1 παράμετρο
def celsius_to_fahrenheit(celsius):
    fahrenheit = (celsius * 9/5) + 32
    return fahrenheit
```

```
# Κλήσεις συνάρτησης
temp1 = celsius_to_fahrenheit(0)      # Όρισμα: 0
temp2 = celsius_to_fahrenheit(100)    # Όρισμα: 100

print(f"0°C = {temp1}°F")
print(f"100°C = {temp2}°F")
```

0°C = 32.0°F  
100°C = 212.0°F

# Επιστροφή αποτελεσμάτων από συναρτήσεις

- Οι συναρτήσεις στην Python πάντα επιστρέφουν κάτι, αν μια συνάρτηση δεν επιστρέφει ρητά κάποια τιμή (με return) τότε επιστρέφει την τιμή None
- Είναι σύνηθες μια συνάρτηση να επιστρέφει περισσότερες από 1 τιμές, δείτε το παράδειγμα:

```
import math

def circle_properties(radius):
    area = math.pi * radius**2
    circumference = 2 * math.pi * radius
    diameter = 2 * radius
    return area, circumference, diameter
```

← χρειάζεται για το math.pi  
που επιστρέφει την τιμή του π  
που είναι κατά προσέγγιση  
3.141592653589793

```
unpacking → area, circum, diam = circle_properties(5)
print(f"Εμβαδόν: {area:.2f}, περίμετρος: {circum:.2f}, διάμετρος: {diam}")
```

Εμβαδόν: 78.54, περίμετρος: 31.42, διάμετρος: 10

# Κλήση συναρτήσεων

- Στην Python υπάρχουν 2 τρόποι που μπορούμε να «περάσουμε» ορίσματα κατά την κλήση μιας συνάρτησης
  - α) είτε να χρησιμοποιηθούν ορίσματα θέσης (positional arguments) που αντιστοιχίζονται σε παραμέτρους με βάση τη θέση τους
  - β) είτε να χρησιμοποιηθούν ορίσματα λέξεων κλειδιών (keyword arguments) ή αλλιώς ονοματισμένα ορίσματα (named arguments), οπότε η σειρά εμφάνισης των ορισμάτων στην κλήση της συνάρτησης μπορεί να είναι οποιαδήποτε
- Μπορούν να χρησιμοποιηθούν και οι 2 τρόποι μαζί στην ίδια κλήση συνάρτησης

# Παράδειγμα κλήσης συνάρτησης με ορίσματα θέσης και με ονοματισμένα ορίσματα

```
def calculate_rectangle_area(length, width):  
    return length * width  
  
r1 = calculate_rectangle_area(10, 20) # ορίσματα θέσης  
r2 = calculate_rectangle_area(length=10, width=20) # ονοματισμένα ορίσματα  
r3 = calculate_rectangle_area(width=20, length=10) # ονοματισμένα ορίσματα  
print(f"{r1=}, {r2=}, {r3=}")
```

```
r1=200, r2=200, r3=200
```

# Παράδειγμα κλήσης συνάρτησης με μίξη ορισμάτων θέσης και ονοματισμένων ορισμάτων

```
def process_order(item_name, quantity, price, tax_rate, express_shipping):  
    total = quantity * price * (1 + tax_rate)  
    if express_shipping:  
        total += 10  
    print(f"Παραγγελία: {quantity} x {item_name} = {total:.2f}€")
```

# Μόνο ορίσματα θέσης

```
process_order("Φορητός Η/Υ", 1, 999.99, 0.24, True)
```

# Μίξη ορισμάτων θέσης και ονοματισμένων ορισμάτων

```
process_order("Κινητό", 2, 599.99, express_shipping=True, tax_rate=0.24)
```

Παραγγελία: 1 x Φορητός Η/Υ = 1249.99€

Παραγγελία: 2 x Κινητό = 1497.98€

```
process_order(tax_rate=0.24, "Κινητό", 2, 599.99, express_shipping=True) ❌
```

Η παραπάνω κλήση της process\_order είναι λάθος διότι πρέπει να προηγούνται τα ορίσματα θέσης από τα ονοματισμένα ορίσματα κατά την κλήση, αλλιώς προκαλείται *SyntaxError: positional argument follows keyword argument*

# Τεκμηρίωση συνάρτησης με συμβολοσειρές τεκμηρίωσης (1/2)

- Οι συμβολοσειρές τεκμηρίωσης (docstrings) είναι συμβολοσειρές πολλών γραμμών (συνεπώς περικλείονται σε τριπλά εισαγωγικά) που τοποθετούνται στην αρχή του σώματος των συναρτήσεων με σκοπό την περιγραφή της λειτουργίας των συναρτήσεων
- Η τοποθέτηση docstrings στις συναρτήσεις είναι προαιρετική, αλλά συνίσταται
- Από τη στιγμή που μια συνάρτηση διαθέτει docstring, μια κλήση της μορφής <όνομα\_συνάρτησης>.\_\_doc\_\_ θα επιστρέψει το κείμενο του docstring της συνάρτησης <όνομα\_συνάρτησης>
  - Το \_\_doc\_\_ είναι ένα παράδειγμα dunder, δηλαδή ενός αναγνωριστικού που ξεκινά και τελειώνει με 2 κάτω παύλες (double underscores)
- Αν κληθεί η help με όρισμα το όνομα της συνάρτησης θα εμφανιστεί το docstring σε περιβάλλον εμφάνισης βοήθειας (για έξοδο πρέπει να πατηθεί το πλήκτρο q)

# Τεκμηρίωση συνάρτησης με συμβολοσειρές τεκμηρίωσης (2/2)

```
def calculate_area(radius):  
    """  
    Calculate the area of a circle given its radius.  
  
    Parameters:  
    radius (float): The radius of the circle.  
  
    Returns:  
    float: The area of the circle.  
    """  
    import math  
    return math.pi * radius ** 2  
  
print(calculate_area.__doc__)
```

Calculate the area of a circle given its radius.

Parameters:

radius (float): The radius of the circle.

Returns:

float: The area of the circle.

# Η εντολή pass

- Κατά τη συγγραφή ενός προγράμματος μπορεί να έχει ανιχνευθεί η ανάγκη ύπαρξης μιας συνάρτησης αλλά να μην επιλεγεί η συγγραφή του σώματός της στη συγκεκριμένη φάση
- Τότε και προκειμένου να διατηρηθεί η συντακτική ορθότητα του κώδικα, γράφεται κανονικά η πρώτη γραμμή της συνάρτησης, αλλά ακολουθείται από την λέξη pass ή τρεις τελείες ... ακριβώς για να υποδηλωθεί ότι το σώμα της συνάρτησης θα συμπληρωθεί αργότερα
- Η εντολή pass είναι χρήσιμη για τη συγγραφή του «σκελετού» ενός κώδικα
- Επίσης, το pass μπορεί να χρησιμοποιηθεί στη θέση ενός μπλοκ κώδικα στις if, for, while που πρόκειται να υλοποιηθεί αργότερα

```
def foo():  
    pass  
  
def bar(a_parameter, another_parameter):  
    ...  
  
foo()  
bar(1,2)  
  
age = 20  
  
if age >= 18:  
    pass # θα υλοποιηθεί αργότερα  
else:  
    print("Ανήλικος")
```



# Προαιρετικές παράμετροι συναρτήσεων

- Οι παράμετροι των συναρτήσεων μπορούν είτε να είναι υποχρεωτικές να συμπληρωθούν για να είναι έγκυρη η κλήση της συνάρτησης, είτε να είναι προαιρετικές οπότε για κάθε προαιρετική παράμετρο ορίζεται μια προκαθορισμένη τιμή (default value) που χρησιμοποιείται αν δεν δοθεί αντίστοιχη τιμή ορίσματος

```
def calculate_interest(principal, rate=0.05, time=1, compound_frequency=1):  
    amount = principal * (1 + rate / compound_frequency) ** (compound_frequency * time)  
    interest = amount - principal  
    return round(interest, 2)
```

```
# default τιμές: επιτόκιο 5%, 1 έτος, ετήσια εφαρμογή επιτοκίου  
interest1 = calculate_interest(1000)  
print(f"Τόκοι: {interest1}€")
```

```
# Προσαρμοσμένες τιμές: επιτόκιο 8%, 2 έτη, ετήσια εφαρμογή επιτοκίου  
interest2 = calculate_interest(1000, rate=0.08, time=2)  
print(f"Τόκοι: {interest2}€")
```

Τόκοι: 50.0€  
Τόκοι: 166.4€

## Άσκηση #4 (συναρτήσεις): Εκφώνηση

- Ορίστε τη συνάρτηση **calculate\_total(price, quantity=1, discount=0)** που πολλαπλασιάζει την τιμή (price) με την ποσότητα (quantity), εφαρμόζει ένα ποσοστό έκπτωσης (discount) που είναι μια τιμή από 0 μέχρι 100 και επιστρέφει το συνολικό κόστος
- Καλέστε τη συνάρτηση για price = 75, quantity = 1, discount = 0 και εμφανίστε το αποτέλεσμα
- Καλέστε τη συνάρτηση για price = 75, quantity = 3, discount = 12 και εμφανίστε το αποτέλεσμα
- Καλέστε τη συνάρτηση για τιμές που δίνει ο χρήστης, πρώτα για την τιμή, μετά για την ποσότητα όπου αν ο χρήστης πατήσει enter χωρίς να εισάγει τιμή να χρησιμοποιείται η προκαθορισμένη τιμή και μετά για το ποσοστό έκπτωσης που αν ο χρήστης πατήσει enter χωρίς να εισάγει τιμή να χρησιμοποιείται επίσης η προκαθορισμένη τιμή

## Άσκηση #4: Λύση

```
def calculate_total(price, quantity=1, discount=0):  
    """Υπολογισμός συνολικού κόστους λαμβάνοντας υπόψη τιμή, ποσότητα και έκπτωση."""  
    subtotal = price * quantity  
    total = subtotal * (1 - discount / 100)  
    return total  
  
print(f"{calculate_total(75):.2f}") # 75.00  
print(f"{calculate_total(75, 3, 12):.2f}") # 198.00  
price = float(input("Τιμή μονάδας: "))  
  
quantity_input = input("Ποσότητα (enter για 1): ")  
if quantity_input.strip() == "":  
    quantity = 1  
else:  
    quantity = int(quantity_input)  
  
discount_input = input("Έκπτωση (enter για 0): ")  
if discount_input.strip() == "":  
    discount = 0  
else:  
    discount = float(discount_input)  
  
total_cost = calculate_total(price, quantity, discount)  
print(f"Σύνολο: {total_cost}")
```

```
75.00  
198.00  
Τιμή μονάδας: 55  
Ποσότητα (enter για 1):  
Έκπτωση (enter για 0): 20  
Σύνολο: 44.0
```

# Συναρτήσεις με μεταβλητό πλήθος ορισμάτων

- Μια συνάρτηση μπορεί να έχει μεταβλητό πλήθος ορισμάτων
- Για να συμβεί αυτό χρησιμοποιείται ο τελεστής \* ή ο τελεστής \*\* που λειτουργούν ως τελεστές ανάπτυξης δομής
- Ειδικότερα ισχύουν τα:
  - \*args = μεταβλητό πλήθος ορισμάτων θέσης (positional arguments)
  - \*\*kwargs = μεταβλητό πλήθος ονοματισμένων ορισμάτων (keyword arguments)
- Τα ονόματα args και kwargs δεν είναι υποχρεωτικά, αλλά χρησιμοποιούνται αρκετά συχνά

# Ορίσματα τύπου \*args

- Η χρήση \*args επιτρέπει την ευέλικτη χρήση ορισμάτων θέσης (positional arguments) που δεν έχουν προκαθοριστεί στον ορισμό της συνάρτησης
- Η ακόλουθη συνάρτηση μπορεί να κληθεί με οποιοδήποτε πλήθος ορισμάτων

```
def my_function(*args):  
    print(f"Ελήφθησαν {len(args)} ορίσματα")  
    for i, arg in enumerate(args):  
        print(f"Όρισμα {i}: {arg}")  
  
my_function() # 0 ορίσματα  
my_function(1) # 1 όρισμα  
my_function(1, 2, 3) # 3 ορίσματα  
my_function("a", "b", "c", "d") # 4 ορίσματα
```

```
Ελήφθησαν 0 ορίσματα  
Ελήφθησαν 1 ορίσματα  
Όρισμα 0: 1  
Ελήφθησαν 3 ορίσματα  
Όρισμα 0: 1  
Όρισμα 1: 2  
Όρισμα 2: 3  
Ελήφθησαν 4 ορίσματα  
Όρισμα 0: a  
Όρισμα 1: b  
Όρισμα 2: c  
Όρισμα 3: d
```

# Ορίσματα τύπου `**kwargs`

- Η χρήση `**kwargs` επιτρέπει την ευέλικτη χρήση ονοματισμένων ορισμάτων (keyword arguments) που δεν έχουν προκαθοριστεί στον ορισμό της συνάρτησης
- Το `kwargs` είναι ένα λεξικό (θα μιλήσουμε για τα λεξικά στη συνέχεια, για την ώρα αρκεί να γνωρίζουμε ότι ένα λεξικό αποτελείται από ζεύγη κλειδί: τιμή)

```
def my_function(**kwargs):  
    print(f"Ελήφθησαν {len(kwargs)} ονοματισμένα ορίσματα")  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")
```

```
my_function() # 0 ορίσματα  
my_function(name="Alice") # 1 ονοματισμένο όρισμα  
my_function(name="Bob", age=30) # 2 ονοματισμένα ορίσματα  
my_function(city="Ioannina", country="GR", \  
            population=115000) # 3 ορίσματα
```

```
Ελήφθησαν 0 ονοματισμένα ορίσματα  
Ελήφθησαν 1 ονοματισμένα ορίσματα  
name: Alice  
Ελήφθησαν 2 ονοματισμένα ορίσματα  
name: Bob  
age: 30  
Ελήφθησαν 3 ονοματισμένα ορίσματα  
city: Ioannina  
country: GR  
population: 115000
```

# Κανόνες για συνδυασμό args, kwargs, \*args και \*\*kwargs

- args = positional arguments (ορίσματα θέσης)
- kwargs = keyword arguments (ονοματισμένα ορίσματα)
- Κατά τον ορισμό συναρτήσεων:
  - Οι παράμετροι \*\*kwargs, αν υπάρχουν, πρέπει πάντα να είναι τελευταίοι
- Κατά την κλήση συναρτήσεων:
  - Τα args πρέπει να βρίσκονται πριν τα kwargs
  - Σε συναρτήσεις με παραμέτρους μετά το \*args, τα ορίσματα που τους δίνουν τιμές πρέπει να είναι kwargs

```
# def f1(**kwargs, x): # SyntaxError: invalid syntax
#     pass
```

```
def f2(x, y):
    print(f"{x=}, {y=}")
```

```
f2(1, 2) # x=1, y=2
```

```
f2(x=1, y=2) # x=1, y=2
```

```
f2(1, y=2) # x=1, y=2
```

```
# f2(1, x=2) # TypeError: f2() got multiple values for argument 'x'
```

```
# f2(x=1, 2) # SyntaxError: positional argument follows keyword argument
```

```
def f3(x, *args, y, **kwargs):
    print(f"{x=}, {args=}, {y=}, {kwargs=}")
```

```
f3(1, 2, 3, y=4, z=5, w=6) # x=1, args=(2, 3), y=4, kwargs={'z': 5, 'w': 6}
```

```
# f3(1, 2, 3, 4, z=5, w=6) # TypeError: f3() missing 1 required keyword-only argument: 'y'
```

## Άσκηση #5 (μεταβλητό πλήθος ορισμάτων): Εκφώνηση

- Ορίστε τη συνάρτηση `evens_max(*args)` που δέχεται οποιοδήποτε πλήθος ακέραιων ορισμάτων και επιστρέφει τη μέγιστη τιμή από τις άρτιες τιμές ορισμάτων ή -1 αν δεν υπάρχει όρισμα που να είναι άρτιο
- Καλέστε από το κύριο πρόγραμμα τη συνάρτηση `evens_max` για τα ακόλουθα ορίσματα και εμφανίστε το αποτέλεσμα:
  - 1, 2, 3, 4, 5
  - -10, -20, -30, -40
  - 1, 3, 5, 7, 9
- Να μη χρησιμοποιηθεί η built-in συνάρτηση `max()` της Python



## Άσκηση #5: Λύση

```
def evens_max(*args):
    """Επιστρέφει τη μέγιστη τιμή από τα άρτια ορίσματα ή -1 αν δεν υπάρχει."""
    max_even = None
    for x in args:
        if x % 2 == 0:
            if max_even is None:
                max_even = x
            elif x > max_even:
                max_even = x

    if max_even is None:
        return -1
    else:
        return max_even

print(evens_max(1, 2, 3, 4, 5))      # 4
print(evens_max(-10, -20, -30, -40)) # -10
print(evens_max(1, 3, 5, 7, 9))      # -1
```

Υπάρχει συντομότερη pythonic  
λύση που χρησιμοποιεί τη  
built-in συνάρτηση max()

# Τοπικές μεταβλητές

- Οι μεταβλητές που δημιουργούνται μέσα σε μια συνάρτηση έχουν τοπική εμβέλεια (ονομάζονται τοπικές μεταβλητές), δηλαδή μπορούν να χρησιμοποιηθούν μόνο μέσα στη συνάρτηση
- Αν γίνει απόπειρα αναφοράς μιας μεταβλητής εκτός της συνάρτησης τότε θα προκληθεί `NameError` και η εκτέλεση του κώδικα θα διακοπεί με μήνυμα σφάλματος ότι το συγκεκριμένο όνομα είναι `not defined`

```
def greet():  
    message = "Hello, world!" # τοπική μεταβλητή που  
    print(message) # χρησιμοποιείται μέσα στη συνάρτηση  
  
greet() # ✓  
print(message) # ✗ Error: NameError: name 'message'  
# is not defined
```

```
Hello, world!  
Traceback (most recent call last):  
  File "/PYTHON-A/week2/functions_example11.py", line 6, in <module>  
    print(message) # ✗ Error: NameError: name 'message' is not defined  
NameError: name 'message' is not defined
```

# Καθολικές μεταβλητές

- Οι καθολικές μεταβλητές (global variables) δηλώνονται εκτός συναρτήσεων και είναι προσπελάσιμες οπουδήποτε στο πρόγραμμα (εντός και εκτός συναρτήσεων)
- Ωστόσο, για να τροποποιηθεί η τιμή μιας καθολικής μεταβλητής με εντολή που βρίσκεται μέσα σε μια συνάρτηση θα πρέπει η καθολική μεταβλητή να δηλωθεί μέσα στη συνάρτηση ως `global`
- Γενικά ισχύει ότι μέσα σε μια συνάρτηση οι εξωτερικές μεταβλητές μπορούν να αναγνωστούν αλλά δεν μπορούν να τροποποιηθούν (εκτός και αν δηλωθούν ως `global` ή `nonlocal` όπως θα δούμε στη συνέχεια)

```
# καθολική μεταβλητή
exchange_rate = 1.15 # EUR -> USD

def convert_to_usd(amount_eur):
    return amount_eur * exchange_rate

def update_rate(new_rate):
    global exchange_rate # πρέπει να υπάρχει
    exchange_rate = new_rate

print(convert_to_usd(50))
update_rate(1.2)
print(convert_to_usd(50))
```

```
57.49999999999999
60.0
```

# Συναρτήσεις μέσα σε συναρτήσεις

- Η Python δίνει τη δυνατότητα να οριστεί μια συνάρτηση, εμφωλευμένα μέσα σε μια άλλη συνάρτηση, οπότε και ονομάζεται εμφωλιασμένη συνάρτηση (nested function) ή εσωτερική συνάρτηση (inner function)
- Η εσωτερική συνάρτηση μπορεί να αποτελεί βοηθητικό κώδικα που δεν θέλουμε να είναι ορατός εκτός της περικλείουσας (εξωτερικής) συνάρτησης, αλλά να χρησιμοποιείται από αυτή
- Οι εσωτερικές συναρτήσεις μπορούν να προσπελαύνουν μεταβλητές από την εξωτερική εμβέλεια
- Η εσωτερική συνάρτηση δημιουργείται κάθε φορά που καλείται η εξωτερική συνάρτηση
- Οι εσωτερικές συναρτήσεις συνδέονται με μια προχωρημένη προγραμματιστική έννοια που ονομάζεται κλειστότητα (closure), στην οποία όμως δεν θα αναφερθούμε

```
def total_price(base_price, tax_rate):  
    # εσωτερική συνάρτηση: υπολογίζει το φόρο  
    def tax(amount):  
        return amount * tax_rate  
  
    # κλήση εσωτερικής συνάρτησης  
    tax_amount = tax(base_price)  
    return base_price + tax_amount  
  
print(total_price(100, 0.24))  
print(total_price(50, 0.10))
```

124.0  
55.0

# Μη τοπικές μεταβλητές

- Οι μη-τοπικές (nonlocal) μεταβλητές μπορούν να εμφανιστούν σε περίπτωση συναρτήσεων με εσωτερικές συναρτήσεις
- Αν μια μεταβλητή σε μια εσωτερική συνάρτηση δηλωθεί ως nonlocal, αυτό σημαίνει ότι αναφέρεται σε μια μεταβλητή με αυτό το όνομα στην πλησιέστερη περικλείουσα συνάρτηση και μπορεί να τροποποιήσει την τιμή της
- Οι μη τοπικές μεταβλητές είναι χρήσιμες όταν υπάρχει μια μεταβλητή σε μια περικλείουσα συνάρτηση που θέλουμε να ενημερωθεί μέσα από την εσωτερική συνάρτηση

```
def shopping_cart():  
    total = 0 # εξωτερική μεταβλητή για την add_items  
  
    def add_items(prices):  
        nonlocal total # επιτρέπει τροποποίηση της total  
        for price in prices:  
            total += price  
            print(f"+ {price}, νέο σύνολο {total}")  
  
    add_items([10, 25, 5])  
    print(f"Τελικό σύνολο: {total}")
```

shopping\_cart()

```
+ 10, νέο σύνολο 10  
+ 25, νέο σύνολο 35  
+ 5, νέο σύνολο 40  
Τελικό σύνολο: 40
```

# Η δεσμευμένη λέξη assert

- Η δεσμευμένη λέξη assert δίνει τη δυνατότητα ελέγχου των ορισμάτων που περνάνε σε μια συνάρτηση έτσι ώστε αν ληφθούν μη επιτρεπτές τιμές να προκαλείται πρόωρος τερματισμός της εκτέλεσης του κώδικα
- Πρόκειται για μια μορφή προστασίας του κώδικα που επιβάλλει την τήρηση περιορισμών έτσι ώστε μόνο εφόσον οι περιορισμοί ικανοποιούνται να προχωρά η εκτέλεση του κώδικα
- Με ένα προαιρετικό μήνυμα στην assert μπορεί να παρέχεται εξήγηση για το σφάλμα

```
def divide(a, b):  
    """  
    Διαίρεση δύο αριθμών, διασφαλίζοντας  
    ότι ο διαιρέτης δεν είναι μηδέν.  
    """  
    assert b != 0, "Denominator must not be zero!"  
    return a / b
```

```
print(divide(10, 2))  
print(divide(5, 0))
```

5.0

Traceback (most recent call last):

File "/PYTHON-A/week2/functions\_example15.py", line 10, in <module>

print(divide(5, 0))

File "/PYTHON-A/week2/functions\_example15.py", line 5, in divide

assert b != 0, "Denominator must not be zero!"

AssertionError: Denominator must not be zero!

## Άσκηση #5 (καθολικές και τοπικές μεταβλητές): Εκφώνηση

- Γράψτε ένα πρόγραμμα που να δημιουργεί τις καθολικές μεταβλητές `temp_sum = 0`, `temp_count = 0`, `temp_max = None` για άθροισμα, πλήθος και μέγιστη από θερμοκρασίες που έχουν καταγραφεί
- Ορίστε τη συνάρτηση `add_temperature(temp)` που με `assert` εξασφαλίζει ότι η θερμοκρασία `temp` είναι μεταξύ -30, 60 και ενημερώνει τις καθολικές μεταβλητές
- Ορίστε τη συνάρτηση `average_temperature()` που επιστρέφει το μέσο όρο των θερμοκρασιών και με `assert` εξασφαλίζει ότι έχει καταγραφεί τουλάχιστον μια θερμοκρασία
- Ορίστε τη συνάρτηση `max_temperature()` που επιστρέφει τη μέγιστη θερμοκρασία και με `assert` εξασφαλίζει ότι έχει καταγραφεί τουλάχιστον μια θερμοκρασία
- Καλέστε τις συναρτήσεις έτσι ώστε να προκληθεί μια φορά `AssertionError` και μια φορά να εμφανιστούν τα αποτελέσματα, μέσος όρος καταγεγραμμένων θερμοκρασιών, και μέγιστη καταγεγραμμένη θερμοκρασία

# Άσκηση #5: Λύση

```
temp_sum, temp_count, temp_max = 0, 0, None
```

```
def add_temperature(temp):  
    """Προσθέτει θερμοκρασία και ενημερώνει τις καθολικές μεταβλητές."""  
    global temp_sum, temp_count, temp_max  
    assert -30 <= temp <= 60, "Η θερμοκρασία πρέπει να είναι μεταξύ -30 και 60"  
    temp_sum += temp  
    temp_count += 1  
    if temp_max is None or temp > temp_max:  
        temp_max = temp
```

```
def average_temperature():  
    """Επιστρέφει το μέσο όρο θερμοκρασιών."""  
    assert temp_count > 0, "Δεν έχει καταγραφεί καμία θερμοκρασία"  
    return temp_sum / temp_count
```

```
def max_temperature():  
    """Επιστρέφει τη μέγιστη θερμοκρασία."""  
    assert temp_count > 0, "Δεν έχει καταγραφεί καμία θερμοκρασία"  
    return temp_max
```

```
# Προσθήκη θερμοκρασιών (έγκυρες)  
add_temperature(20); add_temperature(25); add_temperature(18); add_temperature(30)  
print("Μέσος όρος θερμοκρασιών:", average_temperature()) # 23.25  
print("Μέγιστη θερμοκρασία:", max_temperature()) # 30  
add_temperature(100) # AssertionError
```

```
Μέσος όρος θερμοκρασιών: 23.25  
Μέγιστη θερμοκρασία: 30  
Traceback (most recent call last):  
  File "/exercise.py", line 38, in <module>  
    add_temperature(100) # Αυτό θα προκαλέσει AssertionError  
  File "/exercise.py", line 9, in add_temperature  
    assert -30 <= temp <= 60, "Η θερμοκρασία πρέπει να είναι  
    μεταξύ -30 και 60"  
AssertionError: Η θερμοκρασία πρέπει να είναι μεταξύ -30 και 60
```



# Αναδρομή

- Μια ενδιαφέρουσα κατηγορία συναρτήσεων είναι οι αναδρομικές συναρτήσεις (recursive functions)
- Μια αναδρομική συνάρτηση καλεί τον εαυτό της και επαναλαμβάνει αυτή τη συμπεριφορά μέχρι να φτάσει στη λεγόμενη «βασική περίπτωση», οπότε και τερματίζεται η αναδρομή
- Γενικά μια αναδρομική συνάρτηση πρέπει να έχει:
  - Βασική περίπτωση: Μια συνθήκη που όταν θα συμβεί η συνάρτηση θα σταματήσει να καλεί τον εαυτό της
  - Αναδρομική περίπτωση: Το τμήμα που η συνάρτηση καλεί ξανά τον εαυτό της για ένα μικρότερο πρόβλημα
- Η αναδρομή (recursion) είναι χρήσιμη σε προβλήματα που μπορούν να διασπαστούν σε παρόμοια προβλήματα, όπως συμβαίνει στη διάσχιση δομών δεδομένων στην περίπτωση των δένδρων και των γραφημάτων

# Παράδειγμα αναδρομικής συνάρτησης (1/2)

- Παραγοντικό: Το παραγοντικό ενός μη αρνητικού ακέραιου αριθμού, συμβολίζεται με ! και ορίζεται ως εξής:
  - $0! = 1$
  - $n! = n * (n-1) * (n-2) * \dots * 1$
- Μπορεί να οριστεί με πολύ φυσικό τρόπο με αναδρομική συνάρτηση που μοιάζει πολύ με τον ακόλουθο αναδρομικό μαθηματικό τύπο:
  - $$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n > 0 \end{cases}$$

# Παράδειγμα αναδρομικής συνάρτησης (2/2)

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n - 1)! & n > 0 \end{cases}$$

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

```
print(factorial(5))  
print(factorial(10))  
print(factorial(20))
```

```
120  
3628800  
2432902008176640000
```

Python Tutor: Visualize Code and Get AI Help for [Python](#), [JavaScript](#), [C](#), [C++](#), and [Java](#)

Python 3.11  
[known limitations](#)

```
1 def factorial(n):  
2     if n == 0:  
3         return 1 # Βασική περίπτωση  
4     else:  
5         return n * factorial(n - 1) # Αναδρομική περίπτωση  
6  
7  
8 print(factorial(5))
```

[Edit this code](#)

→ line that just executed  
→ next line to execute

Step 21 of 26

**NEW:** teachers get [free access](#) to ad-free/AI-free mode

Improve this tool by taking a [3-question survey](#)

[Move and hide objects](#)

Print output (drag lower right corner to resize)

Frames

Global frame

factorial

function factorial(n)

factorial

n 5

factorial

n 4

factorial

n 3

factorial

n 2

factorial

n 1

factorial

n 0

Return value 1

<https://pythontutor.com/>

## Άσκηση #6 (αναδρομή): Εκφώνηση

- Γράψτε μια αναδρομική συνάρτηση που να εξετάζει αν μια συμβολοσειρά είναι παλίνδρομη, διαβάζεται δηλαδή η ίδια από αριστερά προς τα δεξιά και από δεξιά προς τα αριστερά
- Καλέστε τη συνάρτηση για τις συμβολοσειρές RADAR, ΣΟΦΟΣ και PYTHON

# Άσκηση #6: Λύση

```
def is_palindrome(s):  
    if len(s) <= 1:  
        return True  
    if s[0] != s[-1]:  
        return False  
    return is_palindrome(s[1:-1])  
  
s1 = "RADAR"  
if is_palindrome(s1):  
    print(f"{s1} είναι παλίνδρομη.")  
else:  
    print(f"{s1} δεν είναι παλίνδρομη.")  
  
s2 = "ΣΟΦΟΣ"  
if is_palindrome(s2):  
    print(f"{s2} είναι παλίνδρομη.")  
else:  
    print(f"{s2} δεν είναι παλίνδρομη.")  
  
s3 = "PYTHON"  
if is_palindrome(s3):  
    print(f"{s3} είναι παλίνδρομη.")  
else:  
    print(f"{s3} δεν είναι παλίνδρομη.")
```

RADAR είναι παλίνδρομη.  
ΣΟΦΟΣ είναι παλίνδρομη.  
PYTHON δεν είναι παλίνδρομη.

# Λάμδα συναρτήσεις

- Μια λάμδα συνάρτηση είναι μια μικρή ανώνυμη συνάρτηση που ορίζεται με τη λέξη-κλειδί `lambda`, με την ακόλουθη σύνταξη:
  - `lambda` ορίσματα: έκφραση
  - Η λάμδα συνάρτηση δέχεται οποιοδήποτε πλήθος ορισμάτων και επιστρέφει την τιμή της έκφρασης
- Οι λάμδα συναρτήσεις πρέπει να είναι μικρές, αλλιώς είναι προτιμότερο να χρησιμοποιούνται κανονικές συναρτήσεις

```
# Παραδοσιακή συνάρτηση
def square(x):
    return x**2
```

```
print(square(5)) # Έξοδος: 25
a = (lambda x: x**2)(5)
print(a) # Έξοδος: 25
f = lambda x: x**2
print(f(5)) # Έξοδος: 25
```

# Παραδείγματα lambda συναρτήσεων

- Οι λάμδα συναρτήσεις είναι χρήσιμες για γρήγορους υπολογισμούς μέσα σε μεγαλύτερες εκφράσεις, για προσαρμοσμένη ταξινόμηση (μέσω της παραμέτρου key των συναρτήσεων ταξινόμησης) και ως ορίσματα συναρτήσεων όπως οι map(), filter(), reduce()
  - οι map() και filter() είναι built-in συναρτήσεις, ενώ η reduce() ορίζεται στο module functools της standard βιβλιοθήκης της Python
- Σε αυτές τις περιπτώσεις συνηθίζεται να χρησιμοποιούνται inline

```
pairs = [(1, 3), (2, 2), (0, 5), (4, 1)]
```

```
# Ταξινόμηση με βάση το άθροισμα των στοιχείων κάθε ζεύγους
pairs_sorted = sorted(pairs, key=lambda pair: pair[0] + pair[1])

print(pairs_sorted) # [(1, 3), (2, 2), (0, 5), (4, 1)]
```

---

```
from functools import reduce
nums = [1, 2, 3, 4, 5, 6]
```

```
# map: υπολογισμός τετραγώνου κάθε τιμής
squared = list(map(lambda x: x**2, nums))
print("Τετράγωνο:", squared) # [1, 4, 9, 16, 25, 36]
```

```
# filter: διατήρηση μόνο των περιττών τιμών
evens = list(filter(lambda x: x % 2 == 0, squared))
print("Άρτιοι αριθμοί:", evens) # [2, 4, 6]
```

```
# reduce: άθροιση των τιμών
total = reduce(lambda x, y: x + y, evens)
print("Σύνολο:", total) # 56
```

# Βασικές δομές δεδομένων της Python

Λίστες (lists), πλειάδες (tuples), σύνολα (sets), λεξικά (dictionaries)



# Λίστες

- Οι λίστες (lists) είναι δομές δεδομένων που μπορούν να περιέχουν ετερογενή στοιχεία που διαχωρίζονται μεταξύ τους με κόμματα και περικλείονται σε αγκύλες τύπου [ και ]
- Μια λίστα μπορεί να αλλάζει κατά την εκτέλεση του προγράμματος, είτε προσθέτοντας ή αφαιρώντας στοιχεία είτε αντικαθιστώντας υπάρχοντα στοιχεία της λίστας (στην ορολογία της Python οι λίστες είναι mutable = μεταλλάξιμες)
- Οι λίστες είναι ακολουθίες (sequences) συνεπώς τα στοιχεία τους είναι διατεταγμένα (ordered)
- Η αναφορά στα στοιχεία μιας λίστας γίνεται μέσω της θέσης τους και οι δείκτες θέσης ξεκινούν από το 0
  - Για παράδειγμα η αναφορά στο δεύτερο στοιχείο της λίστας a\_list γίνεται με το a\_list[1]

```
>>> a_list = [1, 3.14, "python", -9]
>>> a_list
[1, 3.14, 'python', -9]
>>> a_list[1]
3.14
```

# Τελεστές που εφαρμόζονται σε λίστες

- Στις λίστες τα στοιχεία είναι διατεταγμένα, η θέση των στοιχείων έχει σημασία καθώς αναφερόμαστε σε αυτά με τη θέση τους
- Μπορεί να γίνει συνένωση δύο λιστών με τον τελεστή +
- Μπορεί από μια λίστα να προκύψει μια νέα λίστα με αντίγραφα των στοιχείων της επί όσες φορές επιθυμούμε με τον τελεστή \*
- Υπάρχει και ο τελεστής del για διαγραφή ενός στοιχείου μιας λίστας

```
>>> a = [10, 20, 30]
>>> b = [70, 80, 90]
>>> a + b
[10, 20, 30, 70, 80, 90]
>>> a * 3
[10, 20, 30, 10, 20, 30, 10, 20, 30]
>>> del a[1]
>>> a
[10, 30]
```

# Τεμαχισμός λιστών

- Οι λίστες μπορούν να τεμαχίζονται όπως και οι συμβολοσειρές χρησιμοποιώντας σύνταξη της μορφής [start:end:step] στο τέλος της λίστας

- Παραδείγματα με τεμαχισμό λιστών

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> numbers[2:5]
[2, 3, 4]
>>> numbers[:4]
[0, 1, 2, 3]
>>> numbers[6:]
[6, 7, 8, 9]
>>> numbers[::2]
[0, 2, 4, 6, 8]
>>> numbers[::-1]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

# Μέθοδοι προσθήκης στοιχείων σε λίστες

- `.append(x)`: προσθέτει ένα νέο στοιχείο `x` στο τέλος της λίστας
  - `.insert(x, i)`: εισάγει ένα νέο στοιχείο `x` σε μια συγκεκριμένη θέση `i` της λίστας
  - `.extend(b)`: προσθέτει μια λίστα `b` το τέλος μιας άλλης λίστας
- Παραδείγματα κλήσης μεθόδων προσθήκης στοιχείων σε λίστες

```
>>> fruits = []
>>> fruits.append("apple")
>>> print(fruits)
['apple']
>>> fruits.append("orange")
>>> print(fruits)
['apple', 'orange']
>>> fruits.insert(1, "cherry")
>>> print(fruits)
['apple', 'cherry', 'orange']
>>> fruits.extend(["grape", "apricot"])
>>> print(fruits)
['apple', 'cherry', 'orange', 'grape', 'apricot']
```

# Μέθοδοι αφαίρεσης στοιχείων από λίστες

- `.pop()`: αφαιρεί το τελευταίο στοιχείο της λίστας και το επιστρέφει (η τιμή που αφαιρείται και επιστρέφεται μπορεί να προσδιοριστεί με βάση τη θέση της, αν περάσουμε τη θέση ως όρισμα)
- `.remove(x)`: αφαιρεί την πρώτη εμφάνιση του στοιχείου `x`, αν το `x` δεν υπάρχει στην λίστα τότε προκαλείται σφάλμα
- Παραδείγματα κλήσης μεθόδων αφαίρεσης στοιχείων από λίστες

```
>>> numbers = [10, 20, 30, 40, 20]
>>> x = numbers.pop(2)
>>> x
30
>>> numbers
[10, 20, 40, 20]
>>> y = numbers.pop()
>>> y
20
>>> numbers
[10, 20, 40]
>>> numbers.remove(20)
>>> numbers
[10, 40]
>>> numbers.remove(99)
Traceback (most recent call last):
  File "<python-input-104>", line 1, in <module>
    numbers.remove(99)
    ~~~~~^~~~~~
ValueError: list.remove(x): x not in list
```

## Άλλες μέθοδοι λιστών

- `.count(x)`: επιστρέφει πόσες φορές υπάρχει το `x` στη λίστα
- `.index(x)`: επιστρέφει την πρώτη θέση εμφάνισης του `x`
- Παραδείγματα κλήσης των μεθόδων `.count()` και `.index()`

```
>>> numbers = [5, 10, 15, 10, 20, 10]
>>> numbers.count(10)
3
>>> numbers.index(10)
1
>>> numbers.index(10, 2) # πρώτη εμφάνιση του 10 από τη θέση 2 και μετά
3
>>> numbers.index(10, 1, 4) # πρώτη εμφάνιση του 10 από τη θέση 1 μέχρι τη θέση 3
1
```

# Διάσχιση λίστας

- Η διάσχιση μιας λίστας (iteration) μπορεί να γίνει με την while ή με την for (προτιμότερο)
- Χρήσιμη είναι και η συνάρτηση enumerate() έτσι ώστε να λαμβάνουμε και το δείκτη κάθε στοιχείου μαζί με το στοιχείο κατά τη διάσχιση μιας λίστας

```
numbers = [10, 20, 30, 40]

i = 0
while i < len(numbers):
    print(numbers[i])
    i += 1

print("-" * 20)

for num in numbers:
    print(num)

print("-" * 20)

for index, num in enumerate(numbers):
    print(f"Index {index}: {num}")
```

```
10
20
30
40
-----
10
20
30
40
-----
Index 0: 10
Index 1: 20
Index 2: 30
Index 3: 40
```

# Ταξινόμηση λίστας

- Η ταξινόμηση μιας λίστας μπορεί να γίνει είτε με τη συνάρτηση `sorted()` είτε με τη μέθοδο `.sort()`
  - Η `sorted(a_list)` επιστρέφει μια νέα ταξινομημένη λίστα, τα αρχικά δεδομένα της `a_list` δεν αλλάζουν
  - Η μέθοδος `.sort()` ταξινομεί τη λίστα για την οποία καλείται, είναι δηλαδή “in place” και επιστρέφει `None`
- Η ταξινόμηση σε αύξουσα σειρά είναι η προκαθορισμένη συμπεριφορά, αλλά μπορεί να αλλάξει σε φθίνουσα θέτοντας το προαιρετικό όρισμα `reverse` σε `True`, τόσο για την `sorted()` όσο και για την `.sort()`
- Επιπλέον, μπορεί να χρησιμοποιηθεί το προαιρετικό όρισμα `key` με τιμή μια συνάρτηση που εφαρμόζεται σε κάθε στοιχείο της λίστας και αυτό προσδιορίζει την τιμή με βάση την οποία θα γίνει η ταξινόμηση
  - για παράδειγμα για να γίνει ταξινόμηση μιας λίστας συμβολοσειρών, `a_list`, βάσει μήκους συμβολοσειρών μπορεί να χρησιμοποιηθεί η ακόλουθη εντολή: `a_list.sort(key=len)`



# Παραδείγματα ταξινόμησης λιστών

```
numbers = [3, 1, 4, 2]
numbers.sort() # ταξινομεί την ίδια την λίστα
print(numbers) # [1, 2, 3, 4]
```

```
numbers = [3, 1, 4, 2]
print(sorted(numbers)) # [1, 2, 3, 4]
print(numbers) # [3, 1, 4, 2]
```

```
words = ["apple", "banana", "kiwi"]
print(sorted(words, key=len)) # ['kiwi', 'apple', 'banana']
print(sorted(words, key=len, reverse=True)) # ['banana', 'apple', 'kiwi']
```

# Εμφωλευμένες λίστες

- Μια λίστα μπορεί να έχει ως στοιχείο της μια άλλη λίστα
- Με αυτό τον τρόπο μπορούν να δημιουργηθούν λίστες πολλών διαστάσεων καθώς και ιεραρχικά δομημένες δομές που μπορεί να ταιριάζουν καλύτερα στην επίλυση συγκεκριμένων προβλημάτων
- Η πρόσβαση σε στοιχεία της λίστας που βρίσκονται βαθύτερα εμφωλιασμένες γίνεται με τη χρήση πολλαπλών δεικτών
- Στο ακόλουθο παράδειγμα ορίζεται ένας δισδιάστατος πίνακας 3 x 3 μέσω μιας λίστας όπου κάθε στοιχείο της είναι μια άλλη λίστα, ακολουθεί τροποποίηση ενός στοιχείου του πίνακα

```
>>> a = [  
... [1,2,3],  
... [4,5,6],  
... [7,8,9]  
... ]  
>>> a  
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
>>> a[0][0]  
1  
>>> a[2][2]  
9  
>>> a[1][2]  
6  
>>> a[1][2] = 99  
>>> a  
[[1, 2, 3], [4, 5, 99], [7, 8, 9]]
```

# Αντιγραφή λίστας

- Η αντιγραφή λίστας είναι tricky
- Είναι σημαντικό να γίνει κατανοητό ότι η εντολή **new\_list = a\_list** δεν πραγματοποιεί αντιγραφή των περιεχομένων της λίστας a\_list στη νέα λίστα new\_list, αλλά δημιουργεί ένα νέο όνομα (ψευδώνυμο) για την λίστα a\_list, δηλαδή και τα δύο ονόματα a\_list και new\_list δείχνουν στα ίδια δεδομένα
- Για να γίνει πραγματική (αλλά ρηχή) αντιγραφή μιας λίστας σε μια άλλη θα πρέπει να δημιουργηθεί ένα αντίγραφο της λίστας όπως στη συνέχεια:  
**new\_list = a\_list[:]**

ή

**new\_list = a\_list.copy()**

- Παράδειγμα με δημιουργία ψευδωνύμου για λίστα και με δημιουργία αντιγράφου λίστας

```
a_list = [1, 2, 3]
```

```
# Αντιγραφή αναφοράς (δημιουργία ενός ψευδωνύμου = alias)
new_list = a_list
new_list.append(4)
print(a_list)    # [1, 2, 3, 4] => η a_list επηρεάζεται
print(new_list)  # [1, 2, 3, 4]
```

```
# Πραγματική αντιγραφή (με slice)
a_list = [1, 2, 3]
new_list = a_list[:]
new_list.append(4)
print(a_list)    # [1, 2, 3] => η a_list δεν επηρεάζεται
print(new_list)  # [1, 2, 3, 4]
```

# Πλειάδες

- Οι πλειάδες (tuples) είναι δομές δεδομένων που μπορούν να περιέχουν ετερογενή στοιχεία που διαχωρίζονται μεταξύ τους με κόμματα και περικλείονται σε παρενθέσεις
  - Στην πράξη, οι παρενθέσεις είναι προαιρετικές, εκτός από ειδικές περιπτώσεις, π.χ. η `print(1,2)` θα εκτυπώσει 2 ξεχωριστά ορίσματα, ενώ η `print((1,2))` θα εκτυπώσει την πλειάδα `(1,2)`
- Τα περιεχόμενα μιας πλειάδας δεν μπορούν να τροποποιούνται κατά την εκτέλεση του προγράμματος (στην ορολογία της Python οι πλειάδες είναι `immutable` = μη-μεταλλάξιμες)
- Οι πλειάδες είναι ακολουθίες (sequences) συνεπώς τα στοιχεία τους είναι διατεταγμένα (ordered)
- Η αναφορά στα στοιχεία των πλειάδων γίνεται μέσω δεικτών θέσης, όπως και με τις λίστες

- Παράδειγμα με δημιουργία πλειάδων, εκτύπωσή τους και πρόσβαση στα στοιχεία τους

```
my_tuple = (1, "apple", 3.14, True)
print(my_tuple) # (1, 'apple', 3.14, True)
```

```
another_tuple = 2, "banana", False
print(another_tuple) # (2, 'banana', False)
```

```
print(1, 2) # 1 2 => είναι δύο ξεχωριστά ορίσματα
print((1, 2)) # (1, 2) => είναι μία πλειάδα
```

```
print(my_tuple[0]) # 1
print(my_tuple[-1]) # True
```

```
my_tuple[1] = "orange" # TypeError
```

```
(1, 'apple', 3.14, True)
(2, 'banana', False)
1 2
(1, 2)
1
True
Traceback (most recent call last):
  File "/tempCodeRunnerFile.py", line 13, in <module>
    my_tuple[1] = "orange" # TypeError
TypeError: 'tuple' object does not support item assignment
```

# Πλειάδες

- Μια λίστα μπορεί να μετατραπεί σε πλειάδα με τη συνάρτηση `tuple()`
- Μια πλειάδα μπορεί να μετατραπεί σε λίστα με τη συνάρτηση `list()`
- Προσοχή στη δημιουργία πλειάδας ενός στοιχείου καθώς αν το στοιχείο απλά τοποθετηθεί σε παρενθέσεις θα θεωρηθεί ως μια απλή τιμή, ενώ αν τοποθετηθεί ένα κόμμα μετά την στοιχείο θα θεωρηθεί ως πλειάδα
  - `type((1))` → `<class 'int'>`
  - `type((1,))` → `<class 'tuple'>`
- Οι πλειάδες είναι αποδοτικότερες από πλευράς χώρου που καταλαμβάνουν και ταχύτητας εκτέλεσης από τις λίστες, λόγω στατικότητας

- Παραδείγματα με μετατροπή από λίστα σε πλειάδα και αντίστροφα καθώς και με πλειάδες ενός στοιχείου

```
my_list = [1, 2, 3]
my_tuple = tuple(my_list) # Μετατροπή λίστας σε πλειάδα
print(my_tuple) # (1, 2, 3)

new_list = list(my_tuple) # Μετατροπή πλειάδας σε λίστα
print(new_list) # [1, 2, 3]

single_value = (5) # απλή τιμή, δεν είναι πλειάδα
single_tuple = (5,) # πλειάδα ενός στοιχείου
print(single_value) # 5
print(single_tuple) # (5,)
print(type(single_value)) # <class 'int'>
print(type(single_tuple)) # <class 'tuple'>
```

# Τεμαχισμός και εμφωλευμένες πλειάδες

- Όπως και στις λίστες, οι πλειάδες υποστηρίζουν τεμαχισμό (slicing) και εμφωλιασμό (πλειάδα μέσα σε πλειάδα)
- Παράδειγμα με εμφωλευμένες πλειάδες και με slicing

```
nested = (1, (2, 3), (4, 5, 6), 7)

print(nested[0])      # 1
print(nested[1])      # (2, 3)
print(nested[1][0])   # 2 => πρόσβαση σε στοιχείο του nested tuple

print(nested[1:3])    # ((2, 3), (4, 5, 6))
print(nested[:2])     # (1, (2, 3))
print(nested[:,2])    # (1, (4, 5, 6)) => κάθε δεύτερο στοιχείο

# Slicing μέσα σε nested tuple
print(nested[2][1:])  # (5, 6)
```

# Σύνολα

- Τα σύνολα (sets) είναι δομές δεδομένων που μπορούν να περιέχουν ετερογενή στοιχεία, **χωρίς διπλότυπα**, που χωρίζονται μεταξύ τους με κόμματα και περικλείονται σε αγκύλες τύπου { και }
- Στα σύνολα δεν υπάρχει διάταξη μεταξύ των στοιχείων τους
- Τα σύνολα είναι mutable (μπορούν να προστεθούν ή να αφαιρεθούν στοιχεία σε ένα σύνολο)
- Η δημιουργία ενός συνόλου με ένα μόνο στοιχείο γίνεται με τη συνάρτηση `set()` και όχι με τις αγκύλες
  - `a = set([99])` → δημιουργεί το σύνολο {99}
  - `a = {99}` → δεν δημιουργεί το σύνολο {99}, αλλά ένα λεξικό (βλ. στη συνέχεια περί λεξικών)

# Μέθοδοι συνόλων

- `.add(x)`: προσθέτει στο σύνολο ένα στοιχείο `x`, αν το στοιχείο ήδη υπάρχει τότε το σύνολο μένει όπως πριν
- `.update(x)`: το όρισμα `x` μπορεί να είναι λίστα, πλειάδα ή σύνολο και κάθε στοιχείο του ορίσματος προστίθεται στο σύνολο
- `.discard(x)`: διαγράφει από το σύνολο το στοιχείο `x`, αν δεν υπάρχει δεν προκαλείται σφάλμα
- `.remove(x)`: διαγράφει από το σύνολο το στοιχείο `x`, αν το στοιχείο `x` δεν υπάρχει τότε προκαλείται σφάλμα

- Παραδείγματα με κλήση μεθόδων συνόλων

```
my_set = {1, 2, 3}
print("Αρχικό σύνολο:", my_set)  # {1, 2, 3}

my_set.add(4)
my_set.add(2)  # ήδη υπάρχει, δεν αλλάζει τίποτα
print("Με add:", my_set)  # {1, 2, 3, 4}

# update(): προσθήκη πολλαπλών στοιχείων από λίστα/tuple/set
my_set.update([5, 6], (7, 8), {1, 9})
print("Με update:", my_set)  # {1, 2, 3, 4, 5, 6, 7, 8, 9}

# discard(): διαγραφή στοιχείου, χωρίς σφάλμα αν δεν υπάρχει
my_set.discard(3)
my_set.discard(100)  # δεν υπάρχει, δεν προκαλεί σφάλμα
print("Με discard:", my_set)  # {1, 2, 4, 5, 6, 7, 8, 9}

# remove(): διαγραφή στοιχείου, σφάλμα αν δεν υπάρχει
my_set.remove(2)
print("Με remove:", my_set)  # {1, 4, 5, 6, 7, 8, 9}
my_set.remove(100)  # KeyError
```



# Πράξεις σε σύνολα

- Με τα σύνολα της Python μπορούμε να κάνουμε πράξεις, όπως στα σύνολα των μαθηματικών (ένωση, τομή, κ.λπ.)
- `s1.union(s2)`: επιστρέφει ένα νέο σύνολο που είναι η ένωση των συνόλων `s1` και `s2`
- `s1.intersection(s2)`: επιστρέφει ένα νέο σύνολο που είναι η τομή των συνόλων `s1` και `s2`
- `s1.difference(s2)`: επιστρέφει ένα νέο σύνολο που περιέχει τα στοιχεία του συνόλου `s1` που δεν ανήκουν στο σύνολο `s2`
- `s1.symmetric_difference(s2)`: επιστρέφει ένα νέο σύνολο που περιέχει τα στοιχεία του συνόλου `s1` που δεν ανήκουν στο σύνολο `s2` και τα στοιχεία του `s2` που δεν ανήκουν στο σύνολο `s1` (δηλαδή τα στοιχεία που ανήκουν σε ένα από τα δύο σύνολα, αλλά όχι και στα δύο)

# Παράδειγμα με πράξεις συνόλων

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}
print(f"{s1=}")
print(f"{s2=}")
print("Ένωση:", s1.union(s2)) # {1, 2, 3, 4, 5, 6}
print("Τομή:", s1.intersection(s2)) # {3, 4}
print("Διαφορά s1 - s2:", s1.difference(s2)) # {1, 2}
print("Διαφορά s2 - s1:", s2.difference(s1)) # {5, 6}
print("Συμμετρική διαφορά:", s1.symmetric_difference(s2)) # {1, 2, 5, 6}
```

```
s1={1, 2, 3, 4}
s2={3, 4, 5, 6}
Ένωση: {1, 2, 3, 4, 5, 6}
Τομή: {3, 4}
Διαφορά s1 - s2: {1, 2}
Διαφορά s2 - s1: {5, 6}
Συμμετρική διαφορά: {1, 2, 5, 6}
```

# Λεξικά

- Τα λεξικά (dictionaries) είναι μια πολύ χρήσιμη δομή δεδομένων στην οποία αποθηκεύονται ζεύγη της μορφής `key: value`
- Τα κλειδιά (keys) πρέπει να είναι μια immutable τιμές (π.χ., συμβολοσειρά, ακέραιος, πλειάδα κ.α.), ενώ οι τιμές (values) μπορεί να είναι οτιδήποτε
- Η δημιουργία ενός λεξικού γίνεται με αγκύλες τύπου `{` και `}` και με ζεύγη της μορφής `key:value` που χωρίζονται μεταξύ τους με κόμματα
- Τα στοιχεία των λεξικών έχουν διάταξη που προκύπτει από τη χρονική στιγμή εισαγωγής κάθε νέου ζεύγους σε αυτά (αυτό ισχύει από την Python 3.7 και μετά), οπότε η διάσχισή τους δίνει κάθε φορά τα ίδια αποτελέσματα
- Μπορούν να προστίθενται νέα ζεύγη `key:value` ή να αφαιρούνται υπάρχοντα ζεύγη με βάση το κλειδί από ένα λεξικό
- Η αναφορά σε ένα στοιχείο ενός λεξικού γίνεται με την τιμή κλειδιού
- Τα κλειδιά πρέπει να είναι μοναδικά σε ένα λεξικό

# Πρόσβαση, τροποποίηση, διαγραφή στοιχείων λεξικών

- Οι αγκύλες [] χρησιμοποιούνται για την προσπέλαση και την τροποποίηση λεξικών, για παράδειγμα:  
    person["age"] = 25  
    person["city"] = "Athens"
- Αν με τις αγκύλες [] γίνει αναφορά σε κλειδί που δεν υπάρχει τότε προκαλείται σφάλμα KeyError
- Ο τελεστής del χρησιμοποιείται για τη διαγραφή ενός ζεύγους key:value με βάση ένα κλειδί, για παράδειγμα:  
    del person["city"]

- Παράδειγμα με δημιουργία λεξικού και εισαγωγή/διαγραφή/προσπέλαση στοιχείων του λεξικού

```
person = {}
person["name"] = "Alice"
person["age"] = 25
person["city"] = "Athens"
print("Αρχικό λεξικό:", person)
# {'name': 'Alice', 'age': 25, 'city': 'Athens'}
# Τροποποίηση στοιχείου του λεξικού
person["age"] = 26
print("Μετά την τροποποίηση:", person)
# {'name': 'Alice', 'age': 26, 'city': 'Athens'}
# Πρόσβαση σε στοιχείο του λεξικού
print("Η ηλικία είναι:", person["age"]) # 26
# Διαγραφή στοιχείου του λεξικού με del
del person["city"]
print("Μετά τη διαγραφή του city:", person)
# {'name': 'Alice', 'age': 26}
print(person["city"]) # KeyError
```

# Μέθοδοι λεξικών

## Χρήσιμες μέθοδοι για διάσχιση λεξικών

- `.keys()`: επιστρέφει μια ακολουθία με τα κλειδιά του λεξικού
- `.values()`: επιστρέφει μια ακολουθία με τις τιμές του λεξικού
- `.items()`: επιστρέφει μια ακολουθία με ζεύγη (πλειάδες 2 στοιχείων) της μορφής κλειδί, τιμή
- Η διάσχιση λεξικών μπορεί να γίνει εύκολα με την εντολή `for` και τη χρήση των μεθόδων `.keys()`, `.values()`, `.items()`

## Άλλες μέθοδοι λεξικών

- `.get(k)`: επιστρέφει την τιμή του κλειδιού `k`, ή την τιμή `None` αν το `k` δεν υπάρχει στο λεξικό
- `.pop(k)`: αφαιρεί το ζεύγος με κλειδί `k`, επιστρέφει την τιμή του ζεύγους
- `.update(a_dict)`: ενημερώνει το λεξικό με βάση τα ζεύγη `key:value` που υπάρχουν στο όρισμα `a_dict`
- `.clear()`: αφαιρεί όλα τα ζεύγη από το λεξικό

# Παράδειγμα διάσχισης λεξικού (3 τρόποι)

```
person = {"name": "Nikolaos", "age": 26, "city": "Athens"}

print("Keys:", person.keys())
# dict_keys(['name', 'age', 'city'])

print("Values:", person.values())
# dict_values(['Nikolaos', 26, 'Athens'])

print("Items:", person.items())
# dict_items([('name', 'Nikolaos'), ('age', 26), ('city', 'Athens')])

print("-" * 20)

for key in person.keys():
    print(key)

print("-" * 20)

for value in person.values():
    print(value)

print("-" * 20)

for key, value in person.items():
    print(f"{key}: {value}")
```

```
Keys: dict_keys(['name', 'age', 'city'])
Values: dict_values(['Nikolaos', 26, 'Athens'])
Items: dict_items([('name', 'Nikolaos'), ('age',
26), ('city', 'Athens')])

-----
name
age
city
-----
Nikolaos
26
Athens
-----
name: Nikolaos
age: 26
city: Athens
```

# Παράδειγμα κλήσης διαφόρων μεθόδων λεξικών

```
person = {"name": "Νικόλαος", "age": 26}
```

```
# get(): επιστρέφει τιμή κλειδιού ή None (ή default αν δοθεί)  
print("Όνομα:", person.get("name")) # Νικόλαος  
print("Χώρα:", person.get("country")) # None  
print("Χώρα (με default):", person.get("country", "Ελλάδα")) # Ελλάδα
```

```
# pop(): αφαιρεί και επιστρέφει την τιμή ενός κλειδιού  
age = person.pop("age")  
print("Ηλικία που αφαιρέθηκε:", age) # 26  
print("Λεξικό μετά το pop:", person) # {'name': 'Νικόλαος'}
```

```
# update(): ενημερώνει/προσθέτει ζεύγη από άλλο λεξικό  
person.update({"city": "Αθήνα", "job": "Μηχανικός"})  
print("Μετά το update:", person)  
# {'name': 'Νικόλαος', 'city': 'Αθήνα', 'job': 'Μηχανικός'}
```

```
# clear(): διαγράφει όλα τα ζεύγη  
person.clear()  
print("Μετά το clear:", person) # {}
```

```
Όνομα: Νικόλαος  
Χώρα: None  
Χώρα (με default): Ελλάδα  
Ηλικία που αφαιρέθηκε: 26  
Λεξικό μετά το pop: {'name': 'Νικόλαος'}  
Μετά το update: {'name': 'Νικόλαος', 'city': 'Αθήνα', 'job': 'Μηχανικός'}  
Μετά το clear: {}
```

# Προσπέλαση στοιχείων λεξικού με βάση το κλειδί τους

- Η προσπέλαση ενός στοιχείου σε ένα λεξικό μπορεί να γίνει με το `[]` για ένα δεδομένο κλειδί, αν το κλειδί δεν υπάρχει στο λεξικό τότε προκαλείται `KeyError`
  - π.χ. `person["name"]`
- Εναλλακτικά, για ασφαλή πρόσβαση στην τιμή ενός ζεύγους κλειδί:τιμή, μπορεί να χρησιμοποιηθεί η μέθοδος `.get("key_value")` που επιστρέφει `None` αν το κλειδί δεν υπάρχει
  - π.χ. `person.get("name")`
- Μπορεί μάλιστα να δεχθεί ένα επιπλέον όρισμα που να ορίζει την προκαθορισμένη τιμή που θα επιστραφεί αν το κλειδί δεν βρεθεί στο λεξικό
  - π.χ. `person.get("name", "John Doe")`
- Παράδειγμα εντοπισμού πλήθους εμφάνισης χαρακτήρων σε ένα κείμενο, α) με `[]` και β) με `.get()`

```
phrase = "hello world"
```

```
# Με χρήση [] (απαιτεί αρχικοποίηση)
```

```
count_brackets = {}
```

```
for c in phrase:
```

```
    if c not in count_brackets:
```

```
        count_brackets[c] = 0
```

```
    count_brackets[c] += 1
```

```
print("Using []:", count_brackets)
```

```
# Με χρήση.get() (απλούστερο)
```

```
count_get = {}
```

```
for c in phrase:
```

```
    count_get[c] = count_get.get(c, 0) + 1
```

```
print("Using .get():", count_get)
```

```
Using []: {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```

```
Using .get(): {'h': 1, 'e': 1, 'l': 3, 'o': 2, ' ': 1, 'w': 1, 'r': 1, 'd': 1}
```



# Περιφραστικές λίστες, σύνολα, λεξικά

- Μια περιφραστική λίστα ή συνοπτική λίστα ή comprehension είναι ένας συμπαγής τρόπος για τη δημιουργία λιστών (ομοίως για σύνολα και λεξικά)
- Συνδυάζει ένα βρόχο και μια έκφραση σε μια γραμμή κώδικα
- Συχνά είναι ευκολότερες στην ανάγνωση και ταχύτερες στη συγγραφή σε σχέση με τις εντολές for
- Βασική σύνταξη:  
[expression for item in iterable if condition]
- όπου:
  - **expression**: η έκφραση που δημιουργεί τις τιμές που θα εισαχθούν στη λίστα
  - **item**: η μεταβλητή που αναπαριστά κάθε στοιχείο που θα επεξεργαστεί προκειμένου να δημιουργηθεί η παραπάνω έκφραση
  - **iterable**: η ακολουθία που θα διασχισθεί
  - **condition**: ένα φίλτρο για τα στοιχεία της ακολουθίας που θα συμπεριληφθούν στην παραγωγή της νέας λίστας

- Παραδείγματα με δημιουργία περιφραστικών λιστών, συνόλων, λεξικών

```
>>> squares = [x**2 for x in range(5)]
>>> squares
[0, 1, 4, 9, 16]
>>> unique = {x % 3 for x in range(10)}
>>> unique
{0, 1, 2}
>>> squares_dict = {x: x**2 for x in range(5)}
>>> squares_dict
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
>>> even_squares = [x**2 for x in range(10) if x % 2 == 0]
>>> even_squares
[0, 4, 16, 36, 64]
```

# Συνδυασμός λιστών με τη συνάρτηση zip

- Η συνάρτηση `zip()` συνδυάζει 2 ή περισσότερες λίστες (ή άλλα διασχίσιμα αντικείμενα) για τη δημιουργία μιας νέας λίστας όπου κάθε στοιχείο της νέας λίστας προκύπτει ως πλειάδα που περιέχει στοιχεία που βρίσκονται στην ίδια θέση από τις λίστες που συνδυάζονται
- Η `zip()` σταματά στη λίστα με το μικρότερο μήκος

```
>>> names = ["Γιάννης", "Ελένη", "Νίκος"]
>>> scores = [90, 85, 78]
>>> pairs = list(zip(names, scores))
>>> pairs
[('Γιάννης', 90), ('Ελένη', 85), ('Νίκος', 78)]
>>> for name, score in zip(names, scores):
...     print(f"{name=} {score=}")
...
name='Γιάννης' score=90
name='Ελένη' score=85
name='Νίκος' score=78
```

## Άσκηση #7 (λίστες και λεξικά): Εκφώνηση

- Γράψτε ένα πρόγραμμα που να διαχειρίζεται το ευρετήριο (inventory) ενός χαρακτήρα σε ένα video game. Το inventory να αποθηκεύεται σε μια λίστα. Ειδικότερα, υλοποιήστε συναρτήσεις για: προσθήκη αντικειμένου (το ίδιο αντικείμενο μπορεί να υπάρχει πολλές φορές στο inventory), χρήση αντικειμένου (που αφαιρεί 1 μονάδα του αντικειμένου από το inventory), και εμφάνιση inventory σε μορφή:

```
Inventory summary:  
sword: 1  
potion: 2  
shield: 1  
Full inventory: ['sword', 'potion', 'shield', 'potion']
```

# Άσκηση #7: Λύση

```
inventory = ["sword", "potion", "potion", "shield"]
```

```
def add_item(inv, item):  
    inv.append(item)  
    print("Added:", item)
```

```
def use_item(inv, item):  
    """Χρησιμοποιεί (αφαιρεί) ένα αντικείμενο από το inventory αν υπάρχει."""  
    if item in inv:  
        inv.remove(item)  
        print("Used:", item)  
    else:  
        print(f"No {item} left to use.")
```

```
def show_inventory(inv):  
    """Εμφανίζει το inventory και πόσες φορές εμφανίζεται κάθε αντικείμενο."""  
    print("Inventory summary:")  
    counted = {}  
    for item in inv:  
        counted[item] = counted.get(item, 0) + 1  
    for item, count in counted.items():  
        print(f"{item}: {count}")  
    print("Full inventory:", inv)
```

```
add_item(inventory, "potion") # Προσθήκη potion  
use_item(inventory, "potion") # Χρήση ενός potion  
show_inventory(inventory)    # Εμφάνιση inventory
```

```
Added: potion  
Used: potion  
Inventory summary:  
sword: 1  
potion: 2  
shield: 1  
Full inventory: ['sword', 'potion', 'shield', 'potion']
```

# Εγκατάσταση βιβλιοθηκών, δημιουργία ιδεατών περιβαλλόντων, notebooks

Libraries, virtual environments, Jupyter Notebook, Jupyter Lab, Google colab

# Εγκατάσταση βιβλιοθηκών με το pip

- Το pip είναι ένα πρόγραμμα που πραγματοποιεί εγκατάσταση πακέτων, πρόκειται δηλαδή για έναν διαχειριστή πακέτων (package manager)
  - Το pip συνδέεται με το PyPI (Python Package Index) για τη λήψη των πακέτων
- Επιτρέπει την εγκατάσταση, ενημέρωση και απεγκατάσταση πακέτων τρίτων δημιουργών (third-party)
- Αν και μπορεί να χρησιμοποιηθεί το pip για εγκατάσταση βιβλιοθηκών στη βασική εγκατάσταση, υπάρχουν πλεονεκτήματα από τη χρήση των λεγόμενων ιδεατών περιβαλλόντων

Εντολή	Περιγραφή
<code>pip install package_name</code>	Εγκατάσταση του πακέτου <code>package_name</code>
<code>pip uninstall package_name</code>	Απεγκατάσταση του πακέτου <code>package_name</code>
<code>pip list</code>	Εμφάνιση λίστας των εγκατεστημένων πακέτων
<code>pip show package_name</code>	Παρουσίαση πληροφοριών όπως όνομα, έκδοση, προαπαιτούμενα πακέτα, περιγραφή κ.α.
<code>pip install --upgrade package_name</code>	Ενημέρωση της εγκατάστασης του πακέτου <code>package_name</code>
<code>pip freeze</code>	Εκτυπώνει λίστα με όλα τα εγκατεστημένα πακέτα και τις εκδόσεις τους, συχνά χρησιμοποιείται ως: <b><code>pip freeze &gt; requirements.txt</code></b> έτσι ώστε να δημιουργηθεί το αρχείο <code>requirements.txt</code> που στη συνέχεια μπορεί να χρησιμοποιηθεί για να αναπαραχθεί ακριβώς το ίδιο περιβάλλον σε μια άλλη εγκατάσταση με την εντολή: <b><code>pip install -r requirements.txt</code></b>

# Ιδεατά περιβάλλοντα

- Ένα ιδεατό περιβάλλον (virtual environment) είναι μια ανεξάρτητη εγκατάσταση της συγκεκριμένης έκδοσης της Python και επιπλέον βιβλιοθηκών που μπορεί να δημιουργείται, να χρησιμοποιείται ως περιβάλλον της Python και να απορρίπτεται όταν πλέον δεν χρειάζεται
- Τα virtual environment επιτρέπουν στη βασική εγκατάσταση της Python να παραμένει «καθαρή», δηλαδή να μην εγκαθίστανται σε αυτή άλλες εξωτερικές βιβλιοθήκες
- Επιτρέπουν την αυτοματοποίηση της διαδικασίας δημιουργίας μια συγκεκριμένης εγκατάστασης Python και βιβλιοθηκών (π.χ. Python 3.11 και βιβλιοθήκες numpy, pandas, scikit-learn, πιθανά σε συγκεκριμένες εκδόσεις όπως 1.3.1 για το scikit-learn)
- Υπάρχουν πολλοί τρόποι με τους οποίους μπορούν να δημιουργηθούν virtual environments, τρία εναλλακτικά δημοφιλή εργαλεία για να γίνει αυτό είναι το **venv**, το **conda** (απαιτεί εγκατάσταση του Anaconda ή του miniconda - <https://www.anaconda.com/download/success>), και το **uv** (απαιτεί εγκατάσταση - <https://docs.astral.sh/uv/>)

# Δημιουργία virtual environment με το venv

## Windows

- Με τις ακόλουθες εντολές, από τη γραμμή εντολών (command prompt) των Windows, δημιουργείται και ενεργοποιείται ένα virtual environment με όνομα myenv (μέσα στον τρέχοντα κατάλογο), γίνεται εγκατάσταση του πακέτου pandas και του πακέτου jupyter notebook
  - > python -m venv myenv
  - > myenv\Scripts\activate.bat
  - > pip install pandas
  - > pip install notebook

## MacOS / Linux

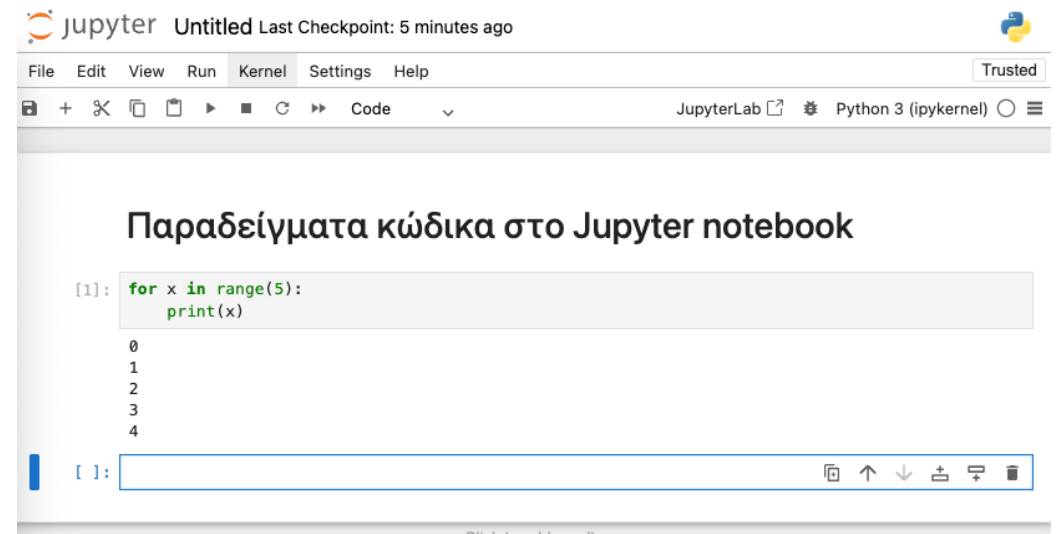
- Με τις ακόλουθες εντολές, από την κονσόλα (shell) στο MacOS και στο Linux, δημιουργείται και ενεργοποιείται ένα virtual environment με όνομα myenv (μέσα στον τρέχοντα κατάλογο), γίνεται εγκατάσταση του πακέτου pandas και του πακέτου jupyter notebook
  - \$ python -m venv myenv
  - \$ source myenv/bin/activate
  - \$ pip install pandas
  - \$ pip install notebook

Η απενεργοποίηση του virtual environment γίνεται με την εντολή deactivate



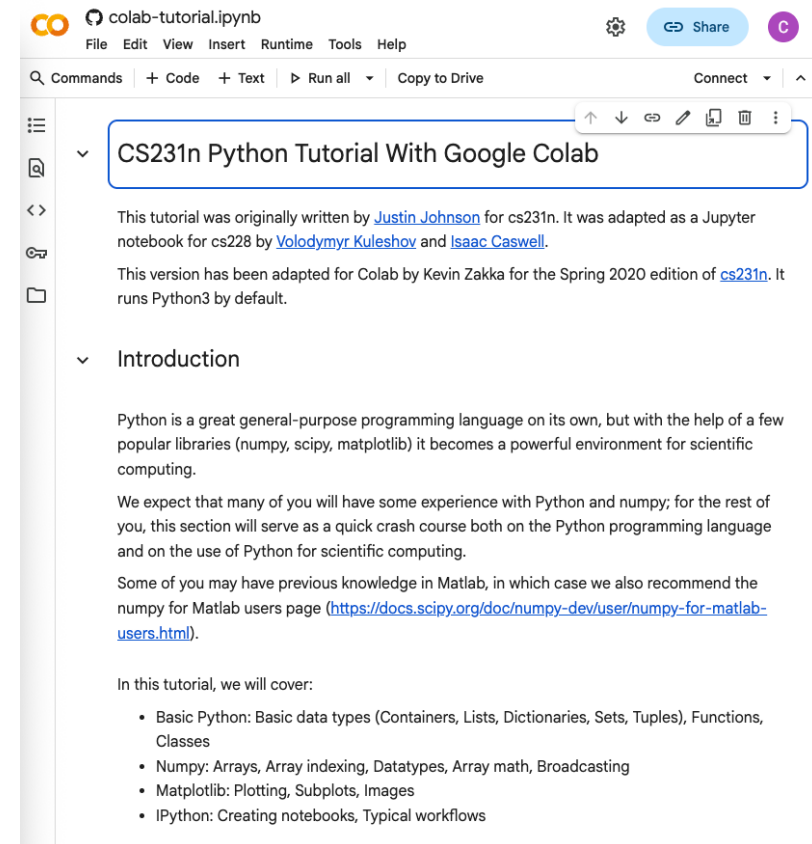
# Jupyter notebooks

- Η εκκίνηση του notebook γίνεται με την εντολή:  
\$ jupyter notebook
- Στην ιστοσελίδα που θα ανοίξει στο <http://localhost:8888/tree> μπορεί να γίνει η δημιουργία ενός νέου notebook με Python 3 (ipykernel), οπότε ανοίγει μια νέα σελίδα <http://localhost:8888/notebooks/Untitled.ipynb>
- Μπορεί να γίνει μετονομασία του notebook και να συμπληρωθούν κελιά με μορφοποιημένο κείμενο (με markdown) και με κώδικα Python



# Google Colab

- Το Google Colab (Colaboratory) είναι ένα δωρεάν, cloud-based Jupyter notebook περιβάλλον που επιτρέπει τη συγγραφή κώδικα Python απευθείας στον φυλλομετρητή χωρίς να χρειάζεται οποιαδήποτε εγκατάσταση
- Ο χρήστης θα πρέπει να έχει λογαριασμό στην Google για να μπορεί να χρησιμοποιήσει το Colab
- Αποκτήστε πρόσβαση στο Colab μέσω του <https://colab.research.google.com/>



<https://colab.research.google.com/github/cs231n/cs231n.github.io/blob/master/python-colab.ipynb>