# Εισαγωγή στον Προγ/μό Υπολογιστών

Διάλεξη 11

*Εξαιρέσεις* (*exceptions*)

# Περιεχόμενα

# Σφάλματα

- Τύποι σφαλμάτων:
  1. *Συντακτικά* λάθη

```
>>> def function(,a)
  File "<stdin>", line 1
    def function(,a)
                 ^
SyntaxError: invalid syntax
```

  2. Σφάλματα *εκτέλεσης*
     - Σε γλώσσες με *διερμήνευση* (πχ, Python) τα *συντακτικά* είναι σφάλματα *εκτέλεσης*

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

  3. Σφάλματα λογικής

```
>>> i = 0
>>> while i < 10:
        print(i)
```

Η *ροή εκτέλεσης* δεν μπορεί να συνεχιστεί χωρίς χειρισμό του σφάλματος

*Εξαιρέσεις*

# *Εξαιρέσεις*

- ***Εξαίρεση (exception):*** η διακοπή της κανονικής ροής εκτέλεσης ενός προγράμματος

- ***Χειρισμός*** *εξαίρεσης (exception* ***handling****)*: *εκτέλεση* ειδικού *κώδικα* που χειρίζεται την εξαίρεση (πχ, εμφάνιση μηνύματος λάθους και συνέχιση της εκτέλεσης)

- Σε περίπτωση που δεν υπάρχει χειρισμός οι *εξαιρέσεις* προκαλούν τερματισμό *εκτέλεσης* προγράμματος
  - Έξω από το *διαδραστικό περιβάλλον*, ο *διερμηνευτής* τερματίζει
  - Στο *διαδραστικό περι*βάλλον, σταματάει η αποτίμηση και εμφανίζεται η προτροπή >>>
  - Εμφάνιση του σημείου εντοπισμού (σηματοδότησης) και του τύπου της *εξαίρεσης*

# *Εξαιρέσεις*

- Το σημείο όπου συνέβη η εξαίρεση περιγράφεται με το ***stack trace***
  - *stack trace*: αλληλουχία κλήσεων όπου προκλήθηκε η εξαίρεση

```
>>> def f():
        g()
>>> def g():
        h()
>>> def h():
        1/0
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
  File "<stdin>", line 2, in g
  File "<stdin>", line 2, in h
ZeroDivisionError: division by zero
```

# *Εξαιρέσεις*

- Οι *εξαιρέσεις* μπορούν να προκληθούν κατά βούληση με την εντολή `raise`

```
>>> raise ZeroDivisionError()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError
```

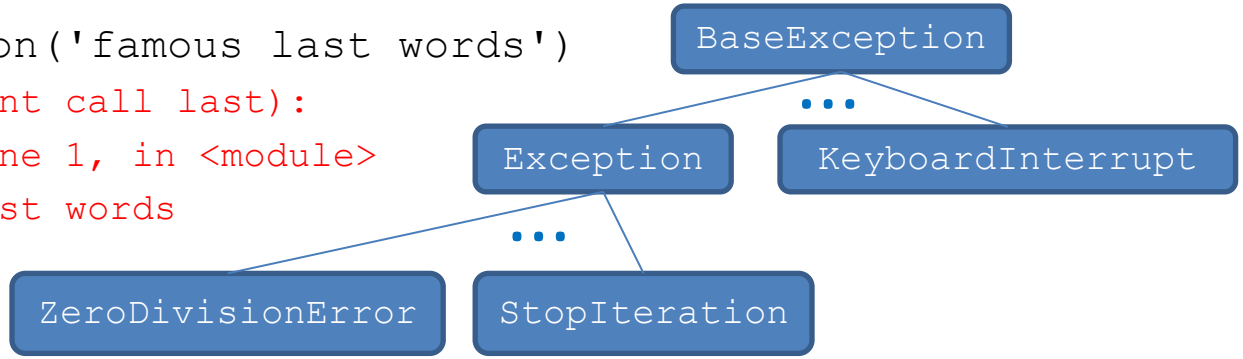
- Κάθε *εξαίρεση* δημιουργεί ένα *αντικείμενο* που περιγράφει το είδος της και άλλες πληροφορίες. Το είδος της *εξαίρεσης* αντιστοιχεί στην *τάξη* του *αντικειμένου*

- Γενική μορφή: `raise <έκφραση>`
  - `<έκφραση>`: *αντικείμενο τάξης* που κληρονομεί την ενσωματωμένη *τάξη* `BaseException`

# *Εξαιρέσεις*

- Συνήθως χρησιμοποιούνται *αντικείμενα τάξης* που κληρονομεί την `Exception`

```
>>> raise Exception('famous last words')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: famous last words
```




- Οι περισσότερες ενσωματωμένες *εξαιρέσεις* κληρονομούν την `Exception`
  - Πχ, `ZeroDivisionError`, `StopIteration`, `ValueError`, `SyntaxError`

- Ο προγραμματιστής μπορεί να ορίσει και να χρησιμοποιήσει σε `raise` δικές του *εξαιρέσεις* που κληρονομούν την `Exception`

# Χειρισμός *εξαιρέσεων*

# Χειρισμός *εξαιρέσεων*

- Χειρισμός σφαλμάτων με την εντολή `try`

```
>>> try:
        print('Hello')
        1/0
        print('Never printed')
    except ZeroDivisionError as e:
        print(e, 'caught')

Hello
division by zero caught
```

- Ο χειρισμός μιας *εξαίρεσης* που σηματοδοτείται μέσα στο *μπλοκ εντολών* της σύνθετης εντολής `try`, γίνεται από το *μπλοκ εντολών* της *επικεφαλίδας* `except` που αφορά τον τύπο αυτής της *εξαίρεσης*

# Χειρισμός *εξαιρέσεων*

- Γενική μορφή:

```
try:
    <μπλόκ_εντολών_try>
except <τάξη_εξαίρεσης> as <όνομα>:
    <μπλόκ_εντολών>
...
```

  - Πρέπει να υπάρχει τουλάχιστον μια επικεφαλίδα `except`

*Εκτέλεση*:

1. *Εκτελούνται* οι εντολές στο `<μπλόκ_εντολών_try>`
2. Αν δεν υπάρξει *εξαίρεση*, μετά το `<μπλόκ_εντολών_try>` τελειώνει η εκτέλεση της `try`. (Δεν *εκτελούνται* τα `<μπλόκ_εντολών>` των *επικεφαλίδων* `except`)
3. Αν προκληθεί *εξαίρεση* στις εντολές του `<μπλόκ_εντολών_try>` ή στις συναρτήσεις/*μεθόδους* που καλούν, σταματάει η *ροή εκτέλεσης* και ο έλεγχος μεταφέρεται στο `<μπλόκ_εντολών>` της πρώτης *επικεφαλίδας* `except` με `<τάξη_εξαίρεσης>` ίδια με (ή πιο γενική από) το *αντικείμενο εξαίρεσης*

# Χειρισμός *εξαιρέσεων*

```
>>> for x in [1, 2, 'hello']:
        print(x)
1
2
hello
```

- Ισοδύναμη υλοποίηση `for` με `while`:

```
>>> iterator = iter([1, 2, 'hello'])
>>> try:
        while True:
            x = next(iterator)
            print(x)
    except StopIteration:
        pass
1
2
hello
```

# Χειρισμός *εξαιρέσεων*

- Παράδειγμα:

```
>>> def inverse(x):
        return 1/x
>>> inverse(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in inverse
ZeroDivisionError: division by zero

>>> def safe_inverse(x):
        try:
            return 1/x
        except ZeroDivisionError():
            print("Can't invert 0.")
>>> safe_inverse(0)
Can't invert 0.
```

# Χειρισμός *εξαιρέσεων*

- Ο χειρισμός μιας *εξαίρεσης* ίσως είναι πιο κατάλληλο να γίνει αλλού: η `safe_inverse` δεν γνωρίζει τι πρέπει να κάνει

- Πιο χρήσιμο παράδειγμα: χειρισμός απομακρυσμένων σφαλμάτων

```
>>> try:
        print('Hello')
        x = int(input('Integer to invert: '))
        print(inverse(x))
    except Exception:
        print('Hey user, don\'t waste my time')

Hello
Integer to invert: 0
Hey user, don't waste my time
```

# Χειρισμός *εξαιρέσεων*

- Οι *εξαιρέσεις* είναι ένας τρόπος χειρισμού σφαλμάτων εκτέλεσης

- Ένας άλλος τρόπος είναι ο έλεγχος εγκυρότητας (πχ, με `if`) και ειδοποίηση σφαλμάτων με ειδικές τιμές επιστροφής των συναρτήσεων
  - Παράδειγμα: `example_without_exceptions.py`
  - Ο χειρισμός σφαλμάτων (έστω για ειδοποίηση εντοπισμού) είναι διάσπαρτος σε όλο το πρόγραμμα

- Με τις *εξαιρέσεις*, ο χειρισμός σφαλμάτων και ο εντοπισμός τους οργανώνονται σε χωριστά τμήματα του προγράμματος
  - Παράδειγμα: `example_with_exceptions.py`
  - Πιο ευανάγνωστα προγράμματα

# Κώδικας χωρίς εντοπισμό/χειρισμό λαθών

```
from operator import add, sub, mul, truediv
def my_parse(s):
    cmd_list = s.split()

    operators = {'+':add, '-':sub, '*':mul, '/':truediv}
    op = operators[cmd_list[1]]
    x = float(cmd_list[0])
    y = float(cmd_list[2])

    return op, x, y



while True:
    cmd = input('My >>> ')
    func, x, y = my_parse(cmd)
    print(func(x, y))
```

Λάθος `KeyError` αν ο τελεστής `cmd_list[1]` δεν είναι `'+'`, `'-'`, `'*'`, ή `'/'`

Λάθος `IndexError` αν η συμβολοσειρά `s` περιέχει λιγότερες από τρεις λέξεις (αριστερός τελεστέος, τελεστής, δεξιός τελεστέος)

Λάθος `ValueError` αν ο τελεστέος `cmd_list[0]` ή `cmd_list[2]` δεν είναι αριθμός

Λάθος `ZeroDivisionError` εάν κληθεί η διαίρεση (`func == truediv`) και `y == 0`

# Εντοπισμός & χειρισμός λαθών χωρίς εξαιρέσεις

```
from operator import add, sub, mul, truediv
def my_parse(s):
    cmd_list = s.split()
    if len(cmd_list) != 3:
        print('Wrong expression')
        return None

    operators = {'+':add, '-':sub, '*':mul, '/':truediv}
    if cmd_list[1] not in operators:
        print('Unknown operator')
        return None
    op = operators[cmd_list[1]]
    if not cmd_list[0].isnumeric() or not cmd_list[2].isnumeric():
        print('Illegal argument')
        return None
    x = float(cmd_list[0])
    y = float(cmd_list[2])
    if op == truediv and y == 0:
        print('Can\'t divide by 0’)
        return None

    return op, x, y


while True:
    cmd = input('My >>> ')
    result = my_parse(cmd)
    if result != None:
        func, x, y = result
        print(func(x, y))
```

# Εντοπισμός & χειρισμός λαθών με εξαιρέσεις

```
from operator import add, sub, mul, truediv
def my_parse(s):
    cmd_list = s.split()
    if len(cmd_list) > 3:
        raise IndexError

    operators = {'+':add, '-':sub, '*':mul, '/':truediv}
    op = operators[cmd_list[1]]
    x = float(cmd_list[0])
    y = float(cmd_list[2])

    return op, x, y


while True:
    try:
        cmd = input('My >>> ‘)
        func, x, y = my_parse(cmd)
        print(func(x, y))
    except IndexError:
        print('Wrong expression')
    except KeyError:
        print('Illegal operator')
    except ValueError:
        print('Illegal argument')
    except ZeroDivisionError:
        print('Can\'t divide by 0')
```