

COMP110: Principles of Computing

9: Data Structures II

Exercise Sheet iii

Due **tomorrow**

Basic containers in Python

Memory allocation

- ▶ Memory is allocated in **blocks**
- ▶ The program specifies the size, in bytes, of the block it wants
- ▶ The OS allocates a **contiguous** block of that size
- ▶ The program owns that block until it frees it
- ▶ Forgetting to free a block is called a **memory leak** (not really possible in Python, but a common bug in C++)
- ▶ Blocks can be allocated and deallocated at will, but can **never grow or shrink**

Containers

- ▶ Memory management is hard and programmers are lazy
- ▶ Containers are an **abstraction**
 - ▶ Hide the details of memory allocation, and allow the programmer to write simpler code
- ▶ Containers are an **encapsulation**
 - ▶ Bundle together the data's representation in memory along with the algorithms for accessing it

Arrays

- ▶ An **array** is a contiguous block of memory in which objects are stored, equally spaced, one after the other
- ▶ Each array element has an **index**, starting from zero
- ▶ Given the address of the 0th element, it is easy to find the i th element:

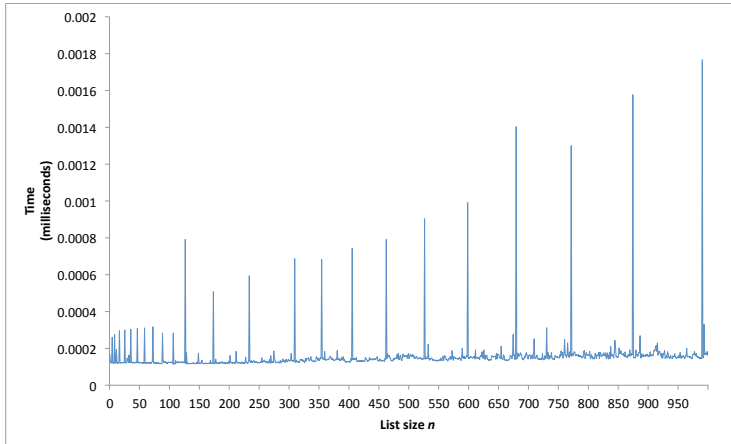
$$\text{address}_i = \text{address}_0 + (i \times \text{elementSize})$$

- ▶ E.g. if the array starts at address 1000 and each element is 4 bytes, the 3rd element is at address $1000 + 4 \times 3 = 1012$
- ▶ Accessing an array element is **constant time** $O(1)$

Lists

- ▶ An array is a block of memory, so its size is **fixed** once created
- ▶ A **list** is a variable size array
- ▶ When the list needs to change size, it **creates** a new array, **copies** the contents of the old array, and **deletes** the old array
- ▶ Implementation details: <http://www.laurentluce.com/posts/python-list-implementation/>

Time taken to append an element to a list of size n



Operations on lists

- ▶ **Appending** to a list is **amortised constant time**
 - ▶ Usually $O(1)$, but can go up to $O(n)$ if the list needs to change size
- ▶ **Inserting** anywhere other than the end is **linear time**
 - ▶ Can't just insert new bytes into a memory block — need to move all subsequent list elements to make room
- ▶ Similarly, **deleting** anything other than the last element is **linear time**

Tuples

- ▶ Tuples are like lists, but are **immutable**
 - ▶ Read-only
 - ▶ Once created, can't be changed
- ▶ Useful for storing sequences of values where adding, inserting, deleting or changing individual values does not make sense
 - ▶ E.g. xy coordinates, RGB colours, ...
- ▶ Create tuples with `()`, just as you create lists with `[]`
 - ▶ Exception: a single element tuple is created as `(foo,)` because `(foo)` would be interpreted as a bracketed expression
- ▶ Can often omit the parentheses entirely, e.g.

```
my_tuple = 1,2,3
```

Unpacking

If `foo` is a list or tuple of length 4, the following are equivalent:

```
a, b, c, d = foo
```

```
a = foo[0]  
b = foo[1]  
c = foo[2]  
d = foo[3]
```

- Unpacking requires the number of elements to match exactly — if `foo` has more than 4 elements, the code on the left will give an error

One weird trick (Java programmers hate it!)

The following are equivalent:

```
a, b = b, a
```

```
temp = a  
a = b  
b = temp
```

Strings are immutable

- ▶ **Strings** are immutable in Python
 - ▶ This is not true of all programming languages
- ▶ But wait... we change strings all the time, don't we?

```
my_string = "Hello "  
my_string += "world"
```

- ▶ This isn't changing the string, it's creating a new one and throwing the old one away!
- ▶ Hence building a long string by appending can be slow (appending strings is $O(n)$)

Dictionaries

- ▶ Dictionaries are **associative maps**
- ▶ A dictionary maps **keys** to **values**
 - ▶ Keys must be immutable (numbers, strings, tuples etc)
 - ▶ Values can be anything (including dictionaries or other containers)
- ▶ A dictionary is implemented as a **hash table**

Using dictionaries

Create them using {}:

```
age = {"Alice": 23, "Bob": 36, "Charlie": 27}
```

Access values using []:

```
print(age["Alice"]) # prints 23  
age["Bob"] = 40      # overwriting an existing item  
age["Denise"] = 21   # adding a new item
```

Iterating over dictionaries

Iterating over a dictionary gives the **keys**:

```
for x in age:  
    print(x)    # prints Alice, Bob, Charlie
```

Use `items` to get **key,value** pairs:

```
for key, value in age.items():  
    print(key, "is", age, "years old")
```


Sets

- ▶ Sets are like dictionaries without the values
- ▶ Sets are **unordered** collections of **unique** elements
 - ▶ Sets **cannot** contain **duplicate** elements
 - ▶ Attempting to `add` an element already present in the set does nothing
- ▶ Certain operations on sets scale better on average than the equivalent operations on lists:

Operation	List	Set
Add element	Append: $O(1)$ Insert: $O(n)$	$O(1)$
Delete element	$O(n)$	$O(1)$
Contains element?	$O(n)$	$O(1)$

Using sets

Create them using {}:

```
numbers = {1, 4, 9, 16, 25}
```

Add and remove members with `add` and `remove` methods

```
numbers.add(36)  
numbers.remove(4)
```

Test membership with `in` operator

```
if 9 in numbers:  
    print("Set contains 9")
```

2-dimensional arrays

2-dimensional arrays

Common problem: we want to represent a **2-dimensional array** of values (i.e. a grid)

$$\begin{array}{cccc} V_{0,0} & V_{1,0} & \cdots & V_{w-1,0} \\ V_{0,1} & V_{1,1} & \cdots & V_{w-1,1} \\ \vdots & \vdots & \ddots & \vdots \\ V_{0,h-1} & V_{1,h-1} & \cdots & V_{w-1,h-1} \end{array}$$

Approach 1: flat list

- ▶ For a $w \times h$ array, create a list of size wh
- ▶ The element in column x row y is accessed by
`list[y * w + x]`
- ▶ E.g. $w = 5, h = 4$:

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

Approach 2: list of lists

- ▶ For a $w \times h$ array, create a list of size w , where each element is a list of size h
 - ▶ Each element of the “outer” list represents a column of the array
- ▶ The element in column x row y is accessed by `list[x][y]`, i.e. the y th element of the x th column

Approach 3: dictionary

- ▶ Represent the array as a dictionary whose keys are (x, y) tuples
- ▶ The element in column x row y is accessed by `list[x, y]`

Approach 4: NumPy array

- ▶ Requires NumPy or SciPy, and can only store numeric types
- ▶ However, highly optimised for intensive calculations (e.g. “tinkering” with image pixel colours...?)

Which is best?

- ▶ Flat list is reasonably efficient but not very readable
- ▶ List of lists is a reasonable trade-off between efficiency and readability
- ▶ Dictionary allows for “sparse” arrays (e.g. some cells can be missing)
- ▶ NumPy array is less versatile but faster in some use cases

There is no single “best” approach — it depends how you use it

Worksheet D

Due **next week**

Exercise Sheet iii