CE151

Introduction to Programming
Part 5
Scope and Lifetime, Tuples, Error-Handling and Files

Variable Scope 1

The ***scope*** of a variable denotes the part of the program in which it is accessible.

If a variable is given a value outside of any function it is said to have ***global*** scope and is accessible anywhere in the program.

Consider the following short program.

```
def printx() :
  print("x is", x)

x = 17
printx()
```

The call to `printx` will generate the output `x is 17` as the variable is accessible inside the function.

Variable Scope 2

A variable which is given a value inside the body of any function is said to have ***local*** scope and is accessible only inside that function.

Consider the following short program.

```
def setxTo9() :
  x = 9

setxTo9()
print(x)
```

If this program is run the following error message will be output:

```
NameError: name 'x' is not defined
```

Variable Scope 3

We saw on slide 2 that it is possible to access a global variable within a function. If, however, the function contains a local variable with the same name, the global variable is hidden by the local variable and cannot be accessed within the function.

Consider the following short program.

```
def printxAndSetxTo9() :
  print(x)
  x = 9

x = 35
printxAndSetxTo9()
```

Variable Scope 4

When a function is called the interpreter generates a ***namespace*** that contains the names of all of the local variables in that function. In the program on the previous slide the variable `x` will be in the namespace for the function `printxAndSetxTo9`, so when the call to `print(x)` is made inside this function, the presence of `x` in this namespace means that the global variable is hidden, and the local variable has not yet been given a value. Consequently an error message will be generated. In this case the message will be

```
UnboundLocalError: local variable 'x' referenced
before assignment
```

Variable Lifetime 1

The ***lifetime*** of a variable indicates when the variable exists during the execution of a program.

The lifetime of a global variable extends from the time it is first given a value until the end of the execution.

The lifetime of a local variable extends only until the end of the current call to the function in which it is defined. Hence if a local variable is given a value during one call to a function that value will not persist until the next call is made to the function.

Variable Lifetime 2

Consider the following function.

```
def myFunc(n) :
   if n>0 : x = n
   print(x)
```

If we call this function with 0 as an argument we would expect an error to occur since the variable `x` is not given a value.

If we call it with a positive argument `x` will be given a value and this value will be printed; if after doing this we call the function again with 0 as an argument the value previously stored in `x` will not have persisted, so once again an error will occur.

Function Arguments 1

A function argument can be viewed as a local variable that is given a value immediately before the code in the function is run.

Consider the following program.

```
def increment(n) :
    n = n + 1
x = 5
increment(x)
print(x)
```

The argument `n` is made to refer to the integer to which `x` refers. Inside the function `n` is made to refer to a different integer and no change is made to the integer to which `x` refers so the output will be 5, not 6; the function does nothing useful.

Function Arguments 2

When a mutable sequence is supplied as an argument to a function the behaviour appears to be somewhat different.

Consider the following program.

```
def increment(l) :
    i = 0
    while i < len(l) :
        l[i] = l[i] + 1
        i = i + 1
myList = [1, 2, 3]
increment(myList)
print(myList)
```

The argument `l` is made to refer to the list to which `myList` refers. Inside the function we change the contents of that list so the call to `print` will output `[2, 3, 4]` .

Function Arguments 3

Although it might appear that list arguments are being handled in a different way to integer arguments this is not actually the case.

The reason our two examples behaved differently is simply that an assignment of the form `n = .....` makes the variable `n` refer to a new object, whereas an assignment of the form `l[i] = .....` changes the contents of the list to which the variable `l` refers.

Tuples 1

Suppose we wish to write a function that finds the smallest and largest values in a list of numbers. If within the function we assign these values to variables called `smallest` and `largest` these variables will no longer exist after the call to the function so the caller will not be able to retrieve the results.

The only way the function can supply numeric values to its caller is by returning the values, but a function can return only one value. Hence we need some way for two integers to be packaged into a single result.

It would be possible to return a list containing two items, but a better approach is to use a different kind of sequence, a ***tuple***.

Tuples 2

A tuple is like a list, but it has a fixed number of elements and is immutable (i.e. its contents cannot be changed).

Since a tuple normally has a small number of elements we do not expect to iterate through the elements one at a time using a loop, and the programmer should know what each element represents. Consequently it is quite reasonable to use tuples whose elements have different types. We could use a tuple to represent the name and age of a person, e.g.

```
monty = ("Monty Python", 40)
```

As seen in this example we create tuples in the same way as lists but use `(` and `)` instead of `[` and `]`.

Tuples 3

We now present the function to return the smallest and largest values in a list.

```
def minmax(l) :
    """
    find smallest and largest values in l
    argument: l: list of int
    returns: (smallest, largest): tuple of 2 ints
    """
    if len(l) == 0 : return None
    smallest = largest = l[0]
    for n in l : # could use n in l[1:]
        if n < smallest : smallest = n
        elif n > largest : largest = n
    return (smallest, largest)
```

Tuples 4

In the function on the previous slide the value `None` was returned if the list was empty – this is probably not the best approach, but the best we can yet write using features of Python that we have so far encountered.

After making a call of the form

```
myTuple = minmax(myList)
```

we need some way of extracting the two results from the tuple. Since a tuple is a sequence we could do this using subscripting (e.g. `myMin = myTuple[0]` and `myMax = myTuple[1]`) but a better approach is to use a multiple assignment:

```
myMin, myMax = myTuple
```

Tuples 5

When using multiple assignment to extract the values of a tuple the number of variables on the left-hand side of the assignment must match precisely the size of the tuple.

Assuming `myTup` refers to the tuple `(1, 2, 3)` the assignment

```
a, b, c = myTup
```

would correctly extract the values but

```
a, b = myTup
```

and

```
a, b, c, d = myTup
```

would both result in errors when run.

Tuples 6

On rare occasions it might be necessary to create a tuple containing just one element.

An assignment such as `newTup = (1)` will not create a tuple as the expression `(1)` does not represent a tuple. (If it did, the expression `(3+4)` would also have to denote a tuple, so `(3+4)*3` would have the value `(7, 7, 7)` as the `*` symbol would denote concatenation instead of multiplication.)

Hence to create a tuple with just one element we have to use

```
newTup = (1,)
```

Exceptions 1

We have encountered several situations where errors can occur when a program is running. Some of these, such as attempts to access an undefined identifier or add a string to a number, are the result of programmer errors (e.g. forgetting to use `int(input(…))` when getting a number from the keyboard). Others result from users supplying incorrect data (e.g. typing a letter when a number is required). Errors can also occur due to problems with the environment in which the program is running (e.g. an attempt to access a file on a network drive that is unavailable).

When an error occurs at runtime an ***exception*** is ***raised***. If this is not ***caught*** an error message with details of the exception is displayed and the program is aborted.

Exceptions 2

When a user supplies invalid input data we should expect a program to display an appropriate message and ask the user to try again. If a file access problem occurs we should expect the program to generate a user-friendly error message before terminating instead of the error message from the interpreter. To do this we need to be able to catch any exceptions that are raised during input and file-access.

`try` and `except` Blocks 1

When writing code that may cause an exception to be raised we should place that code in a ***try block***. This is a suite of statements that follows a line `try:` , e.g.

```
try :
    age = int(input("Enter your age: "))
```

This should be followed by one or more ***except blocks***, each of which catches and handles a different type of exception. If an exception is raised during execution of the `try` block, control will be passed immediately to the appropriate `except` block. If there is no block for the exception the program will terminate with an error message (unless the code containing the `try` block is in a function that has been called from within another `try` block, in which case its `except` blocks will be used.)

`try` and `except` Blocks 2

The exception that will be raised when a string that does not comprise digits is passed as an argument to the `int` function is an object of type `ValueError`. Hence to catch this exception we need an `except` block such as

```
except ValueError as e :
  print("Your age should be an integer.")
```

Alternatively we could write a more generic `except` block that will handle all exceptions:

```
except Exception as e :
  print("Invalid input")
```

The identifier used after the keyword `as` is the name of a variable in which a reference to the exception object will be stored – we can use this if necessary to obtain details of the error.

`try` and `except` Blocks – an Example 1

On the next slide we present an example of the use of `try` and `except` blocks to obtain a number from the user and store it in a variable called `age`. If the user does not supply valid input he or she will be asked to try again – this will be repeated until a valid number has been obtained.

We use a boolean variable called `gotOne` to keep track of whether a number has been successfully input.

`try` and `except` Blocks – an Example 2

```
gotOne = False
while not gotOne :
  try :
    age = int(input("Enter your age"))
    gotOne = True
  except ValueError as e :
    print("Invalid input - try again")
```

If there are several places in a program where integer input is needed it would be better to write a function to obtain the input. The function on the next slide takes the prompt as an argument and returns the input value. It does not repeat the original prompt when the user has to try again but instead uses a try-again message as a prompt.

try and except Blocks – an Example 3

```
def getInt(prompt) :
  """
  gets integer from keyboard
  argument: prompt message: str
  returns: input value: int
  """
  gotOne = False
  while not gotOne :
    try :
      num = int(input(prompt + ": "))
      gotOne = True
    except ValueError as e :
      prompt = "Invalid input - try again"
  return num
# sample call: age = getInt("Enter your age")
```

Using Files 1

To read from or write to a file we first need to open the file:

```
f = open("myfile.txt")
```

The `open` function attempts to open a file and if successful returns a reference to an object of type `File` that can subsequently be used to access the file.

When the function is called with a single argument the file is opened for reading; if the file does not exist or cannot be opened due to insufficient access permissions an exception of type `IOError` will be raised.

Having opened the file there are several ways of reading from it. For a text file the simplest approach is to use the `readline` method to read the contents of the file one line at a time.

Using Files 2

If **f** refers to a file that has been opened for reading a call of the form

```
myLine =  f.readline()
```

will read the next line from the file and store it as a string. The newline character at the end of the line will be included in the string so the string will be non-empty even if the line is blank. The only time that a string of length 0 will be returned is when the end of the file has been reached, so to input an entire file a line at a time we should use a loop that repeatedly calls **readline** until a string of length 0 is returned.

The code on the following slide will read the entire contents of a file and display it on the screen.

Using Files 3

```
fname = input("Enter filename: ")
try :
  f = open(fname)
  myLine = f.readline()
  while (len(myLine)>0) :
    # print generates a newline so we do not want
    #       the newline from the string
    print(myLine[:-1])
    myLine = f.readline()
  f.close()
except IOError as e :
  print("Problem opening file")
```

Using Files 4

After we have finished processing a file it should be closed using the `close` method, as seen on the previous slide.

To open a file for writing we need to use two arguments to the call to `open`; the second argument should be `'w'` if we wish to overwrite the existing contents of the file or `'a'` if we wish to append our output to the end of the file, e.g.

```
f2 = open("M:ce151\myfile.txt", 'w')
f3 = open("file2.txt", 'a')
```

In both cases if the file does not exist it will be created (assuming that the folder in which we are trying to create it exists and we have appropriate access permissions to write to that folder).

Using Files 5

To write a string to a text file we can use the **`write`** method, e.g.

```
f2.write("Hello")
f3.write(myString)
```

It can often be more convenient to use the **`print`** function with which we are already familiar, since this can take arguments other than strings. To do this we need to supply, after the arguments containing the values that we want to print, an extra argument of the form **`file = f`**. (This must be the last argument and the identifier **`file`** must be used; **`f`** denotes the variable that refers to the **`File`** object.)

For example

```
print("Age:", myAge, file = f2)
for i in myList : print(i, file = f3)
```

Using Files 6

We may wish to write a program that will normally write its output to a file, but will output to the screen if the file cannot be opened. We would not want to have to write two copies of the code, one with `file = .....` arguments in the calls to `print` and one without. If the file could not be opened we should instead make the file variable refer to the ***standard output*** stream (which is usually the window in which the program is running).

This stream is accessible via a variable called `stdout` defined in the `sys` module, so we need to import this and make the file variable refer to it. An example is seen on the next slide.

(There are also variables `stdin` and `stderr` which refer to the standard input and standard error streams.)

Using Files 7

```
from sys import stdout
fname = input("Enter filename: ")
try :
  f = open(fname, 'w')
except IOError as e :
  print("Problem opening file")
  print("Using standard output instead")
  f = stdout
for i in myList :
  print(i, file = f)
if f!=stdout :
  f.close()
```