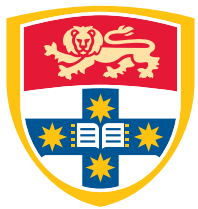
Introduction to Programming

Dr. John Stavrakakis

School of Computer Science, University of Sydney



Copyright Warning

Week 4: Loops, arrays and lists

We will cover: Using `continue`, looping with `while`, `break` and `continue`, and introduction to arrays and lists

You should read: §§1.4 of Sedgewick

Lecture 7: Arrays, Lists and Iteration

Simple structures for storing data in your programs

Information of the same type

- Write a program that displays the name of 50 students
- Write a program that calculates and displays the total from 100 values
- Write a program that counts how many people in this room have a birthday today

What is the data type for each problem?

If we write these programs we need to use one variable to store each value

Arrays

An array is a *contiguous block of memory* containing multiple values of the same type.

Array solves the problem of having many variables.
e.g. you don't have to declare 100 int variables.

```
1    # bad style bro
2    value001 = 32
3    value002 = 18
4    value003 = 4
5    ...
6    value100 = 6
```

Idea: Instead of using value046, use the 46th value from the memory of the array

Arrays — ideas about memory

Lists are not Arrays, Arrays are not Lists - Python

We need to first understand the principles of arrays to understand how to use the list data structure

Python presents the usage of a list as an array. However, lists have different operations and internal memory organisation.

We will first present the array ideas, then the list ideas

Arrays — creation

A list is specified as two square brackets [and]. An empty array

```
1    x = [ ]
```

Obtaining the length of the array

```
1    x_len = len(x)
```

Printing the empty array

```
1    print("the array is:" + str(x)  )
2    print("the length of this array is: " + str( len(x) )  )
```

There is not much to do with an empty array. We still need it

- It is used to describe the data type (initial value is a *list*)
- It is defining a collection of things, where it represents an empty collection
- As we will see, it is useful for further operations with *list*s

Arrays — creation (cont.)

A single element array

```
1 x = [ 47 ]
```

Accessing the only element in the array

```
1 first_element = x[0]
```

Printing the array

```
1 print("the array is:" + str(x))
2 print("the length of this array is: " + str(len(x)))
3 if len(x) > 0:
4     print("the first element of this array is: " + str(x[0]))
```

oops!

```
1 second_element = x[1]
```

Arrays — creation (cont.)

A two element array

```
1 x = [ 47, 25 ]
```

A multi-element array is defined as comma separated *objects* of the same type within the square brackets [and]

```
1 x = [ -3, -2, -1, 0, 1, 2, 3 ]
```

Arrays — creation (cont.)

Accessing each element

```
1  print("the array is:" + str(x)  )
2  print("the length of this array is: " + str(len(x)))
3  if len(x) > 0:
4      print("first element is: " +   str(x[0]))
5  if len(x) > 1:
6      print("second element is: " +  str(x[1]))
7  if len(x) > 2:
8      print("third element is: " +   str(x[2]))
9  if len(x) > 3:
10     print("fourth element is: " +  str(x[3]))
11 if len(x) > 4:
12     print("fifth element is: " +   str(x[4]))
13 if len(x) > 5:
14     print("sixth element is: " +   str(x[5]))
15 if len(x) > 6:
16     print("seventh element is: " + str(x[6]))
```

Arrays — Initialisation

Initialise each value of the array at declaration

```
1 names = [ "Bill", "Ted", "Larry" ]
2 print(names[0])
3 print(names[1])
4 print(names[2])
```

which creates an array of String objects, the contents of which are the three strings “Bill”, “Ted” and “Larry”. The length of the array is set *implicitly* to the number of elements in the strings provided in brackets.

Arrays — index

The array contains multiple values of the same type

Each element of the array is described by an index number.

The numbering starts at 0, not 1. The first element of the array is referred to as the "*the zero-th element*" or "0th element" of the array.

To access the *i*th element of an array `A` we just use `A[i]`.

If `i` is more or less than one whole length of array then Python will throw an exception [1]

If `i` is bigger than the array size initialised, Python will throw an exception

[1] Write a program to test this behaviour: see what happens.

Using the array

Predefine the number of elements in the array using the list operator *

```
1  values = [0] * 4
2
3  values[0] = 1
4  values[1] = 2
5  values[2] = 3
6  values[3] = 4
7
8  print("first value: " + str(values[0]))
9  print("last value: " + str(values[3]))
10
11 sum = values[0] + values[1] + values[2] + values[3]
12 print("sum: " + str(sum))
```

len: Number of items in a Python object

There is a very nice feature of the data structures in Python that is built-in: for any list *x* that you define, the value `len( x )` is the length of the list.

That means if you declare an array of `String` items like this:

```
1 fruits = [ 'Apple', 'Banana', 'Cherry' ]
```

the length of `fruits` is 3.

Accessing the last element

```
1 fruits = [ "Apple", "Banana", "Cherry" ]
2 print("first fruit is " + fruits[0] )
3 print("second fruit is " + fruits[1] )
4 print("last fruit is " + fruits[ len(fruits) - 1] )
```

IndexError

When you attempt to run a program that assumes you have some `String` arguments, but don't give it any, you will get an *Exception* called `IndexError`

```
1 import sys
2 print("The third argument is " + sys.argv[3])
```

```
~> python MissingArgs.py
Traceback (most recent call last):
  File "MissingArgs.py", line 2, in <module>
    print("The third argument is " + sys.argv[3])
IndexError: list index out of range
```

See? Look back at the code: the element to be accessed is the 4th (`argv[3]`) but the array `argv` has *no arguments from the command line*. So its length is 1 and there isn't an `argv[3]`. [2]

[2] confirm with `print( len(sys.argv) )`

ArrayOfStrings.py

```
1 import sys
2 i = 0
3 while i < len(sys.argv):
4     argument = sys.argv[i]
5     print(argument)
6     i = i + 1
```

Running this gives the following behaviour:

```
~> python ArrayOfStrings.py one two
ArrayOfStrings.py
one
two
```

Arrays of Objects

You can have arrays of *anything*: we have used `int` type and `String` type, but consistently.

As the programmer, you are deciding what memory you need to have and then how it is processed.

- print an array of integers
- print an array of strings
- print a mixed array of integers and strings...oh wait...does this work?

Mixing different objects into the same array can become problematic. There are quick and dirty solutions to this, however, they make both testing and debugging much harder.

Use the same type of object in your arrays!

When to use Arrays (True/False)?

a todo list of tasks

stock market price of one company over a period of time

number of stairs at the Opera House

number of passengers in each of 8 train carriages

100 different variables used in automobile software e.g. fuel, passenger light on, odometer etc.

a catalogue of 100 book titles

a 3,000 word essay

List

A data structure to represent a collection of items

Lists

A multi-element array is defined as comma separated ***objects of the same type*** *within the square brackets [and]*

The list data structure has different characteristics than an array

- The size of the list is not fixed, it is mutable
- Any type of object can be be included
- Elements can be inserted or removed in any position
- Lists can be concatenated together
- A subset of the list can be created from the list
- Very different memory structure

Lists – creation

List is the same initialisation style. We show the difference between Arrays and Lists

```
1  # declare an initialise a list of 1 integer and 1 string
2  z = [ 33, "the number is 33" ]
3  print("array z is " + str(z) )
4
5  # declare an initilise a list of 5 integer
6  w = [0] * 5
7  print("array w is " + str(w) )
```

```
array z is [33, 'the number is 33']
array w is [0, 0, 0, 0, 0]
```

List operation – search with loops

Searching is the most fundamental tool of any programmer.

We want to find the particular value among a collection of items. There may be different goals:

- To learn if it exists,
- To learn where it is,
- To learn how many exist

A search requires reading each element of the collection until the search criteria has been satisfied.

List operation – search with loops (cont.)

Consider this list

```
1 animals = [ "armadillo", "bear", "kangaroo", "okapi", "squirrel"
      ]
```

Is there a bear in there? Yes/No

Where is the kangaroo? what is the index?

How many squirrels?

List operation – search with loops (cont.)

Is there a bear in there? Yes/No

```
1  animals = [ "armadillo", "bear", "kangaroo", "okapi", "squirrel"
       ]
2  has_bear = False
3  while True:
4      ...
5
6
7
8
9
10
11
12
13
14 if has_bear == True:
15     print("There is a bear in there!")
16 else
17     print("no bear, unable to find bear, cannot bear")
```

List operation – search with loops (cont.)

Where is the kangaroo? what is the index?

```
1 animals = [ "armadillo", "bear", "kangaroo", "okapi", "squirrel"
      ]
```

List operation – search with loops (cont.)

How many squirrels?

```
1 animals = [ "armadillo", "bear", "kangaroo", "okapi", "squirrel"
      ]
```

List operation – insert

Insert new elements at a given position.

list.insert(index, object) where 0 <= index <= len(list)

Insert at the end of the list

```
1 countries = [ "Malaysia", "Mozambique", "Benin", "Ukraine" ]
2 countries.insert(len(countries), "South Sudan")
3 print(countries)
```

Insert as the very first element of the list

```
1 letters = [ "b", "c", "d" ]
2 letters.insert(0, "a")
3 print(letters)
```

List operation – insert (cont.)

Insert in the middle

```
1 index = ?
2 days = [ "Sunday", "Monday", "Tuesday", \
3           "Wednesday", "Friday", "Saturday" ]
4 days.insert( index , "Thursday")
5 print(days)
```

List operation – insert (cont.)

What is the list after executing the following code?

```
1 mylist = []
2 x = 1
3 while x < 10:
4     mylist.insert( len(mylist), x)
5     x += 1
```

What is the list after executing the following code?

```
1 mylist = []
2 x = 1
3 while x < 10:
4     mylist.insert( 0, x)
5     x += 1
```

List operation – remove

Searches the list and removes the *first* object matching the value

```
list.remove(object)
```

Only one! and it is the first one

```
1 numbers = [ 7, 10, 01, 7 ]
2 numbers.remove(7)
3 print(numbers)
```

List operation – remove (cont.)

Searching for the *value* of the object given to the remove function

```
a = 600+100
b = "700"
c = 700+0.0
d = 700
mixed_list = [ a, b, c, d ]
mixed_list.remove(700)
mixed_list.remove(700)
print(mixed_list)
```

Error is thrown if no such object.

```
numbers = [ 7, 7, 7 ]
numbers.remove(8)
print(numbers)
```

How can we avoid errors caused by remove in advance?

List operation – remove (cont.)

Reordering the list using insert + remove

What is the first day of the week?

```
1 days = [ "Sunday", "Monday", "Tuesday", \
2          "Wednesday", "Thursday", "Friday", \
3          "Saturday" ]
4 days.remove("Sunday")
5 days.insert( len(days), "Sunday" )
6 print(days)
```

```
1 parcels = [ 'normal', 'fragile', 'priority', \
2             'normal', 'priority' ... ]
3 delivery_truck = [ ]
4 # search through the collection for priority parcels
5 # remove one priority item from parcels received
6 # add the priority item to the delivery_truck
7 # until no more priority parcels
8 # then process normal/fragile parcels
```

List operation – concatenation

So far the + operator is used for arithmetic and string operations. Lists can be concatenated in the same way

Example

```
1 a = [ 1, 2, 3 ]
2 b = [ 4, 5, 6 ]
3 ab = a + b
4 print(ab)
```

Recall that lists may contain zero or more elements.

```
1 empty = []
2 one = [ 1 ]
3 two = [ "one", "two" ]
4 print( empty )
5 print( empty + empty)
6 print( one + empty + one )
7 print( empty + two + one )
```

Supplementary

Further list operations

List operation – search with `in`

The `in` keyword can be used to test if an object can be found within a list.

boolean expression := Object in List

```
1  mylist = [ "sand", 45, 3.14, "teacake" ]
2
3  if "teacake" in mylist:
4      print("found teacake!")
5  else:
6      print("oh no. no teacake")
7
8  if "almond" in mylist:
9      print("found an almond!")
10 else:
11     print("oh no. no almond")
```

In our case for a list, it will search for the equivalent value OR the object reference.

List operation – search with `in` (cont.)

Spot the error

```
1  ticket = [ 14, 7, 23, 9 ]
2
3  number_drawn = input("enter the next lottery number: ")
4  if number_drawn in ticket:
5      print("you have the number!")
```

List operation – search with `in` (cont.)

Spot the error

```
1  ticket = [ 14, 7, 23, 9 ]
2
3  number_drawn = input("enter the next lottery number: ")
4  if ticket in number_drawn:
5      print("you have the number!")
```

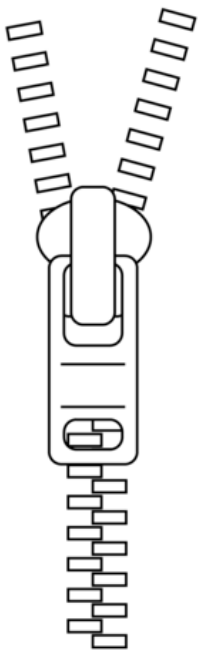
List operation – search with `in` (cont.)

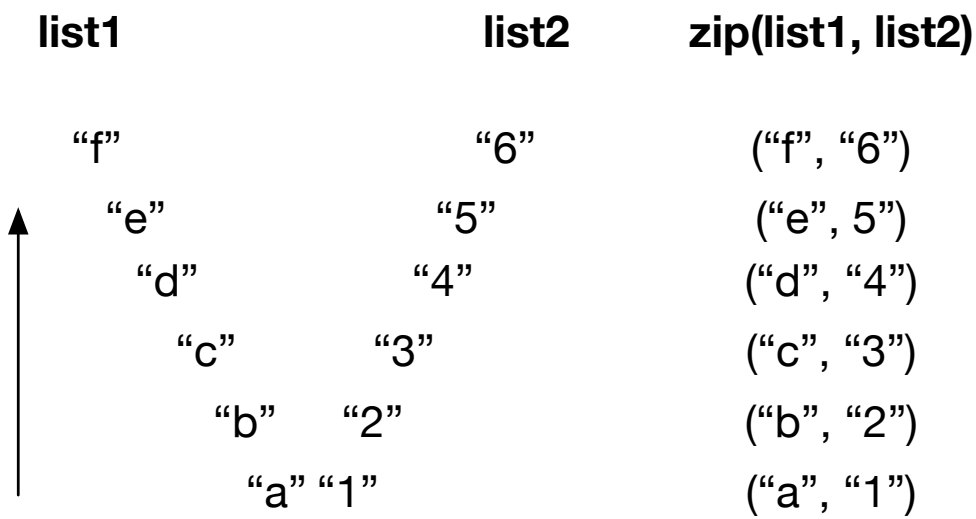
Spot the error

```
1  # multi ticket edition
2  ticket1 = [ 14, 7, 23, 9 ]
3  ticket2 = [ 40, 3, 4, 38 ]
4  lottery = [ ticket1, ticket2 ]
5
6  number_drawn = input("enter the next lottery number: ")
7  if number_drawn in lottery:
8      print("you have the number!")
```

List operation – zip

zip - like a zipper!





List operation – zip (cont.)

```
1 list1 = [ "a", "b", "c", "d", "e", "f" ]
2 list2 = [ "1", "2", "3", "4", "5", "6" ]
3
4 # create a zip object
5 result = zip(list1, list2)
6
7 # we need to make a list from the zip object
8 new_list = list(result)
```

How to describe zip?

- make a list of pairs
- merge two lists in sequence
- create a list that is the union of two lists, where the union operation is only for the first k elements of each list, where k is the smaller of the two list lengths

List operation – zip (cont.)

zip can help define an association between related objects

```
1  name_list = [ "Sunset", "Amber Jewel", \
2                "Teagan", "Queen Garnet" ]
3  weight_list = [ 0.872, 0.972, 0.797, 1.332 ]
4  unit_cost_list = [ 2.99, 6.99, 3.99, 17.99 ]
5  price_list = [2.61, 6.79, 3.18, 23.96]
6
7  # present invoice of minimum information
8  simple_bill = list(zip(name_list, price_list))
9  print(simple_bill)
10
11 # present invoice of all information with extra markup
12 itemised_bill_full = list( \
13     zip(name_list, weight_list, ["kg @ $"] *4, \
14         unit_cost_list , ["/kg = $"] *4, price_list))
15 print(itemised_bill_full)
16
17 # just one item
18 print(itemised_bill_full[1])
```

List operation – split

A String operation. Split the string into a list of smaller strings based on the delimiter `sep`

str.split(sep=None, ...) where sep represents the delimiter found

We have seen this in week 1. There is a list returned from this function. Each element of the list is a string. We can choose how the string is broken up.

```
1 text = "The unit of measure, a zeptonewton, \
2 is equal to one billionth of a trillionth of a newton"
3 words = text.split(sep=' ')
4 words2 = text.split(sep='on')
5 words3 = text.split(sep='ll')
```

List operation – split (cont.)

More practical with structured text data

```
places="50°04′N   19°56′E; Kraków, Poland; 56°07′N   101°36′E; Bratsk, Russia; 4
6°49′N   71°13′W; Quebec City, Canada; 67°34′S   68°08′W; Rothera, United Kingdo
m; 22°17′N   114°10′E; Hong Kong,China; 24°34′N   81°47′W; Key West, United Stat
es; 43°50′N   87°36′E; Ürümqi,China; 7°12′S   39°20′W; Juazeiro do Norte, Brazil
; 14°33′N   121°02′E; Makati, Philippines"
```

```
1  # each place is *two* strings
2  place_list = places.split(';')
3
4  # search for Canada in the list. Use a loop!
5  canadian_place_coord = places_list[ ? ]
6  canadian_place_name = places_list[ ? ]
7
8  # split this up as coordinate, city, country
9  latitude = canadian_place_coord.split('   ')[0]
10 longitude = canadian_place_coord.split('   ')[1]
11
12 city = canadian_place_name.split(',')[0]
13 country = canadian_place_name.split(',')[1]
```

References

Image: zipper