Προγραμματιστικά Εργαλεία και Τεχνολογίες για Επιστήμη Δεδομένων

Παράδοση 24/9/2020, Νίκος Παπασπύρου.

Συναρτήσεις και μετατροπή τύπων παραμέτρων

1. Οι παράμετροι έχουν δυναμικούς τύπους. Αν μια συνάρτηση θέλει να ελέγξει τους τύπους των παραμέτρων της, μπορεί να το κάνει π.χ. με τη συνάρτηση `type`. Οι συναρτήσεις `int` και `str` μετατρέπουν σε ακέραιο και string αντίστοιχα.

```
def fib(n):
    convert = False
    if type(n) == str:
        n = int(n)
        convert = True
    a, b = 0, 1
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    if convert:
        return str(b)
    return b

print(fib(42))    # the result is the number 267914296
print(fib("42"))  # the result is the string "267914296"
```

2. Το τελευταίο `if` στο παραπάνω πρόγραμμα μπορεί να γραφεί:

```
return str(b) if convert else b
```

που είναι αντίστοιχο του τελεστή `a ? b : c` στη C/C++. Στην Python θα το γράφαμε `b if a else c`.

3. Έλεγχος τύπων: η συνάρτηση `type` επιστρέφει τον τύπο μίας τιμής. Π.χ.

```
type(42) == int
type("hello") == str
type(True) == bool
```

4. Καλύτερος έλεγχος τύπων που ακολουθεί την ιεραρχία των κλάσεων της Python: με τη συνάρτηση `isinstance`. Η διαφορά φαίνεται με τιμές του τύπου `bool`, ο οποίος όπως έχουμε πει είναι υποτύπος του `int`.

```
type(True) == bool      # επιστρέφει True
type(True) == int       # επιστρέφει False
isinstance(True, int)   # επιστρέφει True
isinstance(True, bool)  # επιστρέφει True, επίσης!
```

Η συνάρτηση `isinstance` μπορεί να χρησιμοποιηθεί και για να ελέγξει αν μία τιμή ανήκει σε κάποιον από πολλούς εναλλακτικούς τύπους:

```
isinstance(x, (int, float, str))
```

5. Χρησιμοποιώντας την `isinstance`, η παραπάνω συνάρτηση `fib` γράφεται ως εξής:

```
def fib(n):
    convert = False
    if isinstance(n, str):
        n = int(n)
        convert = True
    a, b = 0, 1
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return str(b) if convert else b
```

Συναρτήσεις και default τιμές παραμέτρων

1. Γενικευμένοι αριθμοί Fibonacci. Οι δύο πρώτοι όροι της ακολουθίας είναι οι παράμετροι `start_a` και `start_b`.

```
def fib(n, start_a=0, start_b=1):
    a, b = start_a, start_b
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return b
```

2. Τώρα μπορεί να κληθεί με οποιονδήποτε από τους παρακάτω τρόπους:

```
print(fib(100))           # start_a = 0, start_b = 1
print(fib(100, 17, 42))
print(fib(100, 17))       # start_b = 1
```

3. Επίσης, μπορεί κανείς να ονομάζει τις παραμέτρους κατά την κλήση και να τις δίνει με διαφορετική σειρά:

```
print(fib(100, start_b=42, start_a=17))
print(fib(100, start_b=42))               # start_a = 0
print(fib(start_a=17, n=100))             # start_b = 1
```

Εμβέλεια μεταβλητών

1. Τοπικές και καθολικές (global) μεταβλητές:

```
a = 17
print(a)

def fib(n):
    a, b = 0, 1
    if n == 0: return 0
    for i in range(n-1):
        c = a+b
        a, b = b, c
    return b

print(fib(7))
print(a)
print(b)
```

   Το εξωτερικό `a` (global) είναι διαφορετικό από το εσωτερικό `a` (local).

2. Κανόνες εμβέλειας

```
a = 42              # this a is global

def reasonable():
    print(a)

def what():
    a = 17
    print(a)

reasonable()        # prints 42
print(a)            # prints 42
what()              # prints 17
print(a)            # prints 42, what???
```

   Η διαφορά βρίσκεται στο ότι η συνάρτηση `reasonable` δεν αναθέτει στη μεταβλητή `a`, ενώ η `what` αναθέτει. Η ανάθεση κάνει την Python να πιστεύει ότι η μεταβλητή πρέπει να είναι local στη `what`. Αυτό μπορεί να διορθωθεί με χρήση του `global`:

```
a = 42              # this a is global
```

```
def what():
    global a
    a = 17
    print(a)

what()              # prints 17
print(a)            # prints 17
```

3. Περισσότερα επίπεδα εμβέλειας:

```
def foo(n):
    print(n)
    def bar():
        n = 17
        print(n)
    bar()
    print(n)

foo(42)
```

τυπώνει:

```
42
17
42
```

4. Το παραπάνω με χρήση του `global`:

```
def foo(n):
    print(n)
    def bar():
        global n
        n = 17
        print(n)
    bar()
    print(n)

foo(42)
print(n)
```

τυπώνει:

```
42
17
42
17
```

5. Το ίδιο με χρήση του `nonlocal`:

```
def foo(n):
    print(n)
    def bar():
        nonlocal n
        n = 39
        print(n)
    bar()
    print(n)

n = 17
foo(42)
print(n)
```

τυπώνει:

```
42
39
39
```

```
17
```

Προσέξτε ότι η μεταβλητή στην οποία ανατέθηκε το 39 ήταν η τοπική της `foo`, όχι η global, η οποία παρέμεινε ανέπαφη.

Συμβολοσειρές

1. Αποτελούνται από χαρακτήρες. Η αρίθμηση ξεκινάει από το μηδέν!

   ```
   s = "hello world"
   print(s[4])           # prints 'o'
   ```

2. Τα περιεχόμενά τους δεν αλλάζουν (strings are immutable)

   ```
   s[4] = 'x'            # ERROR!
   ```

   αλλά αυτό είναι OK (φτιάχνει νέο string):

   ```
   s = s[:4] + "x" + s[5:]
   print(s)
   ```

3. "Φέτες" (slices)

   ```
   print(s[:4])      # "hell"        --- from start until 4
   print(s[4:])      # "o world"     --- from 4 until end
   print(s[4:8])     # "o wo"        --- from 4 until 8
   print(s[::2])     # "hlowrd"      --- every second char (step = 2)
   print(s[::-1])    # "dlrow olleh" --- in reverse (step = -1)

   def palindrome(s):
       return s == s[::-1]   # that was quick...
   ```

Πλειάδες (tuples)

Αντιπαράβαλε με τα παραπάνω (strings):

1. Αποτελούνται από οτιδήποτε

   ```
   s = (1, 2, "what", 3)
   print(s[1])               # prints 2

   s = tuple("hello world")  # ('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
   print(s[4])               # prints 'o'
   ```

2. Τα περιεχόμενά τους δεν αλλάζουν (tuples are immutable)

   ```
   s[4] = 'x'            # ERROR!
   ```

   αλλά αυτό είναι OK (φτιάχνει νέο tuple):

   ```
   s = s[:4] + ("x",) + s[5:]
   print(s)
   ```

   Προσέξτε το κόμμα στο `("x",)` που ορίζει ένα tuple με μήκος 1.

3. "Φέτες" (slices)

   ```
   print(s[:4])      # ('h', 'e', 'l', 'l')
   print(s[4:])      # ('o', ' ', 'w', 'o', 'r', 'l', 'd')
   print(s[4:8])     # ('o', ' ', 'w', 'o')
   print(s[::2])     # ('h', 'l', 'o', 'w', 'r', 'd')
   print(s[::-1])    # ('d', 'l', 'r', 'o', 'w', ' ', 'o', 'l', 'l', 'e', 'h')
   ```

Λίστες (lists)

Αντιπαράβαλε με τα παραπάνω (strings και tuples):

1. Αποτελούνται από χαρακτήρες

```
s = [1, 2, "what", 3]
print(s[1])              # prints 2

s = list("hello world")  # ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
print(s[4])              # prints 'o'
```

2. Τα περιεχόμενά τους αλλάζουν (lists are mutable)

```
s[4] = 'x'           # OK!
print(s)
```

και αυτό είναι OK αλλά φτιάχνει νέα λίστα:

```
s = s[:4] + ["x"] + s[5:]
print(s)
```

3. "Φέτες" (slices)

```
print(s[:4])      # ['h', 'e', 'l', 'l']
print(s[4:])      # ['o', ' ', 'w', 'o', 'r', 'l', 'd']
print(s[4:8])     # ['o', ' ', 'w', 'o']
print(s[::2])     # ['h', 'l', 'o', 'w', 'r', 'd']
print(s[::-1])    # ['d', 'l', 'r', 'o', 'w', ' ', 'o', 'l', 'l', 'e', 'h']
```

4. Μετατροπή λίστας σε string (συνένωση πολλών strings)

```
print("".join(s))   # "hello world"
print("-".join(s))  # "h-e-l-l-o- -w-o-r-l-d"
```

Γεννήτριες (generators)

1. List comprehensions

```
b = [i*i for i in range(10)]
print(b)    # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

b = [i*i for i in range(10) if i*i%2==1]
print(b)    # [1, 9, 25, 49, 81]
```

2. Η έκφραση `i*i for i in range(10)` παραπάνω είναι μία γεννήτρια (generator), όπως και το ίδιο το `range(10)`. Γεννάει μία ακολουθία από τιμές, τις οποίες μπορούμε να τοποθετήσουμε σε μία λίστα ή άλλη δομή δεδομένων, να τις διατρέξουμε με ένα `for` loop, κ.λπ. Η γεννήτρια αυτή αντιστοιχεί στο μαθηματικό σύνολο $\{i^2 \mid i \in \{0..9\}\}$.

3. Μπορεί κανείς να φτιάξει μία δική του γεννήτρια, ορίζοντας μία συνάρτηση που αντί για `return` κάνει `yield` τις τιμές που γεννάει:

```
def gen_fib(n):
    a, b = 0, 1
    yield 0
    yield 1
    for i in range(n-1):
        c = a+b
        a, b = b, c
        yield b

for i in gen_fib(42):
    print(i)        # prints 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
```

4. Μπορεί επίσης να κατασκευάσει "άπειρες" γεννήτριες, οι οποίες παράγουν διαρκώς τιμές:

```
def gen_all_fib():
    a, b = 0, 1
    yield 0
    yield 1
    while True:
        c = a+b
        a, b = b, c
        yield b
```

Ο παρακάτω κώδικας τυπώνει τους πρώτους 100 αριθμούς Fibonacci (γιατί σταματά με το `break`), και στη συνέχεια τυπώνει τον 101ο με χρήση της συνάρτησης `next`, που φέρνει το επόμενο στοιχείο μίας γεννήτριας.

```
i = 0
g = gen_all_fib()
for x in g:
    print(x)
    i += 1
    if i >= 100: break
print(next(g))
```

Λεξικά (dictionaries)

1. Γνωστά και ως συσχετιστικοί πίνακες (associative arrays). Είναι ουσιαστικά arrays που τα στοιχεία τους προσπελάζονται μέσω κλειδιών (keys) οποιουδήποτε τύπου — όχι μόνο με ακέραιους όπως οι συμβολοσειρές, οι πλειάδες και οι λίστες.

   ```
   d = {}
   d["yes"] = "ναι"
   d["no"] = "όχι"

   print(d["yes"])
   ```

   Το παρακάτω προκαλεί εξαίρεση τύπου `KeyError`:

   ```
   print(d["maybe"])
   ```

2. Οι τύποι των κλειδιών μπορούν να είναι (σχεδόν) οτιδήποτε:

   ```
   d = {
     "maybe": "ίσως",
     "why not?": "γιατί όχι;"
   }
   d[1, 2, 3] = "ναι"
   d[4, 5, 6, 7] = "όχι"
   print(d[4, 5, 6, 7])
   ```

   Πρέπει όμως είτε να είναι immutable (δηλαδή όχι λίστες ή άλλα λεξικά) είτε να ορίζουν τις μεθόδους `__hash__` και `__eq__`.

3. Μπορούμε να διατρέξουμε όλα τα κλειδιά ενός λεξικού:

   ```
   for key in d:
       print(key)
   ```

   και να τυπώσουμε και τις τιμές:

   ```
   for key in d:
       print(key, d[key])
   ```

   ή καλύτερα:

   ```
   for key, value in d.items():
       print(key, value)
   ```

Σύνολα

1. Αναπαριστούν μαθηματικά σύνολα με στοιχεία (σχεδόν) οποιουδήποτε τύπου (πρέπει να πληρούν τις ίδιες προϋποθέσεις με τα κλειδιά ενός λεξικού).

   ```
   s = set()
   s.add(1)
   s.add(2)
   s.add(3)
   print(2 in s)   # prints True
   print(4 in s)   # prints False
   s.remove(2)
   print(2 in s)   # prints False now
   ```

2. Πράξεις συνόλων:

```
s1 = set([1, 2, 3])
s2 = {3, 4, 5}        # equivalent to set([3, 4, 5])

print(s1 | s2)        # union: {1, 2, 3, 4, 5}
print(s1 & s2)        # intersection: {3}
print(s1 - s2)        # difference: {1, 2}
```

3. Set comprehensions:

```
s = set(i*i for i in range(10))     # {0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

ή ισοδύναμα:

```
s = {i*i for i in range(10)}
```

4. Τα σύνολα (set) είναι mutable. Υπάρχει και η immutable εκδοχή τους (frozenset) που μπορούν να χρησιμοποιηθούν ως κλειδιά σε λεξικά ή ως στοιχεία σε άλλα σύνολα.

```
s1 = frozenset([1, 2, 3])   # two immutable sets
s2 = frozenset([4, 5, 6])
s = {s1, s2}                # and a (mutable) set containing them
```

Εξαιρέσεις

1. Η χρήση του `d["haha"]` σε ένα λεξικό `d` που δεν περιέχει κλειδί `"haha"` προκαλεί μία εξαίρεση τύπου `KeyError`. Η εξαίρεση αυτή μπορεί να προκληθεί και ρητά, με την εντολή `raise`:

```
raise KeyError
```

2. Η Python υποστηρίζει ένα μηχανισμό χειρισμού εξαιρέσεων, με τη δομή `try ... except ...`

```
try:
    print(d["oui"])
    print(d["haha"])
    print(d["non"])
except:
    print("something wrong happened")
```

Το παραπάνω τυπώνει:

```
yes
something wrong happened
```

γιατί όταν προκαλείται η εξαίρεση `KeyError` στη δεύτερη γραμμή του `try`, η ροή ελέγχου μεταφέρεται στο σκέλος `except` και το πρόγραμμα συνεχίζει από εκεί.

3. Χειρισμός συγκεκριμένων τύπων εξαιρέσεων:

```
try:
    print(d["oui"])
    print(d["haha"])
    print(d["non"])
except KeyError:
    print("something wrong happened")
```

Όμως, αυτό το σκέλος `except` δε θα πιάσει κάποια εξαίρεση άλλου τύπου, όπως για παράδειγμα μία διαίρεση με το μηδέν:

```
try:
    print(d["oui"])
    print(1//0)
    print(d["haha"])
    print(d["non"])
except KeyError:
    print("something wrong happened")
```

Για περισσότερους τύπους εξαίρεσης:

```
try:
    print(d["oui"])
    print(1//0)
    print(d["haha"])
    print(d["non"])
except (KeyError, ZeroDivisionError):
    print("something wrong happened")
```

ή και για διαφορετικό τρόπο χειρισμού κάθε μίας:

```
try:
    print(d["oui"])
    print(1//0)
    print(d["haha"])
    print(d["non"])
except KeyError:
    print("something wrong happened")
except ZeroDivisionError:
    print("something wrong with your math")
```