list comprehension

In set theory, we learn a couple of ways of creating a set.
a) enumeration : All the elements are listed.
Example: s = { 1, 3, 5, 7, 9}
b) rule based or builder method:
s = { x | x is odd and 1 <= x <= 10 }

So far we have learnt how to create lists by enumeration. In this section, we shall learn how to create lists using rules. This method of list creation is called list comprehension.

We shall examples from the file 1_list_comprehension.py.

```
l1 = [ 'hello' for x in range(5)]
```

This statement would create a list containing hello five times.
The above statement is equivalent to the following piece of code.

```
l1 = []
for x in range(5):
        l1.append(x)
```

The general form of list comprehension is
[<expr> for <variable> in <iterable>]
Semantically, this is equivalent the following.
Create an empty list. Execute the for part of the list comprehension. Evaluate the expr each time. Append that to the list. The result is the list so created.

Observe the similarity with map.

Let us look at some more examples.

```
# squares of numbers from 1 to 5
l2 = [ x * x for x in range(5)]
print(l2)
```

Is the above code self explanatory?

```
# list of tuples having a number and its square
l3 = [ (x, x * x) for x in range(5)]
print(l3)

# list of strings and its length
l4 = [ (x, len(x)) for x in ['bangalore', 'mysore', 'hubballi', 'shivamogga']]
print(l4)
```

The one below is an example of nested loops in list comprehension.

```
# cartesian product
l5 = [ (x, y) for x in range(4) for y in range(4)]
print(l5)
```

The one below is an example of selection amongst the many produced by the loops. Observe the similarity with filter.
In fact it is a combination of map and filter.

```
# relation: partial order
l6 = [ (x, y) for x in range(4) for y in range(4) if x < y]
print(l6)

# convert all words to uppercase
# map
a = ['bangalore', 'mysore', 'hubballi', 'shivamogga' ]
b = [ x.upper() for x in a ]
print(b)

# filter
# find all words whose len exceeds 7
b = [ x for x in a if len(x) > 7]
print(b)
```

```
# convert all words to uppercase if len exceeds 7
# combine
b = [ x.upper() for x in a if len(x) > 7]
print(b)
```

list comprehension provides an alternate mechanism to functional programming constructs map and filter.
We also have set, dict comprehension. These are not part of the course – hence not discussed.

Exception: Exceptional story
Every morning you leave home (or the hostel) – come to the college directly – attend all classes – go home in the evening. Occasionally you get distracted – a movie is being shown – a shooting (of some sort) is happening on the way – a strong nice aroma is coming out of an eatery – so you change your path. That is not normal – therefore is an exception.

A program executes some piece of code normally. Sometimes it takes not a normal path. An exception does not always mean it is an error.
If we walk through some iterable using the for loop, we should come to an end of the iterable at some time. Then python signals it as stop iteration.
Sometimes exception could also be an error. The index specified to get an element of the list may be beyond the list boundary. Then python indicates this as index error.

When an exception, the program is terminated. Can we avoid this termination? Can we get a chance to proceed further? Can we have graceful degradation? This concept is called exception handling.

A few components and terminologies used with exception handling:
a) try:
We indicate to the Python runtime that something unusual could happen in the suite of code.

b) except [<type>]:

We should have at least one except block to which the control shall be transferred if and only if something unusual happens in the try suite.

Let us say you walk home or the hostel every day from the college. That is normal. Some day, you get drenched in rain. Some other day, you trip and your footware breaks down. Do you treat theses unusual cases the same way?

Similarly, within the try suite, there could be any one of the number of exceptional cases happening – each of which might require different ways of handling.

So try block may be followed by one or more except block – all but one of them specifying the name of the exception – one of them may provide a common and default way of handling all exceptions.

You get dirty – fall into a ditch, a vehicle favours with a mud shower, a pig runs into you, you want to become light – is not the solution same?

c) builtin exceptions:

There are a number of expections which Python knows. We call them builtin exceptions.

Examples:

index error

key error

name error

These are automatically raised when they happen.

d) raising exception:

We indicate something is unusual or python runtime itself will indicate something exceptional using raise statement.

raise Exception_name(<message>)

This causes creation of an exception object with the message. The object also remembers the place in the code (line number, function name, how this function got called on). The raising of exception causes the program to abort if not in a try block or transfer the control to the end of try block to match the except blocks.

e) matching of except blocks:

The raised or thrown exception object is matched with the except blocks in the order in which they occur in the try-except statement. The code following the first match is executed. It is always the first match and not the best match. If no match occurs and there is an default except block (with no exception specified), then that block will be executed.

f) finally block:
This is optional. This follows all the except blocks. It is part of the try statement. This block shall be executed on both normal flow and exceptional flow.

Let us understand the flow of execution.
i) normal flow
try block – finally block if any – code following try block
ii) exceptional flow
try block – exit the try block on an exception – find the first matching except block – execute the matched except block – finally block – code following try block
Observe there is no mechanism in any language including Python to go back to the try block – no way to resume at the point of exception.

g) user defined exception:
It is not possible for a language to specify all possible unusual cases. So the users can also specify exception as a class which inherits from a class called exception.

h) exception propagation:
We say that a try block has dynamic scope. Any exception raised in the block or any function called from there are considered part of the try block. Any exception raised within these functions called from there will cause the control to be propagated backwards to the except blocks of this try block.

Let us go through a few examples.

The following code is taken from the file **2_exception_intro.py.**

You will observe that the program aborts as soon as any one of these statements is executed. You get know a few builtin exceptions in this program.

```
# example 1
# print("res : ", 10 / 0)# ZeroDivisionError: division by zero
# example 2
# print(myvar)# NameError: name 'myvar' is not defined
# example 3
# open("unknown.txt") #FileNotFoundError: [Errno 2] No such file or directory: 'unknown.txt'
```

Let us observe some code from the file 2_exception.py

```
m = 10
#n = 2
n = 0
try:
        print("one")
        print("res : ", m / n)
        print("two")
except Exception as e:
        print(e)
```

print("two") : This statement is executed only if the earliest statement does not raise an exception.
On an exception, object e of the class DivisionByException is created.
This class is the derived class of class Exception. The variable is of type Exception. An object of base class can always receive an object of the derived class. print(e) calls e.__str__() and displays the resulting string.

The code below shows how to have multiple except blocks and how the ordering matters.

Try putting default except block as the first and see what happens.

```
m = 10
#n = 2
n = 0
try:
        print("one")
        print("res : ", m / n)
        print("two")
        print(myvar)
        print("three")
except NameError as e:
        print("no such name : ", e)
#except Exception as e:
#       print("all other exceptions : ", e)

except:
        print("all exceptions")
```

Also observe that on an exception :
- first match is executed
- one and only one except block is executed
- there is no way to go back to the try block
no resume at the point of exception.

The following code is from the file 4_exception.py.
It shows how to make our own exception object.

The code is self explanatory. Please explore by experimenting.

```
# user defined exception
class MyException(Exception):
        def __init__(self, str):
                self.str = str
        def __str__(self):
                return self.str

# check whether n is between 1 and 100
n = int(input("enter a number:"))
try:
        if not 1 <= n <= 100 :
                raise MyException("number not in range")
        print("number is fine : ", n)
except MyException as e:
        print(e)
print("thats all")
```

Thats about exceptions.