Introduction to Python Programming

11 – File I/O, Exceptions, Encodings

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23

Running Python

- a) from Idle (or some other IDE)
- b) from the command line
- see http://www.coli.uni-saarland.de/courses/python

Reading input from keyboard

```
1  def main():
2      result = 0
3      while True:
4          number = input('Please enter a number: ')
5          if number == '':
6              break
7          result += int(number)
8      print('The result is:', result)
9
10 if __name__ == '__main__':
11     main()
```

Reading input from command line

```
1   import sys
2
3   def main():
4       result = 0
5       for arg in sys.argv[1:]:
6           result += int(arg)
7       print(result)
8
9   if __name__ == '__main__':
10      main()
```

Execute the program (“add.py”) from the command line:

```
$ python add.py 1 2 3 4
```

Reading input from command line

```
1   import sys
2
3   def main():
4       result = 0
5       for arg in sys.argv[1:]:
6           result += int(arg)
7       print(result)
8
9   if __name__ == '__main__':
10      main()
```

Execute the program (“add.py”) from the command line:

```
$ python add.py 1 2 3 4
```

Reading input from a file

```
1   def grep(filename, word):
2       '''Returns True if filename contains word'''
3       f = open(filename)
4       while True:
5           line = f.readline()
6           if line == '': # no input => stop
7               break
8           if word in line:
9               return True
10      f.close()
11      return False
```

Reading input from a file

- `open(filename)`
  - opens a file for reading
  - returns a “file object”
- `f.readline()`
  - reads one line from file object `f`
  - the empty string (`''`) indicates the end of file
- `f.close()`
  - closes file object `f`
  - (close a file = no further file operations possible)

Open a file

- `open(filename, mode)`
  - opens a file for reading or writing (depending on mode)
- `mode` is a string:
  - r – open the file for reading (the default)
  - w – open the file for writing, truncating the file
  - a – open the file for writing, appending to the end of file
  - t – text mode (the default)
  - b – binary mode

More file operations

- `f.write(something)`
  - writes the string something to file f
- `f.read([size])`
  - reads at most size characters from file f
  - … or the complete file if size not specified
- `f.readlines()`
  - returns a list of lines as strings (= lines of file f)
- `f.readline()`
  - returns one line as a string (= line of file f)

Exercise

```
$ python wc.py goldbug.txt
13760 words
```

- Write a program that counts the number of words in a given input file.
- Hints:
  - `s.split()` splits a string `s` into a list of separate words
  - `s.strip()` returns a copy of the string `s` with leading and trailing whitespace removed
  - `s.rstrip()` returns a copy of the string `s` with trailing whitespace removed

Answer

Stdin, Stdout, Stderr

- Python has automatically three file objects open:
  - sys.stdin = standard input (“keyboard”)
  - sys.stdout = standard output (“monitor”)
  - sys.stderr = standard error (“monitor”)
- Can be extremely useful when combined with unix pipes!
  - gzip -dc example.gz | python myprogram.py
  - cat file1 file2 file3 | python myprogram.py
  - python myprogram.py | gzip > compressed-output.gz

Style guide

- Always close a file object when you’re done with it
- Why? Resources are limited!
  - maximum of 12288 open files on this laptop
  - maximum of 1024 open files on our linux machines

What‘s wrong with this?

```
1   def grep(filename, word):
2       '''Returns True if filename contains word'''
3       f = open(filename)
4       while True:
5           line = f.readline()
6           if line == '': # no input => stop
7               break
8           if word in line:
9               return True
10      f.close()
11      return False
```

What‘s wrong with this?

```
1   def grep(filename, word):
2       '''Returns True if filename contains word'''
3       f = open(filename)
4       while True:
5           line = f.readline()
6           if line == '': # no input => stop
7               break
8           if word in line:
9               return True    # ← !
10      f.close()              # ← !
11      return False
```

The with statement

- Always close a file object when you‘re done with it
- The with statement is a convenient way to do this automatically

```
1 def grep(filename, word):
2     with open(filename) as f:
3         while True:
4             line = f.readline()
5             if line == '':
6                 break
7             if word in line:
8                 return True
9     return False
```

The with statement

- `with open(filename) as var`
  - opens the file ⟨filename⟩
  - assigns the corresponding file object to ⟨var⟩
  - automatically closes the file when we leave the with-block
- More generally:
  - `with` can be used with any kind of “context manager”
  - Context-managers are useful to automatically trigger certain actions when we enter or leave the with-block

Reading input with for-loops

- File objects can be used in for loops
  - in each iteration step, we read one line of the input file

```
1 def grep(filename, word):
2     with open(filename) as f:
3         for line in f:
4             if word in line:
5                 return True
6     return False
```

Style guide

```
with open(filename) as f:
    for line in f:
        ...
```

```
with open(filename) as f:
    for line in f.readlines():
        ...
```

- 2nd version reads in the complete file before we start iterating over individual lines
  - doesn't work with very large files!
  - ⇒ Prefer 1st version

Exercise #1

Write a program grep.py that searches a given input file for lines containing a word, and prints all matching lines. For instance,

```
$ python grep.py goldbug.txt Island
```

should print all lines in goldbug.txt that contain the string “Island”. (4 lines)

Exercise #2

Implement a program that reads in a file and prints for each word how often it occurs in the file.

```
$ python wordcount.py wsj00.txt
Mortimer 1
foul  1
Heights  4
four  13
...
```

Answer

Exercise #3

Write a program wordcount2.py that counts how often each word occurs in the file, and how often it has been tagged with which POS.

```
$ python wordcount2.py wsj00-pos.txt
Mortimer 1  NNP   1
foul  1  JJ 1
...
reported 16 VBN   7  VBD   9
before   26 RB 6  IN 20
allow 4  VB 2  VBP   2
...
```

Encodings

Encodings

```
with open('example.txt') as f:
    for line in f:
        <more code>
...
UnicodeDecodeError: 'utf8' codec can't decode bytes in
position 118-123: unsupported Unicode code range
```

example.txt

Eine Ausnahme oder Ausnahmesituation (engl. exception) bezeichnet in der Computertechnik ein Verfahren, Informationen über bestimmte Programmzustände - meistens Fehlerzustände - an andere Programmebenen zur Weiterbehandlung weiterzureichen ...

What are Encodings?

- Strings are sequences of characters
- Internal representation of strings:
  - sequences of bytes / numbers
- Encodings …
  - specify how to represent a sequence of characters as sequences of bytes.

Some Encodings

- ASCII
  - characters of the English alphabet are represented as numbers between 32 and 127
  - special characters like Ä, Å, … cannot be represented
- Latin-1 (ISO-8859-1)
  - superset of ASCII
  - some (but not all) special characters can be represented
- UTF-8
  - support for (almost?) all characters of all languages
  - superset of ASCII
  - standard encoding in Python

Files & Encodings

```
...
with open('example.txt', encoding='latin1') as f:
    content = f.read()
...
```

Unicode & Byte-Strings

- Python has two types for strings:
  - "Hello" ⇒ a string (Unicode)
  - b"Hello" ⇒ a byte-string
- Byte-string = sequence of bytes
  - ⇒ max. 255 different characters
  - Note: b"Hello"[0] ⇒ 72
- Useful if encoding not known or the text file is not properly encoded (e.g. mix of UTF-8 and latin-1).
- See www.python.org for more details.

Unicode & Byte-Strings

- Converting between strings and byte-strings:
  - `s.encode([encoding])`
    returns a byte-string for string s
  - `b.decode([encoding])`
    returns a string for byte-string b

```
>>> "Hällo".encode("latin1")
b"H\xe4llo"
>>> b"H\xe4llo".decode("latin1")
"Hällo"
```

Achtung!

```
>>> "Hällo".upper()
'HÄLLO'
>>> "Hällo".encode("latin1").upper().decode("latin1")
'HäLLO'
>>> len("Hällo")
5
>>> len("Hällo".encode("latin1"))
5
>>> len("Hällo".encode("utf-8"))
6
```

Achtung!

```
>>> "Hällo".encode("utf-8")
b'H\xc3\xa4llo'
>>> "Hällo".encode("utf-8")[:2]
b'H\xc3'
>>> "Hällo".encode("utf-8")[:2].decode("utf-8")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf8' codec can't decode byte
0xc3 in position 1: unexpected end of data
```

Exceptions

Exceptions

```
1   import sys
2
3   def main():
4       words = 0
5       with open(sys.argv[1]) as f:
6           for line in f:
7               words += len(line.split())
8       print(words, 'words')
9
10  if __name__ == '__main__':
11      main()
```

```
$ python wordcount.py exampl.txt
...
IOError: [Errno 2] No such file or
directory: 'exampl.txt'
```

Exceptions

- Exceptions are errors that occur at runtime
  - ⇒ usually result in an error message
  - ⇒ python stops executing the program :-(:-(:-(
- Some errors are fatal
  - ⇒ stopping the program is the only things we can do
- However, not all errors are fatal and should be explicitly handled within the program itself.

An Example

```
3  def main():
4      words = 0
5      try:
6          with open(sys.argv[1]) as f:
7              for line in f: words += len(line.split())
8      except IOError:
9          print('cannot open file:', sys.argv[1])
10     except IndexError:
11         print('No input file specified!')
12     else:
13         print(words, 'words')
14
15 if __name__ == '__main__':
16     main()
```

Catching exceptions

```
try:
    # statements that can cause exceptions
except Exception_1:
    # here we handle exceptions of type Exception_1
...

except Exception_k:
    # here we handle exceptions of type Exception_k
else:
    # executed if try-block caused no exceptions
finally:
    # always executed, clean-up code
```

- Exceptions are not mutually exclusive (class hierarchy)
  - ⇒ the first appropriate exception handler is executed

Another example

```
def incr(d, k):
    '''Adds 1 to value of key k in dict d'''
    try:
        d[k] += 1
    except KeyError:
        d[k] = 1
```

Some Builtin Exceptions

- ArithmeticError
  - for instance: 1/0
- IOError
  - file not found, disk full, etc.
- IndexError
  - access to a list with a too large index (['a', 'b'][3])
- KeyError
  - access to a dict with key not found in the dict ({'a':1}['b'])
- …

Raising exceptions

- The statement raise ⟨Exception⟩ raises an exception

```
1 def find(pairs, key):
2     for (k, v) in pairs:
3         if k == key:
4             return v
5     raise KeyError
6
7 print(find([('a',1), ('b',2), ('c',3)], 'b')
# prints 2
8 print(find([('a',1), ('b',2), ('c',3)], 'd')
KeyError
```

Exercise #4

- Implement a calculator for simple arithmetic expressions in revese polish notation. Make sure that your implementation detects and signals invalid inputs.