Advanced Programming
Lecture 5, 19 January 2012

Errors and exceptions

We have a dictionary d and a key k for which we want to do the following:

```
if d[k] exists
   increment d[k]
else
   initialize d[k] = 1
```

If k is not a key of d, trying to check for d[k] raises an error. One way to get around this is to write

```
if k in list(d.keys()):
   d[k] = d[k] + 1
else:
   d[k] = 1
```

However, this constructs a list of all keys in d and, in general, takes time linearly proportional to the size of d.keys(). On the other hand, dictionaries are optimized to quickly look up keys, so it would be more efficient to directly look up d[k] and take corrective action if this fails.

The type of code we want to write is

```
try to increment d[k]
if this fails, set d[k] = 1
```

This is typical of the following pattern

```
In the normal course of things, do action A
If something unexpected happens, do B instead
```

We need a way to "trap" or "catch" messages about unexpected, or exceptional, situations within the code and take corrective action. This is done using the try ... except ... mechanism of Python.

Here is how we could write the example above.

```
try:
  d[k] = d[k] + 1
except:
  d[k] = 1
```

The statements between "try:" and "except:" are the default control flow. If any statement here generates an error, control switches to the "except:" block. If the "try:" block terminates naturally, control skips the "except:" block.

We can do more than one thing in a "try" block. For instance:

```
try:
  d[k] = d[k] + 1
  l[d[k]] = l[d[k]] + 1
except:
```

```
    d[k] = 1
```

What if we get an error because d[k] is not a valid index within the list l? We can have multiple except clauses, naming the error that triggers each one. To find the name to use, look up the Python documentation, or generate the error manually in the Python shell. For instance, a key lookup error for a dictionary yields a KeyError while an index lookup error for a list yields and IndexError, as indicated by the following interaction with the Python interpreter.

```
>>> d = {}
d = {}
>>> d[1]
d[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1
>>> l = []
l = []
>>> l[1]
l[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

Here is how to write code that checks for different types of errors.

```
try:
  d[k] = d[k] + 1
  l[d[k]] = l[d[k]] + 1
except KeyError:
  d[k] = 1
except IndexError:
  pass
```

The statement "pass" does nothing and is useful for situations where Python requires a statement but nothing useful needs to be done. In this code, IndexErrors are "caught" but ignored. If we left out the second except, a KeyError would be handled internally while an IndexError would abort the code.

Note that the first error encountered aborts the try block and moves to the except clauses, so there can never be a situation where two errors are "competing" for attention. If we write an expression like

```
x = f(d[k]) + g(l[n])
```

it is possible that d[k] generates a KeyError and l[n] generates an IndexError. The order in which the expression on the right is evaluated will determine which one is generated first --- this order may not be predictable in general, but the statement will abort the moment the first error is generated.

As we saw before, an except clause without a named error catches all errors. So, we could also write the following to ignore all errors other than KeyErrors.

```
try:
  d[k] = d[k] + 1
```

```
    l[d[k]] = l[d[k]] + 1
  except KeyError:
    d[k] = 1
  except:
    pass
```

An exception that is not caught is passed up to the calling function. Thus, if our main program calls f() and f calls g() and g generates an error that is not caught in a try block, this error is passed to f. Thus, in f one could write

```
  try:
    g()
  except:
    .....
```

to take care of errors generated by g. If f, in turn, does not catch this error, the error goes back to the main program. An exception that escapes the main program results in the code being aborted and an error being displayed to the user.

We can assign an error to a name and use this name. For example:

```
  try:
    d[k] = d[k] + 1
    l[d[k]] = l[d[k]] + 1
  except KeyError:
    d[k] = 1
  except (IndexError,ValueError) as err:
    print("Index or Value error", err)
  except Exception as othererr:
    print("Other error", othererr)
```

Note that the second except clause catches multiple types of exceptions. "Exception" is the most generic type of error and can be used if we want to assign a name in the default case.

Finally, we can also explicitly raise our own exceptions using the statement raise.

```
  def factorial(n):
    if n < 0:
      raise ValueError(n)
    return(n*factorial(n-1))
```

We can call raise without an argument in an except clause to pass on the current error.

```
  try:
    d[k] = d[k] + 1
    l[d[k]] = l[d[k]] + 1
  except KeyError:
    d[k] = 1
  except:
    raise
```

There are two other clauses that go with "try""; "else:" and "finally:". As for loops, "else:" is excecuted if the try completes normally, without flagging an error. "finally:" is always executed after the "try:" and is used for 'cleanup' actions.

```
 def divide(x, y):
   try:
     result = x / y
   except ZeroDivisionError:
     print("division by zero!")
   else:
     print("result is", result)
   finally:
     print("executing finally clause")
```

Basic Input and Output

If we execute Python code directly, rather than interactively in the Python shell, we need a mechanism to pass data to the program and to export information out of the program. The easiest way to pass data in and out of a program is to type input using the keyboard and display output on the screen.

The function print() displays data on the screen. It takes any number of arguments, converts each to a string by implicitly calling the str() function and displays the result on a line.

```
  x = 7
  print("The value of x is",x)
```

displays

```
  'The value of x is 7'
```

Note that Python inserts a space automatically between arguments. Each print() generates a new line. To continue the next print where the current print left off, specify the string to print instead of the newline as the last parameter. For instance,

```
  x = 7
  print("The value of x is",x, end = "--")
  y = 8
  print("the value of y is",y)
```

displays

```
  'The value of x is 7--the value of y is 8'
```

You can format your output more precisely to align columns of successive prints etc, but we will not go into these details. You can look up the details in the Python documentation.

The function to read input from the screen is input(). You should assign the value read to a name.

```
  x = input()
```

When this statement is reached, the program will pause and wait for a line of input to be typed, which will be assigned to x as a string. To notify the user that input is expected, you can print a message when input() executes.

```
  x = input("Enter a number")
```

The value passed is always a string. If we type "378" at this point, x is the string "378", not the number 378. To convert the string to an integer, we need to explicitly call the int() function.

```
x = input("Enter a number")
x = int(x)
```

or, more concisely,

```
x = int(input("Enter a number"))
```

More precisely, the input string passed to x includes the newline character, written "\n", so the value of x after the input statement is actually "378\n". If the user types a few blanks before 378, x would be something like " 378\n". To remove white space (blanks, tabs, new lines) at both ends of a string, invoke the function strip() as follows.

```
x ="   Some leading blanks and trailing junk   \n"
y = x.strip()
```

The value of y is now "Some leading blanks and trailing junk". Only the leading and trailing white space is stripped --- all internal spaces are preserved. Note that x does not change as a result of this.

Use x.lstrip() and x.rstrip() to strip white space only at the left and right end, respectively.

Quite often, we might want to input multiple items on a line. If these are separated by whitespace, we can get the individual components using split.

```
x = input("Enter two integers separated by space")
```

Suppose we type " 333 467" and press Enter. This sets x to the " 333 467\n". We can now write

```
y = x.split()
```

This leaves x unchanged, but y becomes the list ["333","467"]. Notice that all white space is eliminated, both between the values and at either end, so split() does an implicit strip(). Remember that y is still a list of strings, so we need to use int() to actually extract the corresponding values as integers. We can supplyn an alternate delimiter to split using an argument.

```
x = "123:456"
y = x.split(":")
```

Now, y is ["123","456"].

Note that we can use "<" and ">" at the shell prompt to redirect input/output
to/from a file. For example

```
$ python3.2 myprogram.py < inputfile > outputfile
```

would take all input from "inputfile" instead of the keyboard and send

all output to "outputfile" instead of to the screen.

Reading and writing files

By using files, we can combine input from multiple sources and write output to multiple destinations. Files are stored on disk. Physically, reads and writes to disk happen in units called blocks. Since disk access is much slower than memory access, reading and writing a block at a time amortizes the delay due to input and output operaions involving a disk.

Thus, when we read a line from a file, the operating system actually fetches a block of data into memory, so that subsequent reads occur from the stored block in memory rather than from the file. Similarly, when we write to a file, the output is accumulated in memory until there is sufficient data to write a block of text out to the disk.

The space in memory that is used to store the temporary data read from or to be written to the disk is typically called a buffer. Using files in Python (or any other programming language) is a three step process.

1. Opening the file

   This associates a buffer with a file on the disk. At the time of opening we also specify the "mode" in which we will use the file --- we can either read from a file or write to it, but not both. Python has three modes, read ("r"), write ("w") and append ("a"). If we write to a file, we overwrite what it already contains. If we append, the new data is added at the end.

2. Actual read/write happens with respect to the buffer

3. Closing the file

   This severs the connection between the buffer and the file. If the file was opened in write/append mode, any pending data is written out to the file (this is called "flushing the buffer").

Here is the Python syntax for opening and closing files:

```
fb = open("myfile.txt","w")

fb.close()
```

The first line associates a buffer called fb with the file "myfile.txt". The first argument to open is a string that gives the path to the file. We can use "/" etc to specify files in different directories from the current one. The second argument to open is the mode, here "w" for "write". The other modes are "r" for "read" and "a" for "append".

The second line closes the file associated with the buffer fb.

If the file cannot be opened (e.g., the file does not exist, or you don't have the correct permission) an exception is raised. For example:

```
>>> fb = open("nonsense","r")
```

```
  fb = open("nonsense","r")
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  IOError: [Errno 2] No such file or directory: 'nonsense'
```

Thus, one should typically use open() within a try block

```
  try:
    fb = open("nonsense","r")
  except IOError:
    ... take approrpriate corrective action
```

We read a line from a file by invoking readline on the corresponding buffer.

```
   fb = open("inputfile","r")
   nextline = fb.readline()
```

Like the input() function, readline() returns a string upto and including the newline character "\n". The end of file is signalled by reading an empty string --- that is, readline() returns "". This is unusual in that normally end of file is signalled by an exception.

There is another function readlines() that reads an entire file and returns a list of strings, one per line.

```
   listoflines = fb.readlines()
```

Each item in the list will have a "\n" at the end. If we have already issued some readline() commands prior to readlines(), we will get all lines from the current point in the file to the end of the file.

There are other functions to read from a buffer such as read() which allows us to specify how many bytes to read. Look up the documentation.

To write to a file, we use the function write().

```
   message = "hello"
   fb.write(message)
   fb.write("\n")
```

Note that write() does not automatically insert the newline character: we have to explicitly write "\n". Use writelines() to write a list of strings. We have to still insert "\n" ourselves, so the function name is misleading: it does not convert each string in the list to a line.

```
   messagelist = ["hello\n","world\n"]
   fb.writelines(messagelist)
```

This writes two lines to the file associated with fb:

```
   hello
   world
```

```
   messagelist = ["hello ","world\n"]
   fb.writelines(messagelist)
```

This write a single line to the file associated with fb:

```
   hello world
```

Thus, writelines() is more accurately described as a sequence of write() operations.

Remember that the effect of a write() may not be immediately visible on the disk. Also, remember to close() the buffer before the program ends to ensure that all pending data is flushed to disk.

==========