Technical University of Lodz
Institute of Electronics

Algorithms and Data Structures

8. Lists

Łódź 2012

Exercise – Sieve of Eratosthenes

The ***Sieve of Eratosthenes*** is an old algorithm that makes a list of primes:

- List the integers begining with 2.
- Circle 2 (it must be prime) and cross out all its multiples (they can not be prime).
- Circle the next integer that is not crossed out (3) and cross out its multiples.
- Repeat.

Nesting

```
1   # sieve.py
2
3   def sieve(n):
4       primes = list(range(2, n + 1))
5       for k in range(2, n + 1):
6           if k in primes:
7               for j in range(2 * k, n + 1, k):
8                   if j in primes:
9                       primes.remove(j)
10      return primes
11
12  def main():
13      n = int(input("Enter upper limit: "))
14      print "The primes up to", n, "are:\n", sieve(n)
15
16  main()
17
```

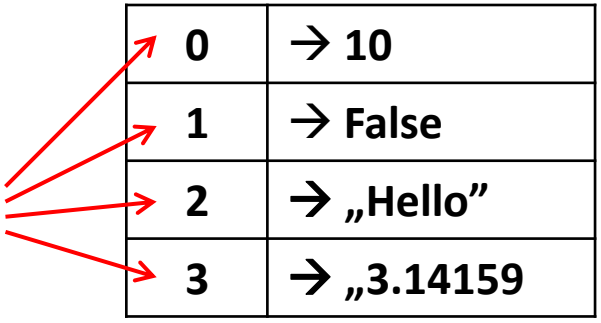
Python **lists**

A Python **list** is a data type that stores a sequence of items. They are similar to strings. But while every item in a string is a single character, list elements may be of any type, e.g.

>>> a_list = [10, False, „Hello”, 3.14159].

A pair of empty brackets [] denotes an empty list.

Pointers (references to memory location)



Python **lists**

The following operations all work the same for lists as they do for strings:
- Indexing using []
- Slicing using [i:j:k]
- Concatenation using +
- Repeated concatenation using *n
- **for** loops
- Accumulation loops
- **in** and **not in**

Caution: Concatenation only works between objects of the same type.
For example,
>>> items = [1, 4, 7] + „abc”
will cause an error because the type of [1, 4, 7] (list) does not match the type of „abc” (string).

Lists Are Also *Not* Like Strings

These operations work with lists but not strings:

- items[i] = ***x*** # Replace items[i] with ***x***
- items[i:j] = ***newitems*** # Replace items in slice with with ***newitems***
- items[i:j:k] = ***newitems*** # Replace items in slice with with ***newitems***

- **del** items[i] # Remove items[i]
- **del** items[i:j] # Remove items in slice
- **del** items[i:j:k] # Remove items in slice

Random functions for lists

The random module includes these functions for lists

choice(items) # One random element from the ***items***

shuffle(items) # Randomly shuffle the elements of the ***items***

The choice() function also works on strings; the shuffle function does not.

Objects and objects methods

The syntax to call a method from an object is called **dot notation**:

<object>.<method>(<arguments>)

List methods

If **items** is a list object, these are some of the methods that may be called on it:

```
>>> items.append(x)  # Add item x to the end of items
>>> items.insert(i, x)   # Insert item x into items at index i
>>> items.pop()          # Remove and return the last item in items
>>> items.pop(i)         #  Remove and return items[i]
>>> items.remove(x)   #  Remove items x from items.
>>> items.reverse()      #  Reverse the order of the elements in items.
>>> items.sort()         # Sort the list items.
```

All these methods modify the list they are called on, and only .pop() returns anything.

Numpy **ndarray**

Operating on the elements in a list can be done through iterative loops, which is computationally inefficient in Python.

The **Numpy** package enables users to overcome this shortcomming by providing a data object called **ndarray**.

The **ndarray** is similar to a list, but only the same type of element can be stored in each column, i.e. all elements must be floats, integers or strings. Besides this limitation, ndarray speeds up the calculations significantly.

The **Numpy** package combines high computational efficiency with Python flexibility – so the language can be used for scientific purposes where arrays are the basic data structures.

Numpy **ndarray** operation time


```
1  # c_time: List versus Numpy ndarray
2
3  import numpy as np
4  import timeit as tim
5
6  arr = np.arange(1e7)  #Create and array with 10^7 elements
7  start = tim.time.clock()
8  b = arr * 1.1   #Multiplication of each array element by a scalar
9  tarr = (tim.time.clock() - start) * 1000
10
11 larr = arr.tolist()   #Converting ndarray to list
12
13 def mult(alist, scalar):        #A loop to multiply each
14     for i, val in enumerate(alist): #list element by a scalar
15         alist[i] = val * scalar
16     return alist
17
18 start = tim.time.clock()
19 mult(larr, 1.1)
20 tlist = (tim.time.clock() - start) * 1000
21 print "Numpy ndarray multiplication time is %.1fms\n" % (tarr)
22 print "Python list multiplication time is %.1fms\n" % (tlist)
23 print "The ndarray operation is %.1f times faster \
24 in this example." % (tlist/tarr)
25
```

Literature

Brian Heinold, Introduction to Programming Using Python, Mount St. Mary's University, 2012 (*http://faculty.msmary.edu/heinold/python.html*).

Brad Dayley, Python Phrasebook: Essential Code and Commands, SAMS Publishing, 2007 (dostępne też tłumaczenie: B. Dayley, Python. Rozmówki, Helion, 2007).

Mark J. Johnson, A Concise Introduction to Programming in Python, CRC Press, 2012.

Eli Bressert, SciPy and Numpy, O'Reilly, 2012