Ελληνικό Μεσογειακό Πανεπιστήμιο
Σχολή Μηχανικών / Τμήμα Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών

# HMY01K06
# Επιστημονικός Προγραμματισμός με Python

## Διάλεξη Τρίτη
## Τελεστές, Είσοδος / Έξοδος
## Δομές Επιλογής και Επανάληψης
## (τα βασικά)

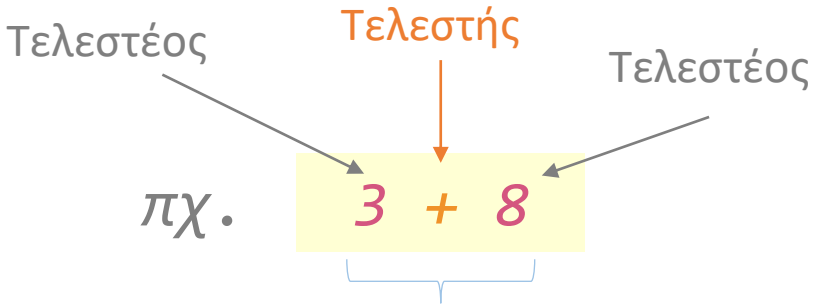
*Φθινόπωρο 2025*

# Εκφράσεις και Τελεστές

Ενας *τελεστής* (*operator*) είναι ένα σύμβολο το οποίο παριστάνει μια πράξη

*πχ.* `+` `-` `*` `%` `**`

Ο τελεστής επιδρά πάνω σε έναν ή περισσότερους *τελεστέους* (*operands*) τους οποίους συνδυάζει σε *εκφράσεις* (expressions) και παράγει ένα αποτέλεσμα



# Εκφράσεις και Τελεστές

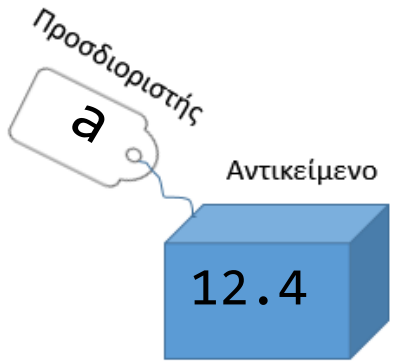
Οι τελεστέοι στην Python είναι *κάθε είδους αντικείμενα* της γλώσσας που εμφανίζονται:

- είτε "*αυτοπροσώπως*" (*κυριολεκτήματα* – *literals*)

  πχ. `12.4` `'Hello'` `[1,2,3]` `True`

- είτε "*μέσω εκπροσώπων*" (*προσδιοριστές* – *identifiers*)

  πχ. `a`, `x1`, `Sum`, `MesosOros`




*Δεν είναι ασυνήθιστο να αποκαλούμε τους προσδιοριστές και "μεταβλητές" κατ' αναλογία με άλλες παραδοσιακές γλώσσες, όμως φυσικά ξέρουμε ότι πρόκειται για κάτι εντελώς διαφορετικό*

# Εκφράσεις και Τελεστές

Οι *εκφράσεις* (expressions) προκύπτουν από συνδυασμό τελεστέων και τελεστών

*Απλή έκφραση*

```
3   +   5
```

Τελεστέος Τελεστής Τελεστέος

*Πιο σύνθετη έκφραση*

```
D = B**2 - 4 * A * C
```

Τελεστές

Στην απλούστερη δυνατή μορφή της μια έκφραση μπορεί να αποτελείται *μόνο από έναν τελεστέο και τίποτε άλλο*

*Απλούστερη δυνατή μορφή έκφρασης*

```
12.4
```

Τελεστέος που εμφανίζεται "αυτοπροσώπως" (σε μορφή κυριολεκτήματος)

*Απλούστερη δυνατή μορφή έκφρασης*

```
Athroisma
```

Τελεστέος που εμφανίζεται μέσω ενός προσδιοριστή του

# Οι τελεστές της Python συνοπτικά

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Αριθμητικοί Τελεστές *Arithmetic Operators* | `+` | `-` | `*` | `/` | `//` | `%` | `**` |
| Τελεστές Σύγκρισης *Relational Operators* | `<` | `<=` | `>` | `>=` | `==` | `!=` | |
| Λογικοί Τελεστές *Logical Operators* | `and` | `or` | `not` | | | | |
| Δυαδικοί Τελεστές *Bitwise(*) Operators* | `&` AND | `\|` OR | `^` XOR | `~` NOT | `<<` LShift | `>>` RShift | |
| Τελεστής Εκχώρησης *Assignment Operator* | `=` | | | | | | |
| Τελεστές Επαυξημένης Εκχώρησης *Augmented Assignment Operators* | `+=` | `-=` | `*=` | `/=` | `//=` | `%=` | `**=` |
| Τελεστές ελέγχου Μέλους *Membership Operators* | `in` | `not in` (πχ. `4 in [2,3,6]` ⇨ `False`) | | | | | |
| Τελεστές ελέγχου Ταυτότητας *Identity Operators* | `is` | `is not` (πχ. `"a" is 'a'` ⇨ `True`) | | | | | |

(*) Bitwise : κάτι που ισχύει σε επίπεδο bits (δυαδικό επίπεδο)

# Αριθμητικοί Τελεστές

- Βασικές αριθμητικές πράξεις ( + - * / )

```
>>> 11+4
15
>>> 11-4
7
>>> 11*4
44
>>> 11/4
2.75
```

Εκτός από αυτούς τους τελεστές υπάρχει και ο μοναδιαίος τελεστής ( - ) που όταν μπει μπροστά από έναν τελεστέο μας επιστρέφει την *αντίθετη* τιμή του

---

- Ακέραια διαίρεση ( // ) και Ακέραιο υπόλοιπο ( % )

```
>>> 11//4    (Μας δίνει 2, δηλαδή το ακέραιο πηλίκο)
2
>>> 11%4     (Μας δίνει 3, δηλαδή το ακέραιο υπόλοιπο)
3
```

| 11 | 4 |
|---|---|
| 3 | 2 |

---

- Ύψωση σε δύναμη (**)

```
>>> 35**4
1500625
```

# Τελεστές Σύγκρισης

- Εναλλακτικά ονομάζονται και σχεσιακοί τελεστές *(relational operators)*
- Χρησιμοποιούνται για *σύγκριση δυο τιμών* που μπορεί να είναι *απλές σταθερές* ή να προκύπτουν ως *αποτέλεσμα πιο σύνθετων εκφράσεων*
- Ανάλογα με τις τιμές και το είδος της σύγκρισης, επιστρέφουν `True` ή `False`

| Τελεστής | | Παράδειγμα | | |
|---|---|---|---|---|
| `<` | Μικρότερο | Αν `a=3 , b=5` τότε η έκφραση | `a < b` | επιστρέφει `True` |
| `<=` | Μικρότερο ή ίσον | Αν `a=3 , b=5` τότε η έκφραση | `b <= a+2` | επιστρέφει `True` |
| `>` | Μεγαλύτερο | Αν `a=3 , b=5` τότε η έκφραση | `a > b-1` | επιστρέφει `False` |
| `>=` | Μεγαλύτερο ή ίσον | Αν `a=3 , b=5` τότε η έκφραση | `b >= a+2` | επιστρέφει `True` |
| `==` | Ίσον με | Αν `a=3 , b=5` τότε η έκφραση | `b-3 == a+1` | επιστρέφει `False` |
| `!=` | Όχι ίσον (διάφορο) | Αν `a=3 , b=5` τότε η έκφραση | `a+b/2 != -8` | επιστρέφει `True` |

# Λογικοί Τελεστές

- Χρησιμοποιούνται για να *συνδυάσουν λογικές εκφράσεις* και ανάλογα με τις τιμές των εκφράσεων που συνδυάζουν, επιστρέφουν `True` ή `False`

| P | Q | P and Q | P or Q | not P |
|---|---|---|---|---|
| True | True | True | True | False |
| True | False | False | True | False |
| False | True | False | True | True |
| False | False | False | False | True |

*Σημείωση*: Στην Python η τιμή `True` αντιστοιχεί στο `1` και η τιμή `False` στο `0`

Έτσι π.χ. `True + 2` μας δίνει 3, ενώ `False * 5` μας δίνει 0

# Δυαδικοί Τελεστές &, | , ^ και ~

Επιδρούν *απ' ευθείας πάνω στα bits* της δυαδικής μορφής των τιμών

Οι τελεστές **&** *(bitwise AND)* **|** *(bitwise OR)* **^** *(bitwise XOR)* και **~** *(bitwise NOT)* συγκρίνουν *ένα-ένα* τα *ζεύγη των bits* των δυο τιμών στη δυαδική μορφή τους

Παραδείγματα:

```
0  0  0  0  0  1  0  1    5
0  0  0  1  0  0  0  1    17
-------------------------------
0  0  0  0  0  0  0  1    5 & 17 επιστρέφει 1
```

Bitwise AND

128 64 32 16 8 4 2 1

```
0  0  0  0  0  1  0  1    5
0  0  0  1  0  0  0  1    17
-------------------------------
0  0  0  1  0  1  0  1    5 | 17 επιστρέφει 21
```

Bitwise OR

128 64 32 16 8 4 2 1

# Δυαδικοί Τελεστές &, | , ^ και ~

Επιδρούν *απ' ευθείας πάνω στα bits* της δυαδικής μορφής των τιμών

Οι τελεστές **&** *(bitwise AND)* **|** *(bitwise OR)* **^** *(bitwise XOR)* και **~** *(bitwise NOT)* συγκρίνουν *ένα-ένα* τα *ζεύγη των bits* των δυο τιμών στη δυαδική μορφή τους

Παραδείγματα:

Bitwise XOR

```
0  0  0  0  0  1  0  1    5
0  0  0  1  0  0  0  1    17
-------------------------------
0  0  0  1  0  1  0  0    5 ^ 17  επιστρέφει 20
```

*128 64 32 16 8 4 2 1*

Το **xor** επιστρέφει **1** όταν τα bits που συγκρίνει είναι διαφορετικά, αλλιώς επιστρέφει **0**

Bitwise NOT

```
0  0  0  0  0  1  0  1    5
-------------------------------
1  1  1  1  1  0  1  0    ~5  επιστρέφει -6
```

*Σημείωση*: το 1111010 είναι η δυαδική αναπαράσταση του -6 στην κωδικοποίηση "συμπληρώματος ως προς 2" (2's complement).

Γενικά το **~a** επιστρέφει **-a-1**

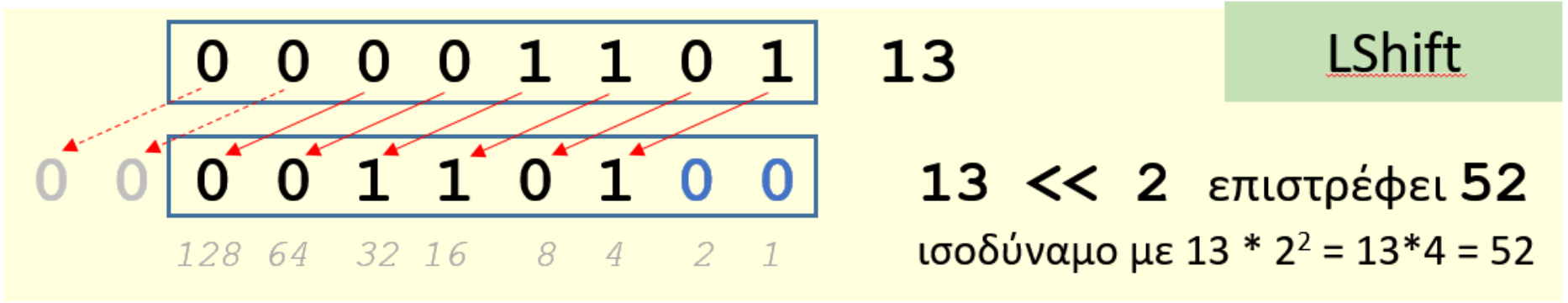
# Δυαδικοί Τελεστές << και >>

Επιδρούν *απ' ευθείας πάνω στα bits* της δυαδικής μορφής των τιμών

Ο τελεστής << *(Left Shift - LShift)* όταν εφαρμόζεται πάνω στην δυαδική αναπαράσταση μιας τιμής **a** μετακινεί *(σπρώχνει)* όλα τα bits της **n** θέσεις προς τα αριστερά (όπου **n** δοσμένος θετικός ακέραιος)

Τα bits που "ξεχειλίζουν" από το *αριστερό άκρο* της δυαδικής αναπαράστασης *χάνονται*, ενώ οι θέσεις που μένουν κενές στο *δεξί άκρο*, γεμίζουν με *μηδενικά*

Ισοδυναμεί στην ουσία με τον *πολλαπλασιασμό* του **a** επί $2^n$



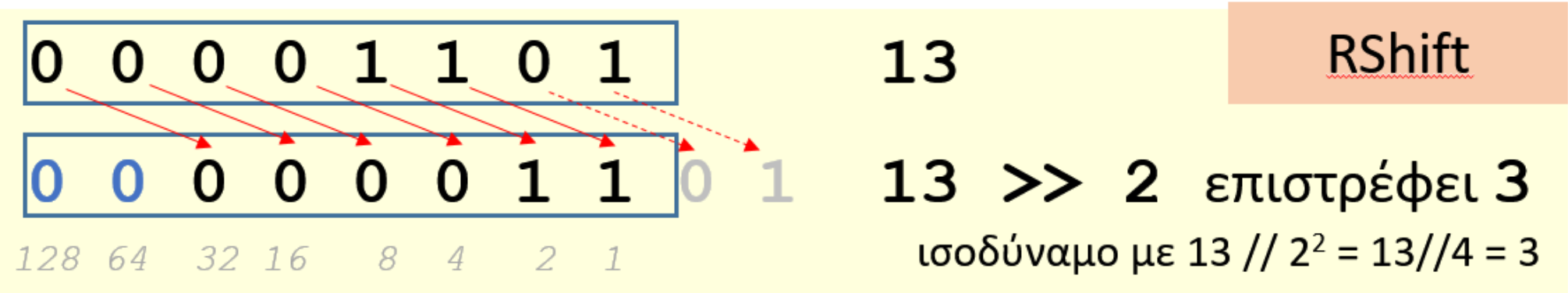
# Δυαδικοί Τελεστές << και >>

Επιδρούν *απ' ευθείας πάνω στα bits* της δυαδικής μορφής των τιμών

Αντίστοιχα ο τελεστής **>>** *(Right Shift - RShift)* μετακινεί *(σπρώχνει)* όλα τα bits **`n`** θέσεις προς τα δεξιά

Τα bits που "ξεχειλίζουν" από το *δεξί άκρο* της δυαδικής αναπαράστασης *χάνονται*, ενώ οι θέσεις που μένουν κενές στο *αριστερό άκρο*, γεμίζουν με *μηδενικά*

Ισοδυναμεί με το *ακέραιο πηλίκο της διαίρεσης* του **`a`** δια $2^n$



# Τελεστής Εκχώρησης

Ο τελεστής εκχώρησης στην Python είναι το ίσον ( = ) και συντάσσεται ως εξής

```
<προσδιοριστής> = <έκφραση>
```

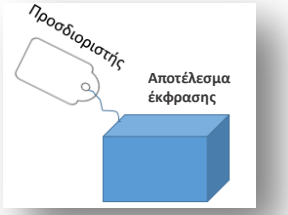
Στην βασική εκδοχή του

- το *αριστερό μέλος (αριστερός τελεστέος)* πρέπει οπωσδήποτε να είναι *προσδιοριστής*
- το *δεξί μέλος* μπορεί να είναι *οποιαδήποτε έκφραση*

Σε αντίθεση με τους περισσότερους τελεστές της Python, ο τελεστής εκχώρησης *δεν επιστρέφει κάποιο αποτέλεσμα*, αλλά αποτελεί *αυτόνομη πρόταση (εντολή)*

Κατά την εκτέλεση μιας εντολής εκχώρησης η *έκφραση* στο *δεξί μέλος* επαληθεύεται και το αποτέλεσμα *(που είναι και αυτό ένα αντικείμενο της Python)* "εκχωρείται(*)" στον *προσδιοριστή* στο *αριστερό μέλος*

(*) *στην πράξη η εντολή εκχώρησης επισημαίνει με εναν προσδιοριστή το αντικείμενο που προέκυψε από τον υπολογισμό της έκφρασης*



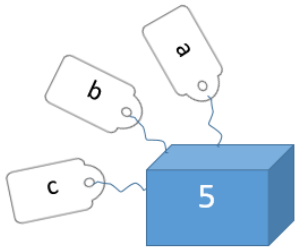
# Εναλλακτικοί τρόποι χρήσης του τελεστή εκχώρησης

- Εκχώρηση του *ίδιου αντικειμένου* σε *περισσότερους από έναν προσδιοριστές*

  *(στην ουσία επισήμανση του ίδιου αντικειμένου με περισσότερους από έναν προσδιοριστές)*

```
a = b = c = 5
```

οι προσδιοριστές a, b και c παίρνουν όλοι την τιμή 5




- Ταυτόχρονη εκχώρηση *διαφορετικών τιμών* σε *διαφορετικούς προσδιοριστές*

```
a, b, c = 5, 12.4, 'Hello'
```

Αποτελεί συντόμευση του `a = 5` `b = 12.4` `c = 'Hello'`

# Τελεστές Επαυξημένης Εκχώρησης

- Οι τελεστές *επαυξημένης εκχώρησης* στην Python είναι :

```
+=   -=   *=  /=   //=   %=   **=
```

- Σχηματίζονται τοποθετώντας έναν αριθμητικό τελεστή μπροστά από τον τελεστή εκχώρησης *(χωρίς ενδιάμεσα κενά)*
- Αποτελούν ένα *πιο σύντομο τρόπο* να περιγράψουμε *την εκτέλεση της αριθμητικής πράξης* ανάμεσα στους δυο τελεστέους *και την εκχώρηση του αποτελέσματος στον πρώτο τελεστή* (ο οποίος -όπως και στην απλή εκχώρηση- πρέπει να είναι πάντα κάποιος προσδιοριστής)

  *Παραδείγματα:*

  Το `a += b` ισοδυναμεί με την έκφραση `a = a + b`

  Το `a %= b` ισοδυναμεί με την έκφραση `a = a % b`

  Το `a **= b` ισοδυναμεί με την έκφραση `a = a ** b`

# Τελεστές Ελέγχου Μέλους

Είναι οι `in` και `not in` που ελέγχουν αν ένα στοιχείο *είναι μέλος (ανήκει) σε ένα σύνολο στοιχείων*

- Π.χ. η έκφραση `4 in [2,3,6]` επαληθεύεται σε `False` επειδή το στοιχείο 4 δεν ανήκει στη λίστα με στοιχεία τους αριθμούς 2, 3 και 6

# Τελεστές Ελέγχου Ταυτότητας

Είναι οι `is` και `is not` που ελέγχουν αν δυο στοιχεία αποτελούν ή όχι *ισοδύναμες εναλλακτικές μορφές του ίδιου αντικειμένου*

- Π.χ. η έκφραση `'Νερό' is "Νερό"` επαληθεύεται σε `True` επειδή και τα δυο στοιχεία αποτελούν εναλλακτικές μορφές διατύπωσης της ίδιας συμβολοσειράς

Η πιο συνηθισμένη χρήση αυτών των τελεστών είναι *για να ελέγξουν αν το αντικείμενο ενός προσδιοριστή είναι κάποιου συγκεκριμένου τύπου*

- Π.χ. η έκφραση `type(a) is int` επαληθεύεται σε `True` αν *κατά την παρούσα στιγμή* το **a** προσδιορίζει μια ακέραια τιμή, ή σε `False` στην αντίθετη περίπτωση

# Προτεραιότητα Τελεστών

Υψηλότερη

Χαμηλότερη

| | |
|---|---|
| `()` | Παρενθέσεις |
| `**` | Εκθέτες |
| `~` | Bitwise not |
| `*, /, //,%` | Πολλαπλασιασμός, Διαίρεση, Ακέραιο υπόλοιπο |
| `+, -` | Πρόσθεση, Αφαίρεση |
| `<<, >>` | Bitwise shifts |
| `&` | Bitwise AND |
| `^` | Bitwise XOR |
| `\|` | Bitwise OR |
| `in, not in, is, is not, <, <=, >, >=,<>, !=, ==` | Τελεστές σύγκρισης, μέλους, ταυτότητας |
| `not` | Boolean NOT |
| `and` | Boolean AND |
| `or` | Boolean OR |

- Αν σε μια έκφραση υπάρχουν διαδοχικοί τελεστές με *ίδια προτεραιότητα* (που φυσικά μπορεί να είναι ίδιοι ή διαφορετικοί) τότε επαληθεύονται από *αριστερά προς τα δεξιά*
- Εξαίρεση αποτελεί ο τελεστής ύψωσης σε δύναμη *(**)* που όταν υπάρχουν διαδοχικοί τελεστές πχ a**b**c τότε επαληθεύονται από *δεξιά προς τα αριστερά*

# Προτεραιότητα Τελεστών - Παραδείγματα

```
10 + 20 * 30  =  10 + 20 * 30 = 10 + 600 = 610
```

*Επειδή ο πολλαπλασιασμός έχει μεγαλύτερη προτεραιότητα από την πρόσθεση*

*Αν θέλουμε όμως να γίνει πρώτα ή πρόσθεση και μετά ο πολλαπλασιασμός, τότε πρέπει να χρησιμοποιήσουμε παρενθέσεις*

```
(10 + 20) * 30  =  30 * 30 = 900
```

---

```
10 * 8 // 5  = 10 * 8 // 5 = 80 // 5 = 16
```

*Πολλαπλασιασμός και ακέραια διαίρεση έχουν ίδια προτεραιότητα οπότε η επαλήθευση γίνεται από αριστερά προς τα δεξιά* →

Πόσο κάνει το **`10 * (8 // 5)`** ;

---

```
2 ** 3 ** 2  = 2 ** 3 ** 2  = 2 ** 9 = 512
```

*Η ύψωση σε δύναμη αποτελεί εξαίρεση. Εδώ η επαλήθευση γίνεται από δεξιά προς τα αριστερά* ←

*Ενώ:* `(2 ** 3) ** 2  = 8 ** 2 = 64`

Υψηλότερη

| |
|---|
| `()` |
| `**` |
| `~` |
| `*, /, //,%` |
| `+, -` |
| `<<, >>` |
| `&` |
| `^` |
| `\|` |
| `in, not in, is, is not, <, <=, >, >=,<>, !=, ==` |
| `not` |
| `and` |
| `or` |

Χαμηλότερη

# Προτεραιότητα Τελεστών - Παραδείγματα

```
print(3 * 5 + 2)          Αποτέλεσμα: 17
print(2 + 3 * 5)          Αποτέλεσμα: 17
print(3 * (5 + 2))        Αποτέλεσμα: 21
```

Τελεστές με ίδια προτεραιότητα επαληθεύονται *εν γένει* από αριστερά προς τα δεξιά

```
print(5 * 2 // 3)      Αποτέλεσμα: 3
print(5 * (2 // 3))    Αποτέλεσμα: 0
```

Εξαίρεση αποτελεί ο τελεστής ** που επαληθεύεται από δεξιά προς τα αριστερά

```
print(2 ** 3 ** 2)        Αποτέλεσμα: 512
print((2 ** 3) ** 2)      Αποτέλεσμα: 64
```

# Προτεραιότητα Τελεστών - Παραδείγματα

Ασυνήθιστες δυνατότητες χρήσεις των τελεστών εκχώρησης και σύγκρισης στην Python

| | |
|---|---|
| `x = y = z` | αποδεκτό στην Python (ισοδύναμο με `x = y` και `x = z`) |
| `a = b = c = d = 5` | αποδεκτό στην Python (όλα παίρνουν την τιμή 5) |
| `x = y += z` | μη αποδεκτό (αποτελεί συντακτικό λάθος) |

---

`x < y < z` δεν σημαίνει *ούτε* `(x < y) < z` *ούτε* `x < (y < z)`

σημαίνει `(x < y) and (y < z)` και επαληθεύεται από αριστερά προς τα δεξιά

---

Η Python χρησιμοποιεί *short circuiting* (βραχυκύκλωμα;) όταν επαληθεύει τους τελεστές `and` και `or`

`and:` αν το αριστερό μέλος επαληθεύεται `False` τότε δεν προχωράει στην επαλήθευση του δεξιού μέλους αφού το αποτέλεσμα θα είναι ούτως ή άλλως `False`

`or:` αν το αριστερό μέλος επαληθεύεται `True` τότε δεν προχωράει στην επαλήθευση του δεξιού μέλους αφού το αποτέλεσμα θα είναι ούτως ή άλλως `True`

Έτσι πχ η έκφραση : `if (s != None) and (len(s) < 10)` δεν 'χτυπάει' αν το `s` δεν έχει οριστεί

# Υπερφόρτωση Τελεστών

Υπερφόρτωση Τελεστών *(operator overloading)* ονομάζεται η ιδιότητα ορισμένων τελεστών να αποκτούν *διαφορετική λειτουργικότητα* ανάλογα με το *περιβάλλον χρήσης* τους *(δηλαδή ανάλογα με τους τελεστέους πάνω στους οποίους επιδρούν)*

Παράδειγμα 1: ο τελεστής της πρόσθεσης (+)

- Σε "κανονική" χρήση, προσθέτει δυο αριθμούς 3 + 5 ⇨ 8
- Μπορούμε όμως να τον χρησιμοποιήσουμε και για να συνενώσουμε συμβολοσειρές

  'ΚΑΛΗ' + 'ΜΕΡΑ' ⇨ 'ΚΑΛΗΜΕΡΑ'

- Ή ακόμα για να συνενώσουμε δυο λίστες

  [1, 2, 5] + [8, "d", 4 ] ⇨ [1, 2, 5, 8, "d", 4]

Η Python μας δίνει τη δυνατότητα να υπερφορτώνουμε και οι ίδιοι τους τελεστές της ώστε να εξυπηρετούν τις δικές μας ανάγκες

Αυτό γίνεται ορίζοντας κατάλληλα ειδικές κλάσεις για αυτή τη δουλειά

Παράδειγμα 2: ο τελεστής του πολλαπλασιασμού (*)

- Σε "κανονική" χρήση, πολλαπλασιάζει δυο αριθμούς 3 * 5 ⇨ 15
- Μπορούμε όμως να τον χρησιμοποιήσουμε και στις συμβολοσειρές ως εξής

  'χα ' * 5 ⇨ 'χα χα χα χα χα ' '=' * 12 ⇨ '============'

# Είσοδος Δεδομένων - Η συνάρτηση *input*

```
>>> name = input('What is your name? ')
What is your name? John Smith
>>> name
'John Smith'
```

Το *προαιρετικό* κείμενο μέσα στην παρένθεση μιας εντολής `input` ονομάζεται *prompt* και εμφανίζεται στην οθόνη όταν αυτή εκτελείται για καθοδήγηση του χρήστη (πχ. `'Δώσε έναν αριθμό: '`)

Η συνάρτηση input επιστρέφει πάντα *συμβολοσειρά (string)*

```
>>> n = input('Enter a number: ')
Enter a number: 50
>>> print(n + 100)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

Είναι λάθος επειδή προσπαθούμε να κάνουμε την πράξη "50" + 100, δηλαδή να προσθέσουμε έναν *ακέραιο* και μια *συμβολοσειρά*

# Είσοδος Δεδομένων

Για να εισάγουμε *αριθμητικά δεδομένα* πρέπει να *μετατρέψουμε* την συμβολοσειρά που μας επιστρέφει η `input`, σε ακέραιο, δεκαδικό, μιγαδικό … ανάλογα με το τι θέλουμε

```
>>> n = int(input('Enter a number: '))
Enter a number: 50
>>> print(n + 100)
150
```

*Εναλλακτικά:*

```
>>> n = input('Enter a number: ')
Enter a number: 50
>>> n = int(n)
>>> print(n + 100)
150
```

```
>>> c = complex(input('Enter a complex number: '))
Enter a complex number: 3+2j
>>> print(c, c + 2)
(3+2j)  (5+2j)
```

# Είσοδος πολλαπλών δεδομένων στην ίδια γραμμή

```
>>> a,b,c = input('Enter three strings separated with comma: ').split(",")
Enter three strings separated with comma: Hello, Goodbye, 100
>>> print(a,b,c)
'Hello' 'Goodbye' '100'

>>> x,y = input('Enter two numbers: ').split()
Enter two numbers: 4 5
>>> print(x + y)
'45'

>>> x, y = int(x), int(y)
>>> print(x + y)
9
```

Η μέθοδος **split**() διαχωρίζει (σπάει) μια συμβολοσειρά σε επί μέρους τμήματα, ανάλογα με τον διαχωριστικό χαρακτήρα ( στην περίπτωση μας το "," )

Η μέθοδος **split**() χωρίς παραμέτρους σπάει μια συμβολοσειρά στα κενα

Εναλλακτικά: `x, y = [int(k) for k in [x, y]]`

Συμπερίληψη
Comprehension
(θα τη μάθουμε αργότερα)

# Έξοδος Αποτελεσμάτων -Η συνάρτηση *print()*

**print(<obj$_1$>, <obj$_2$>, … <obj$_k$>)**

Μετατρέπει όλα τα αντικείμενα *(objects)* σε *συμβολοσειρές* και τα εκτυπώνει στην οθόνη, χωρισμένα με *κενά*[1], προσθέτοντας *αυτόματα* στο τέλος τον ειδικό χαρακτήρα *(\n) NEWLINE* ώστε να προχωρήσει στην επόμενη γραμμή[2]

```
>>> First = 'John'
>>> Last = 'Smith'
>>> Age = 23
>>> print(First, Last, ' is ', Age, ' years old')
John Smith is 23 years old
```

[1] *μπορούμε αν θέλουμε να χρησιμοποιήσουμε και άλλο χαρακτήρα, θα δούμε σε λίγο πως γίνεται*

[2] *μπορούμε αν θέλουμε να παραμείνουμε στην ίδια γραμμή, θα δούμε σε λίγο πως γίνεται*

# Έξοδος Αποτελεσμάτων -Η συνάρτηση *print()*

Η συνάρτηση **`print`** εκτυπώνει αντικείμενα *οποιουδήποτε τύπου*, ακόμα και κάποια πολύ… ασυνήθιστα.

```
>>> a = [1, 2, 3]
>>> type(a)
<class 'list'>

>>> b = -12
>>> type(b)
<class 'int'>

>>> type(len)
<class 'builtin_function_or_method'>

>>> type(print)
<class 'builtin_function_or_method'>
```

---

```
>>> print(a, b, None, len, print)
[1, 2, 3] -12 None <built-in function len> <built-in function print>
```

# Οι παράμετροι `sep` και `end` στη συνάρτηση *print()*

Είναι *ορίσματα-κλειδιά* (*keyword arguments*) *θα μάθουμε αργότερα τι είναι αυτά* τα οποία τοποθετούνται προαιρετικά στο τέλος της λίστας των ορισμάτων της **`print`** και καθορίζουν την συμπεριφορά της

Η παράμετρος **`sep`** καθορίζει τον/τους *διαχωριστικούς χαρακτήρες* που θέλουμε να εμφανίζονται *ανάμεσα* στα διαδοχικά ορίσματα που εκτυπώνονται. Η *προκαθορισμένη τιμή* της παραμέτρου **`sep`** είναι το *κενό* `' '` όμως αυτό μπορεί να το αλλάξει κάποιος κατά βούληση

```
>>> print('foo', 42, 'bar')  Δεν ορίζεται το sep - τα αποτελέσματα διαχωρίζονται με κενό
foo 42 bar
>>> print('foo', 42, 'bar', sep='/') Τα αποτελέσματα διαχωρίζονται με /
foo/42/bar
>>> print('foo', 42, 'bar', sep='...') Τα αποτελέσματα διαχωρίζονται με ...
foo...42...bar
```

*sep = separator (διαχωριστικό)*

# Οι παράμετροι **sep** και **end** στη συνάρτηση *print()*

Η παράμετρος **end** καθορίζει έναν ή περισσότερους χαρακτήρες που θέλουμε να εκτυπωθούν *στο τέλος της γραμμής*, αφού έχουν πια εκτυπωθεί όλες οι τιμές

Η *προκαθορισμένη τιμή* της παραμέτρου **end** είναι το *newline* `'/n'` ώστε να *κατεβαίνει αυτόματα στην επόμενη γραμμή*, όμως αυτό μπορεί να αλλάξει δίνοντας διαφορετική τιμή στην παράμετρο **end**

```
print('Πρωί') Δεν ορίζεται το end, επομένως θεωρείται ως '/n' – Κατεβαίνει στην επόμενη γραμμή
print('Βράδι')
Πρωί
Βράδι
```

```
print('Πρωί', end ='') Το end ορίζεται ως κενή συμβολοσειρά - παραμένει στην ίδια γραμμή
print('Βράδι')
ΠρωίΒράδι
```

```
print( Πρωί', end ='_Μεσημέρι_') Τυπώνει την συμβολοσειρά '_Μεσημέρι_', και παραμένει στην ίδια γραμμή
print('Βράδι')
Πρωί_Μεσημέρι_Βράδι
```

# Μορφοποίηση συμβολοσειρών για εκτύπωση 1/2

*(Θα δούμε τις συμβολοσειρές πιο αναλυτικά στο επόμενο μάθημα)*

## 1ος Τρόπος: Ο τελεστής %

```
>>> Name = "John"
>>> age=23
>>> print("%s is %d years old" % (Name, age))
John is 23 years old
```

---

```
>>> a=20
>>> b=3
>>> c=a/b
>>> print("%d / %d is %.4f" % (a,b,c))
20 / 3 is 6.6667
```

---

```
>>> k=127
>>> print ("%d in hex is %X" % (k,k))
127 in hex is 7F
```

`%s` - String (ή ο,τιδήποτε μπορεί να αναπαρασταθεί σαν string – πχ αριθμοί)

`%d` - Ακέραιοι

`%f` - Δεκαδικοί

`%.<αρ. ψηφίων>f` - Δεκαδικοί με συγκεκριμένο πλήθος ψηφίων

`%x/%X` – Ακέραιοι σε δεκαεξαδική μορφή (πεζά abcdef /κεφαλαία ABCDEF)

# Μορφοποίηση συμβολοσειρών για εκτύπωση 2/2

*(Θα δούμε τις συμβολοσειρές πιο αναλυτικά στο επόμενο μάθημα)*

## 2ος Τρόπος : χρήση της μεθόδου format

```
>>> Name = "John"
>>> age=23
>>> print("{} is {} years old".format(Name,age))
John is 23 years old

>>> print("{1} is {0} years old".format(age, Name))
John is 23 years old

>>> print("{Name} is {age} years old".format(Name="John",age=23))
John is 23 years old
```

## 3ος Τρόπος : χρήση fstrings (formatted string literals)

Προσέξτε το `f` μπροστά από την συμβολοσειρά!

```
>>> print(f"{Name} is {age} years old")
John is 23 years old
```

# Δομές επιλογής και επανάληψης της Python

# Η έννοια του μπλοκ εντολών στην Python

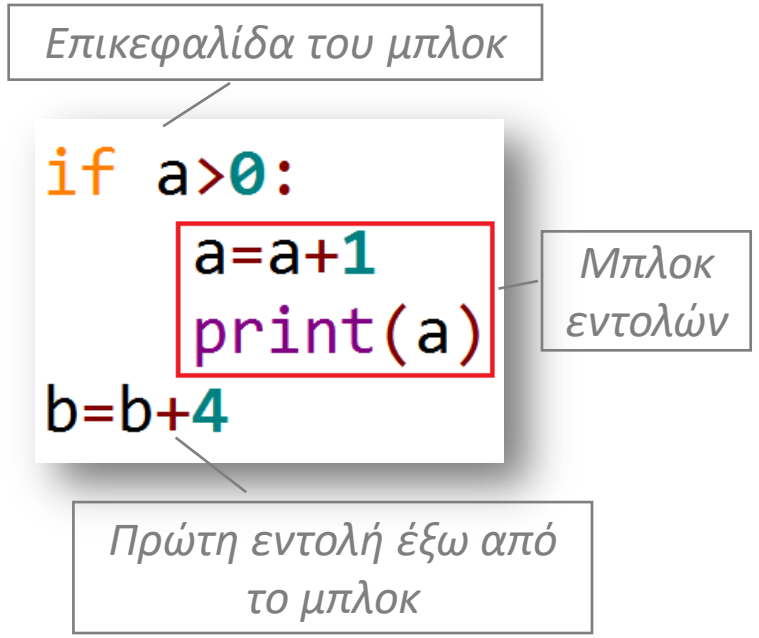
Ενα *μπλοκ εντολών* στην Python είναι μια *ακολουθία εντολών* που αποτελεί μια *αυτόνομη ενιαία οντότητα.*

Παραδείγματα:

— ο *κώδικας* μιας συνάρτησης

— οι *εντολές* μέσα σε μια επανάληψη (for, while κλπ.)

— οι *εντολές* που ακολουθούν ένα if ή ένα else




- Ένα μπλοκ εντολών ορίζεται από εσοχές (indentations)
- Συνήθως η εσοχή γίνεται με *4 κενά* ή με το *πλήκτρο tab*, *αλλά γενικά προτείνεται η χρήση 4 κενών για συνέπεια*
- Οι εντολές που ανήκουν στο ίδιο μπλοκ πρέπει να έχουν το ίδιο επίπεδο εσοχής

Αντίθετα με άλλες γλώσσες προγραμματισμού που χρησιμοποιούν σύμβολα όπως {} ή begin/end για τον καθορισμό των μπλοκ, *η Python βασίζεται αποκλειστικά στις εσοχές*

# Πολλαπλά μπλοκ εντολών (nested blocks)

- *Πολλαπλά μπλοκ εντολών* μπορούν να περιέχονται το ένα μέσα στο άλλο (nested blocks) και διακρίνονται με *διαφορετικά επίπεδα εσοχής*

- Η *εσοχή* σε κάθε νέο μπλοκ είναι συνήθως *4 κενά σε σχέση με το προηγούμενο μπλοκ*

- Η εντολή που *προηγείται* κάθε μπλοκ *(επικεφαλίδα του μπλοκ) τελειώνει πάντα* με τον χαρακτήρα άνω-κάτω τελεία (:)

```
if a > 0:
    a = a + 1
    if a > b:
        b = b + 1
        print(b)
    print(a)
    if a > c:
        c = a * b
        print(c)
print(d)
```

- Κάθε εντολή που πληκτρολογείται *διαδραστικά (πχ στο IDLE) νοείται από μόνη της ως ένα στοιχειώδες μπλοκ*

*Η Python βασίζεται αποκλειστικά στις εσοχές:* Αν δεν υπάρχει εσοχή ή αν δεν ταιριάζει η εσοχή μεταξύ των γραμμών, η Python θα επιστρέψει σφάλμα σύνταξης (IndentationError)

# Δομές Επιλογής

## Απλό if

Δεν ξεχνάμε την υποχρεωτική εσοχή

```
a = 33
b = 200
if b > a:
    print("b is greater than a")
```

Δεν ξεχνάμε την άνω-κάτω τελεία

*Εναλλακτικός* τρόπος γραφής σε μια μόνο γραμμή
(Ισχύει γενικά αν ένα μπλοκ περιέχει *μια μόνο* εντολή)

```
if b > a: print("b is greater than a")
```

***Η εσοχή πριν από το print είναι απαραίτητη***

Λείπει η εσοχή

```
if b > a:
print("b is greater than a")
```

```
SyntaxError: expected an indented block
```

## if - else

```
if b > a:
  print("Max = ", b)
else:
  print("Max = ", a)
```

## Πολλαπλό if - else

```
if b > a:
  print("Max = ", b)
elif b == a:   (elif που σημαίνει else if)
  print("They are equal")
else:
  print("Max = ", a)
```

Δεν ξεχνάμε την υποχρεωτική εσοχή στο κάθε μπλοκ

# Δομές Επιλογής – Παράδειγμα 1

```
num = int(input("Δώσε έναν θετικό ακέραιο: "))
```
*( Θυμόμαστε γιατί χρειάζεται το **int**; )*

```
if 9 < num < 100:
    print(num, "είναι διψήφιος")
elif 99 < num < 1000:
    print(num, " είναι τριψήφιος ")
elif 999 < num < 10000:
     print(num, " είναι τετραψήφιος ")
else:
    print("ο αριθμός που έδωσες είναι είτε < 10 είτε > 10000")
```

⇦ Εναλλακτικός(*) τρόπος γραφής του `(num > 9) and (num < 100)`

**Παράδειγμα εκτέλεσης προγράμματος:**

```
Δώσε έναν θετικό ακέραιο: 1122
1122 είναι τετραψήφιος
```

(*) Python - chaining comparison operators

# Δομές Επιλογής - Παράδειγμα 2

```
var1 = ['apples', 'oranges', 'cherries']
if (type(var1) == int):
    print("Type of the variable is Integer")
elif (type(var1) == float):
    print("Type of the variable is Float")
elif (type(var1) == complex):
    print("Type of the variable is Complex")
elif (type(var1) == bool):
    print("Type of the variable is Bool")
elif (type(var1) == str):
    print("Type of the variable is String")
elif (type(var1) == tuple):
    print("Type of the variable is Tuple")
elif (type(var1) == dict):
    print("Type of the variable is Dictionary")
elif (type(var1) == list):
    print("Type of the variable is List")

else:
    print("Type of the variable is Unknown")
```

**Αποτέλεσμα:**

```
Type of the variable is List
```

# Δομές Επιλογής - Παράδειγμα 3 (simple calculator)

```
a, sign, b = input('Γράψε μια απλή αριθμητική πράξη: ').split()
a, b = float(a), float(b)
if sign == "+":
  print(a, sign, b, ' = ', a+b)
elif sign == "-":
  print(a, sign, b, ' = ', a-b)
elif sign == "*":
  print(a, sign, b, ' = ', a*b)
elif sign == "/":
  print(a, sign, b, ' = ', a/b)
else:
  print('Άγνωστο σύμβολο: ', sign)
```

ισοδύναμο με `.split(" ")`

Αν δεν ορίσουμε κάποιο διαχωριστικό χαρακτήρα στη συνάρτηση split(), εννοείται το κενό

**Αποτέλεσμα εκτέλεσης προγράμματος:**

```
Γράψε μια απλή αριθμητική πράξη: 8 / 3
8.0 / 3.0  =  2.6666666666666665
>>>
```

```
Γράψε μια απλή αριθμητική πράξη: 8 & 3
Άγνωστο σύμβολο: &
>>>
```

# Εμφωλευμένες (nested) Δομές Επιλογής

Δομές επιλογής που έχουν μέσα τους άλλες δομές επιλογής

Παράδειγμα: Εύρεση *μέγιστου* τριών αριθμών

```
x, y, z = input('Δώσε τρεις αριθμούς: ').split()
x, y, z = float(x), float(y), float(z)
print ("Ο μεγαλύτερος είναι ", end="")
if x > y:
    if x > z:
        print(x)
    elif y > z:
        print(y)
else:
    if y > z:
        print(y)
    else:
        print(z)
```

ισοδύναμο με `.split(" ")`

Αν δεν ορίσουμε κάποιο διαχωριστικό χαρακτήρα στη συνάρτηση split(), εννοείται το κενό

```
Αποτέλεσμα εκτέλεσης προγράμματος:
  Δώσε τρεις αριθμούς: 1 5 2
  Ο μεγαλύτερος είναι 5.0
  >>>
```

# Δομές επανάληψης

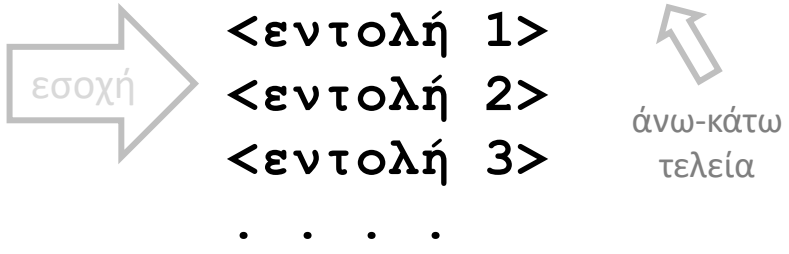
Η Python έχει δυο δομές επανάληψης: **`while`** και **`for`**

- Το **`while`** χρησιμοποιείται κυρίως για *επαναλήψεις* το *πλήθος* των οποίων *δεν είναι γνωστό από πριν*. Σε κάθε επανάληψη ελέγχεται η *συνθήκη τερματισμού* και αποφασίζεται αν θα συνεχίσει ή όχι

- Το **`for`** χρησιμοποιείται κυρίως για να διατρέχει
  - *είτε πεπερασμένες αριθμητικές ακολουθίες* (δημιουργούνται με την εντολή **`range`**)
  - *είτε ακολουθίες αντικειμένων* σε λίστες, σύνολα, λεξικά, συμβολοσειρές κλπ.

- Η Python υποστηρίζει *εμφωλευμένες δομές επανάληψης (nested loops)* δηλαδή επαναλήψεις που περιέχουν μέσα τους άλλες επαναλήψεις

# Η δομή επανάληψης while




```
while <συνθήκη> :
    <εντολή 1>
    <εντολή 2>
    <εντολή 3>
    . . . .
```

Πχ.

```
i=0
while i<10:
    i=i+1
    print(i)
```

- Η επανάληψη *συνεχίζεται* όσο εξακολουθεί να *ισχύει* η συνθήκη
- Προσθέτοντας ένα `else` στο τέλος του `while` ορίζουμε ένα μπλοκ εντολών που εκτελούνται αμέσως μετά την έξοδο από την συνθήκη, εφόσον η έξοδος αυτή γίνει *φυσιολογικά*, δηλαδή χωρίς την χρήση της εντολής `break`

*(θα τη δούμε παρακάτω)*

```
while i<10:
    i=i+1
    print(i)
else:
    print("Η επανάληψη τερμάτισε φυσιολογικά – χωρίς break")
```

# break και continue μέσα στη δομή επανάληψης while

- Για να *διακόψουμε* τον κύκλο των επαναλήψεων νωρίτερα (παρόλο που η συνθήκη ελέγχου εξακολουθεί να ισχύει) χρησιμοποιούμε την εντολή **break**. Στην περίπτωση αυτή δεν εκτελείται το μπλοκ του **else** *(αν υπάρχει)*
- Για να *παραλείψουμε* τις εντολές του μπλοκ που απομένουν στην *παρούσα επανάληψη* και να προχωρήσουμε στην *αμέσως επόμενη*, χρησιμοποιούμε την εντολή **continue**

*Παραδείγματα*

```
a=4
while a > 0:
    a -= 1
    print(a)
    if a==2: break
```

```
3
2
```

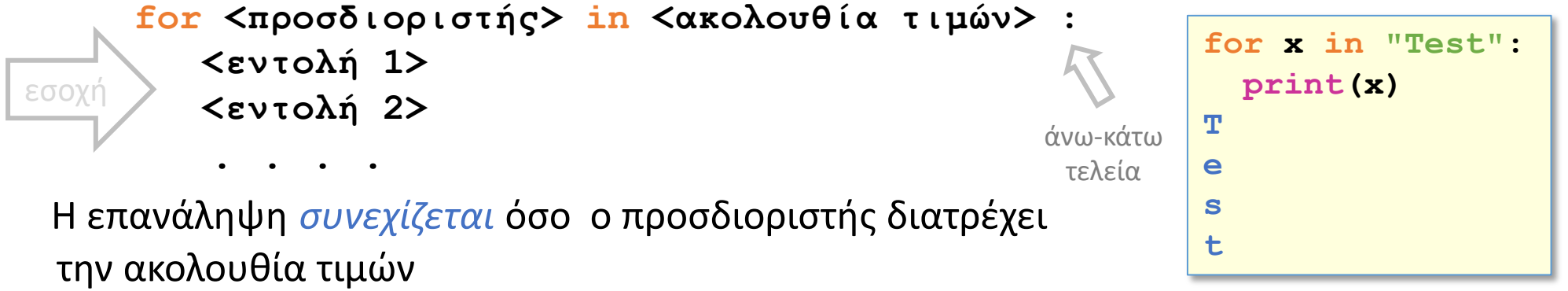
```
a=4
while (a > 0):
    a -= 1
    print(a)
    if a==1: break
else:
    print("Φτάσαμε στο 0")
```

```
3
2
1
```

```
a=4
while a > 0:
    a -= 1
    if a==2: continue
    print(a)
else:
    print("Φτάσαμε στο 0")
```

```
3
1
0
Φτάσαμε στο 0
```

# Η δομή επανάληψης for




- Η επανάληψη *συνεχίζεται* όσο ο προσδιοριστής διατρέχει την ακολουθία τιμών
- Προσθέτοντας ένα **else** στο τέλος του **for** ορίζουμε μια ή περισσότερες εντολές που θα εκτελεστούν αμέσως μετά την εξάντληση της ακολουθίας τιμών

```
fruits = ["apple","banana","cherry"]
for x in fruits:
   print(x)
```

```
apple
banana
cherry
```

```
fruits = ["apple","banana","cherry"]
for x in fruits:
   print(x)
else:
  print("no more fruits")
```

```
apple
banana
cherry
no more fruits
```

# break και continue μέσα στη δομή επανάληψης for

Ισχύουν ακριβώς τα ίδια που περιγράψαμε νωρίτερα για το while

- Για να *διακόψουμε* νωρίτερα το `for` *(πριν ο προσδιοριστής του εξαντλήσει την ακολουθία τιμών)* χρησιμοποιούμε την εντολή `break`. Στην περίπτωση αυτή δεν εκτελείται το μπλοκ του `else` *(αν υπάρχει)*
- Για να *παραλείψουμε* τις εντολές που απομένουν στην *παρούσα επανάληψη* και να προχωρήσουμε στην *αμέσως επόμενη*, χρησιμοποιούμε την εντολή `continue`

```
fruits = ["apple","banana","cherry"]
for x in fruits:
   if x == "banana": break
   print(x)
else:
  print("no more fruits")
```

```
apple
```

```
fruits = ["apple","banana","cherry"]
for x in fruits:
   if x == "banana": continue
   print(x)
else:
  print("no more fruits")
```

```
apple
cherry
no more fruits
```

# Εμφωλευμένες δομές επανάληψης

## Nested loops

Προκύπτουν *συνδυαστικά* από επαναλήψεις που βρίσκονται *μέσα σε άλλες επαναλήψεις*

Δεν υπάρχει πρακτικό όριο για το *επίπεδο βάθους*

*(αν και συνήθως αυτό δεν ξεπερνά τα 2 ή 3 επίπεδα)*

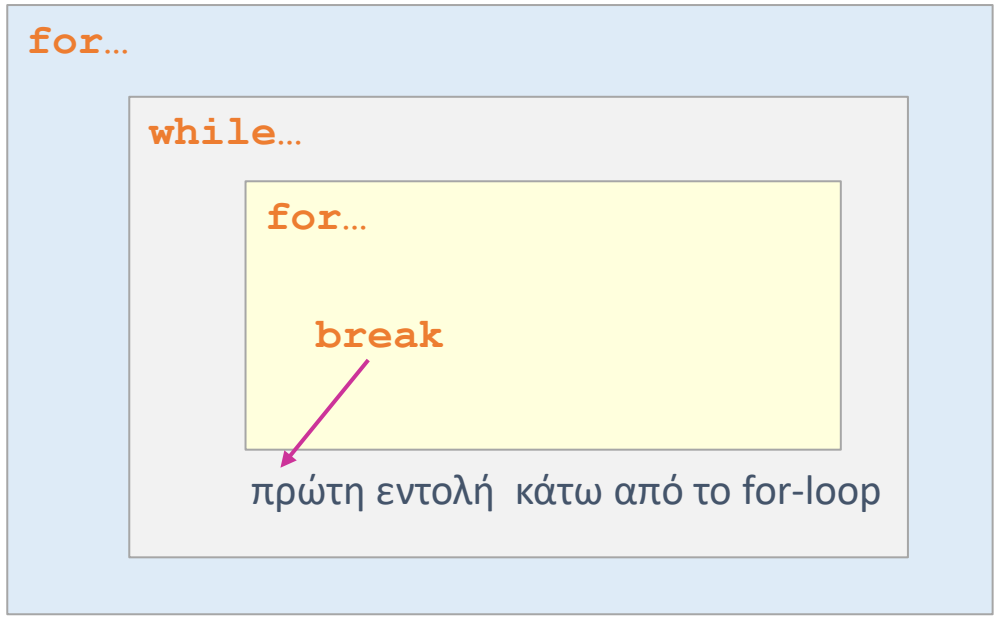
*Παραδείγματα*:


Η εντολή break μιας *εσωτερικής επανάληψης* ισχύει μόνο ***τοπικά*** δηλαδή μόνο στο δικό της επίπεδο βάθους - οι *εξωτερικές* από αυτήν επαναληπτικές δομές *δεν επηρεάζονται*

# Εμφωλευμένες δομές επανάληψης

## Συμπεριφορά της εντολής break

Η εντολή break μιας *εσωτερικής επανάληψης* ισχύει μόνο ***τοπικά*** δηλαδή μόνο στο δικό της επίπεδο βάθους - οι *εξωτερικές* από αυτήν επαναληπτικές δομές *δεν επηρεάζονται*




Εδώ το break διακόπτει *(σπάει)* το *εσωτερικό* for-loop και η εκτέλεση συνεχίζεται με την πρώτη εντολή του while κάτω από αυτό

# Παράδειγμα τριπλής εμφωλευμένης επανάληψης με for

```
epitheta = ["νόστιμα", "ζουμερά", "φρέσκα"]
megethos = ["μεγάλα ", "μικρά"]
frouta = ["μήλα", "αχλάδια", "πεπόνια"]

for x in  epitheta:
    for y in  megethos:
        for z in frouta:
            print(x, y, z)
```

```
νόστιμα μεγάλα μήλα
νόστιμα μεγάλα αχλάδια
νόστιμα μεγάλα πεπόνια
νόστιμα μικρά μήλα
νόστιμα μικρά αχλάδια
νόστιμα μικρά πεπόνια
ζουμερά μεγάλα μήλα
ζουμερά μεγάλα αχλάδια
ζουμερά μεγάλα πεπόνια
ζουμερά μικρά μήλα
ζουμερά μικρά αχλάδια
ζουμερά μικρά πεπόνια
φρέσκα μεγάλα μήλα
φρέσκα μεγάλα αχλάδια
φρέσκα μεγάλα πεπόνια
φρέσκα μικρά μήλα
φρέσκα μικρά αχλάδια
φρέσκα μικρά πεπόνια
```

# Η συνάρτηση `range()`

Η συνάρτηση **`range()`** είναι μια *γεννητορική συνάρτηση* της Python *(περισσότερα σε επόμενες διαλέξεις)* η οποία χρησιμοποιείται για την παραγωγή ακολουθιών ακεραίων και ορίζεται ως εξής:

```
range([start], stop, [step])
```

```
range(stop)
range(start, stop)
range(start, stop, step)
```

- Τα ***start***, ***stop*** και ***step*** είναι ακέραιες εκφράσεις
- Τα ***start*** και ***step*** είναι προαιρετικά
- Παράγει διαδοχικούς ακέραιους μέχρι το ***stop*** *χωρίς αυτό να συμπεριλαμβάνεται (δηλ. μέχρι το stop–1)*
- Αν δεν υπάρχει το προαιρετικό ***start*** η συνάρτηση ξεκινάει *από το 0*
- Αν δεν υπάρχει το προαιρετικό ***step***, τότε ως βήμα παραγωγής θεωρείται το 1

| | |
|---|---|
| `range(4)` | Παράγει την ακολουθία αριθμών `0,1,2,3` *ισοδύναμο με* `range(0,4)` |
| `range(1,7)` | Παράγει την ακολουθία αριθμών `1,2,3,4,5,6` |
| `range(1,7,2)` | Παράγει την ακολουθία αριθμών `1,3,5` |
| `range(8,5,-1)` | Παράγει την ακολουθία αριθμών `8,7,6` |
| `range(8,5)` | Δεν παράγει τίποτε *(γιατί;)* |

# Η συνάρτηση range() στα for loops

*Η συνάρτηση range() χρησιμοποιείται ευρύτατα στα for loops για να δημιουργεί ακολουθίες τιμών*

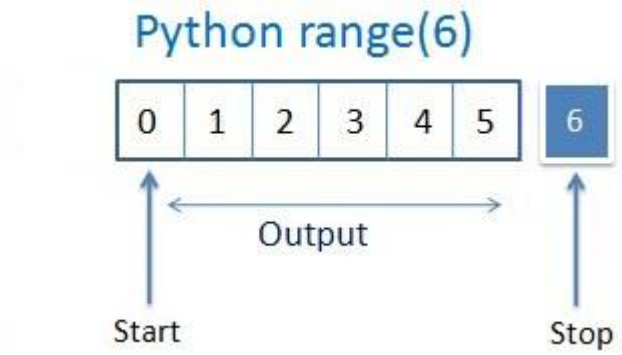
`range(j)` : Επιστρέφει διαδοχικούς ακεραίους από το `0` μέχρι και το `j-1`

`range(i, j)` : Επιστρέφει διαδοχικούς ακεραίους από το `i` μέχρι και το `j-1`

`range(i,j,k)` : Επιστρέφει διαδοχικούς ακεραίους από το `0` μέχρι και το `j-1` ανά βήμα `k`

```
print("Python example to print numbers from 0 to 5")
for i in range(6):
    print(i, sep=', ')  (Θυμόμαστε τι ακριβώς κάνει το sep;)
```

```
Python example to print numbers from 0 to 5
0, 1, 2, 3, 4, 5,
```




```
for i in range(1,6,2): print(i, sep=', ')
1,3,5,
for i in range(10,2,-1): print(i, sep=', ')
10, 9, 8, 7, 6, 5, 4, 3,
```

# Παραδείγματα

```
print("Τα διψήφια πολλαπλάσια του 12 είναι:")
for x in range(1,100):
    if (x%12) == 0: print(x, sep=" ")
```

```
Τα διψήφια πολλαπλάσια του 12 είναι:
12 24 36 48 60 72 84 96
```

---

```
k = int(input("Δώσε ένα διψήφιο ακέραιο: "))
print("Τα διψήφια πολλαπλάσια του", k,  "είναι:")
for x in range(k,100,k):
    print(x, sep=" ")
```

```
Δώσε ένα διψήφιο ακέραιο: 11
Τα διψήφια πολλαπλάσια του 11 είναι:
11 22 33 44 55 66 77 88 99
```

Ποιο από τα δυο αυτά παραδείγματα παράγει πιο *αποτελεσματικό* (εν προκειμένω *ταχύτερο*) κώδικα, και *γιατί;*

# Τέλος Διάλεξης

Ερωτήσεις;

*Τμήματα αυτής της διάλεξης περιέχουν πληροφορίες από πηγές που είναι ελεύθερες στο διαδίκτυο όπως η Βικιπαίδεια και ανοιχτές σημειώσεις παρεμφερών διαλέξεων.*