FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 6

Semester 2, 2018

Objectives of this tutorial

- To understand how to work with ADTs.
- To understand Classes and Objects.
- To understand namespaces and scoping.

Exercise 1 *

This exercise and the next one are about scoping and namespaces. Examine these nested functions:

```
def silly():
    x = 'ping'
    def g():
        print(x)
    x = 'pong'
    def f(x):
        print(x)
    g()
    f(x)

silly()
```

Solution

The call to `silly()` will print "'pong" twice. Here's why:

```
def silly():
        x = 'ping'      # here x is 'ping'
        def g():        # and here g is defined
                print(x)    # and it's true that this x means the enclosing scope's x
        x = 'pong'      # but here we change the enclosing scope's x to mean 'pong'
        def f(x):
                print(x)
        g()             # so by the time we call g(), the enclosing scope's x
        f(x)            # is the same than when we call f(x), that is, 'pong'

silly()             # therefore, this will print "pong" twice, on two separate lines
```

Exercise 2 *

More scoping and namespaces. Now, with objects! A student named Juan is asked to make a Member class for a community pool. The only thing we care about is our pool's name, and our members' name, age and gender. So Juan writes:

```
class PoolMember:
    poolname = "FIT1008 students community pool"
    name = ""
    age = 0
    gender = None

    def __init__(self, name, age, gender):
        PoolMember.name = name
        PoolMember.age = age
        PoolMember.gender = gender
```

Juan makes himself the first member of the pool, and also its admin. It all seems to go so well!

```
>>> admin = PoolMember('Juan', 18, 'male')
>>> admin.name
'Juan'
>>> admin.age
18
>>> admin.gender
'male'
>>> admin.poolname
'FIT1008 students community pool'
>>>
```

But an order comes from above. The pool has to be open to any Monash students and staff, not just FIT1008 students. And we need to change the name before other students come! So we tell Juan to change the pool name before we make the new membership card for a well-respected professor, Emilia, who will be our supervisor:

```
>>> admin.poolname = "MONASH all-inclusive community pool"
>>> admin.poolname
'MONASH all-inclusive community pool'                # it looks allright
>>> supervisor = PoolMember(Emilia, 26, 'female')    # let's make Emilia's user
```

Phew! Disaster averted! All seems alright again, until Emilia wants to check that the pool's name was really changed

```
>>> supervisor.poolname
'FIT1008 community pool'          # what, this can't be!
>>> PoolMember.poolname           # let's check the global name
'FIT1008 community pool'          # it's the same!
```

It's still the old name! Emilia wants to find who is responsible for the mistake, so she looks at the details for the admin:

```
>>> admin.name
'Emilia'
>>> admin.age
26
>> admin.gender
'female'
```

- What? Is Juan framing Emilia for his mistakes? Or is it just an unfortunate series of bugs?
- What did Juan do wrong? Can you write the correct class and the correct way to change the pool's name?

Solution

What did Juan do wrong? More like what did Juan do right, right?

First, when he defined his class thus:

```
class PoolMember:
        poolname = "FIT1008 students community pool"
        name = ""
        age = 0
        gender = None

        def __init__(self, name, age, gender):
                PoolMember.name = name
                PoolMember.age = age
                PoolMember.gender = gender
```

He made `name`, `age` and `gender` properties of the class. The initialiser modifies the class property instead of making a variable for each instance of the class.

Thus, all members of the pool will share the same name, age and gender. Not only that but the name, age and gender of all pool members will be the ones for the latest person we've added. If a grannie joins the pool, suddenly all users will be elderly females; if a new baby boy enrol, suddenly all users will become newborn males. That's why the admin user details are actually Emilia's.

The second mistake Juan made was to modify the pool's name in his own instance. In doing so, he created an instance variable `poolname` that was only accessible through the admin user, via the qualified access `admin.poolname`.

This would be the correct way of writing the class:

```
class PoolMember:
        poolname = "FIT1008 students community pool"

        def __init__(self, name, age, gender):
                self.name = name
                self.age = age
                self.gender = gender
```

And this is how you can change the name of the pool and check that everything is fine:

```
>>> admin = PoolMember('Juan', 18, 'male')
>>> PoolMember.poolname = "MONASH all-inclusive community pool"
>>> admin.poolname
        'MONASH all-inclusive community pool'
>>> supervisor = PoolMember('Emilia', 26, 'female')
>>> supervisor.poolname
        'MONASH all-inclusive community pool'
>>> admin.name
        'Juan'
```

Exercise 3 *

Consider the following function, which again uses the abstract `List` data type defined in week 5:

```
def mystery(a_list1, a_list2):
    temp = ""
    n1 = length(a_list1)
    n2 = length(a_list2)
    for i in range(n1):
        if i < n2:
            item2 = get_item(a_list2,i)
            temp +=  item2
            if get_item(a_list1,i) > item2:
                return temp
    return temp
```

(a) Again without executing the code, indicate what the code does and illustrate with a few different examples.

(b) What is the best and worst Big O complexity of this function for two lists of strings? Explain.

Solution

The function sums up in `temp` the elements `a_list2[i]` for which `a_list1[i]` exists, until it finds an `a_list1[i]` that is greater than `a_list2[i]`. It then returns `temp`.

The best case takes a single iteration, and happens when the first element in `a_list1` is greater than the first element in `a_list2`. This iteration has all constant steps except for the comparison, which depends on the elements (the strings) themselves. Comparison of strings will be often O(m), where m is the length of the string. If so, the best case is O(m). In general, whatever the BigO of the comparison is (say O_comp), then the BigO of our mystery function will be O_comp.

The worst case takes $n1$ iterations, where $n1$ is len(`a_list1`), and happens when every element in `a_list1` is smaller or equal than the elements in `a_list2`. In this case, the BigO is O($n1$)*O_comp. If we assume as above that O_comp is O(m), then we would have O($n1 * m$).

Interestingly, the code can be easily modified to return as soon as list `a_list2` finishes:

```
def mystery(a_list1, a_list2):
    temp = ""
    n1 = length(a_list1)
    n2 = length(a_list2)
    for i in range(min(n1,n2)):
        item2 = get_item(a_list2,i)
        temp += item2
        if get_item(a_list1,i) > item2:
            return temp
    return temp
```

In this case, the best case is the same, but the worst case might be better, since the number of iterations is the minimum of $n1$ and $n2$, where $n1$ is as before and $n2$ is the length of `a_list2`. And the code is much clearer!

Exercise 4 *

Given below is a snippet of a problem description:

A coffee store in a popular shopping centre is planning to introduce a customer loyalty system. This program is designed to offer rewards for their frequent customers, as well as make their ordering process easier. Each customer has an ID (assigned automatically by the system), name, phone number, and a count of loyalty points. The system will record points earned, accumulated by each customer when they order their coffee. These points can then be redeemed (used) for free coffee on subsequent visits. The system also manages up to 5 different coffees that the store makes. Each coffee has a name and price (in whole dollars). Each coffee is also half price on one day of the week. This day is different for each coffee, and is checked when an order is placed.

1. Describe the ADT's you would require for a program to implement this customer loyalty system.

Solution

There are different options. One option is to maintain a list of customers and a list of coffees.

1. CoffeeShop - maintains a list of customers and a list of coffees.
2. Customer - contains Customer attributes and methods. Attributes include name, phone number and number of points. The methods might include one that updates the number of points, each time a coffee is bought.
3. Coffee - contains Coffee attributes and methods. Attributes include type, price, and day of the week in which the coffee is discounted.
4. ListADT - as defined in the lectures: it includes attributes the_array and length, and methods _empty, is_full, add, delete, size.

The student could think about some issues with this model. For example, if we wish to search for a customer, what if the name of a customer is not unique? Should we ideally have a customer ID instead?

Later we will learn about dictionaries and hash tables, and the student may wish to think about an implementation with dictionaries. When searching for a customer 'record' by name, would using a dictionary speed things up? By how much?