File Operations, Pickle & Dictionary

Computer Science & Engineering Department

The Chinese University of Hong Kong

Content

- input and eval
- File operations
- Pickle
- Dictionary

Input

- We use input() to read input from user
  input([prompt])   [ ]: optional
-
  ```
  >>>  num = input("Enter a number: ")
  Enter a number: 36
  >>> num
  '36'
  ```

- All input are in the form of strings
- We can use string methods to parse the content eg. split()

Input

- We can then use *int*() or *float*() to convert into preferred form

```
>>>int('36')
36
>>>float('36')
36.0
```

What will be the result?

- int('12   int')
- Error!
- int('12.34')
- Error
- int('12   ')
- 12
- int('1010', 2)
- 10   int(x, base=10)

Input

- We can use eval() to convert also

```
>>>eval('36')
36
>>>eval('2+3*6')
20
```

- Provided the string contains valid expression

File

- Persistent storage even after program ends
- Represented in Python as type *file* (object)
- Typical file processing involves:
  1. File open
  2. Read/write operations
  3. File close

File

- In previous intro prog course, we learnt that file is usually opened with access *mode*

  ```
  file_handle = open( filename, mode)
  ```

- Typical mode:

1. "r" read
2. "w" write, create new file if not exists
3. "a" append to current file
4. "r+" read write access
5. "rb" to guarantee binary file can be read

File methods

f = open("filename")	open a file, return file value
f = open("filename", encoding)	open a file with encoding
f.read()	return a single character value
f.read(n)	return no more than n character values
f.readline()	return the next line of input
f.readlines()	return all the file content as a list
f.write(s)	write string s to file
f.writelines(lst)	write list lst to file
f.close()	close file

File

- **`readline`** return the next line of text, including a newline (return) character at the end
- Returns empty string when file empty

- Using for statement can also have the same result

```
>>> f = open('message.txt')
>>> for line in f:
...  print (line)
```

Recovering from Exceptions

- File I/O operations can generate exceptions, an IOError
- Exception handling can prevent these errors

```
try:
    f = open('input.txt')
except IOError as e:
   print ('unable to open the file with error: "{}"'.format(e.args[-1]))
else:
    print ('continue with processing')
    f.close()
print ('continue')
>>> unable to open the file with error: "No such file or directory"
continue
```

Handling files

- A better way is using *with*
- Ensure file is closed when the block inside with is exited
- But exception handling is still required as needed

```
with open('hello.txt', 'w', encoding='utf-8') as outf:
    # perform file operation
    outf.write("Hello world")

# Check if the file has been automatically closed.
print(outf.closed)  # prints True

print ('continue with processing')
```

Operating System Command

- Useful OS command can be executed from python by including os module

```
>>> import os
>>> os.remove("gone.txt")   # removing file

>>> os.curdir    # get current directory
'.'
>>> os.rename('oldfile.txt', 'newfile.txt')
>>>
```

Standard I/O

- **`print`** writes characters to a file normally attached to display window
- Input functions read from a file attached to keyboard
- These files can be accessed through **`sys`** module
- Input file : **`sys.stdin`**, output file: **`sys.stdout`**, error messages: **`sys.stderr`**
- **`stderr`** normally goes also to **`stdout`**

Standard I/O

- Can change these settings through sys

```
import sys
sys.stdout = open('output.txt', 'w')
sys.stderr = open('error.txt', 'w')
print ("see where this goes")
print (5/4)
print (7.0/0)
sys.stdout.close()
sys.stderr.close()
```

In output.txt

```
see where this goes
1.25
```

In error.txt

```
Traceback (most recent call last):
  File "try1.py", line 6, in <module>
    print (7.0/0)
ZeroDivisionError: float division by zero
```

OS functions

- **`exit`** terminate a running Python program – **`sys.exit("message")`**
- **`sys.argv`** is a list of command line options being passed

```
import sys
print ('argument of program are ', sys.argv)
>>>argument of program are
  ['D:\\ypchui\\COURSE\\Python\\lab\\lab3\\lab3.py']
```

If run directly in same folder

```
argument of program are  ['lab3.py']
```

Pickle

- Useful in saving and restoring Python variables as an archive
- Also called *serialization*

```
import pickle       # import to use
listOne = list()  # we can use list() to create a list
listTwo = list()
listOne.append( 12 )
listTwo.append( 'abc' )
listOne.append( 23 )
listOne.pop()
f = open('pickle1.pyp','wb')  # store the variables in a file
pickle.dump([listOne, listTwo], f)
```

Pickle

- Later on in another program

```
>>> import pickle
>>> f = open('pickle1.pyp', "rb")
>>> [listOne, listTwo] = pickle.load(f)
>>> print (listOne.pop())
12
>>> print (listTwo.pop())
abc
```

Dictionaries

- Indexed data structure - uses also square bracket notation
- Any *immutable* type can be used as index
- Braces create dictionary

```
>>> dct = { }          # create new dictionary
>>> dct['name'] = "Donald Duck"
>>> dct['age'] = 90
>>> dct['eyes'] = "black"
```

- Index is called a *key* (LHS)
- Element stored that associated with key is called a *Value* (RHS)

Dictionaries

- Also called maps, hashes or associative arrays

```
>>> print (dct['name'])
Donald Duck
>>> print (dct.get('age'))
90
>>> print (dct['weight'])
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
KeyError: 'weight'
```

Dictionaries

- Exception when no value with designated key
- Can be prevented by using built-in get method to check

```
>>> print (dct.get('weight', 0)) # 0 is default value
0
>>> dct['age'] = 18
>>> dct['age']
18
```

Dictionaries

- Del used to delete an element from list

```
>>> del dct['age']
>>> print (dct['age'])
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
KeyError: 'age'
>>>
```

- Can be initialized using colon ':' separated tuple also

```
>>> info = {'name':'Batman', 'age':82, 'weight':180}
>>> print (info['name'])
Batman
```

Dictionaries

- Dictionary values have no restriction, can be any *object*
- But two important properties

1. No duplicate keys allowed

```
>>> dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}
>>> print ("dict['Name']: ", dict['Name'])
dict['Name']:  Manni
>>> dict
{'Name': 'Manni', 'Age': 7}
```

Dictionaries

2. Keys must be immutable i.e. lists not allowed

```
>>> dict = {['Name']: 'Zara', 'Age': 7}
Traceback (most recent call last):
 dict = {['Name']: 'Zara', 'Age': 7};
TypeError: unhashable type: 'list'
>>>
```

Dictionary operations

Operation	Description
len(d)	number of elements in d
d[k]	item in d with key k, if k is not found in d, raises a KeyError.
d[k]=v	set item in d with key k to v
d.clear()	remove all items from dictionary d
d.copy()	make a shallow copy of d
k in d	return True if d has key k, False otherwise
d.items()	return a list of (key,value) pair
d.keys()	return a list of keys in d
d.values()	return a list of values in d
d.get(k)	same as d[k] except if k is not found in d, returns None
d.get(k,v)	return d[k] if k is valid, otherwise return v

Same as has_key(k) in Python 2.7 or earlier,

Return list in Python 2.7 or earlier, but return view in 3.X or later

Views Instead Of Lists

- Dict methods return "views" instead of lists in 3.X
- views like a window on the keys and values (or items) of a dictionary

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> keys  # No eggs anymore!
dict_keys(['sausage', 'bacon', 'spam'])
```

List Comprehension & Dictionary

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
>>> a = [1, 2, 3]
>>> [x*x for x in a]
[1, 4, 9]
```

- Operations on dictionaries performed by selecting values from range of keys, then returning items with selected keys

```
d = {1:'fred', 7:'sam', 8:'alice', 22:'helen'}
>>>[d[i] for i in d.keys() if i%2==0]
['alice', 'helen']
```

Appendix

Dictionary copy() Method Vs = Operator

```
d = {'name':'Batman', 'vehicle':['Batboat','Batcopter']}
b = d # copy of reference to d
c = d.copy()  # new object but shadow copy of d
print("c:",c)
d['name']='Robin'
d['vehicle'][0]='Batcycle'
print("c:",c) # c is partically updated
d.clear()
print("d:",d)
print("b:",b) # b is gone
print("c:",c) # d’elements referenced
```

```
c: {'name': 'Batman', 'vehicle':
['Batboat', 'Batcopter’]}

c: {'name': 'Batman', 'vehicle':
['Batcycle', 'Batcopter’]}

d: {}
b: {}
c: {'name': 'Batman', 'vehicle':
['Batcycle', 'Batcopter']}
```