Python for Language Processing

(4) Files, Exceptions, Encodings

Dr. Jakob Prange

Fakultät für Angewandte Informatik - Universität Augsburg

CL Fall School 24

Credit: This course is based on material developed by
Annemarie Friedrich, Stefan Thater, Michaela Regneri, and Marc Schulder at Saarland University

Running Python

Through which interfaces can Python programs be executed?

Running Python

Through which interfaces can Python programs be executed?

- IDE (PyCharm, VSCode, IDLE)
  - from the command line

Reading input from the 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 command line like this:

```
$ python3 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`
  - no further file operations possible
  - memory can be freed up

Opening 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, overwriting old contents
  - `'a'` – open the file for writing, appending to the end
  - `'t'` – text mode (the default)
  - `'b'` – binary mode

More file operations

- `f.write(somestring)`
  - writes the string `somestring` to the 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 strings (= lines of file `f`)

Exercise

```
$ python3 wc.py wsj00-200.txt
4783 words
```

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

Stdin, stdout, stderr

Python automatically keeps 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`

Stdin, stdout, stderr

```
1  import sys
2
3  def main():
4    count = 0
5    while True:
6      line = sys.stdin.readline()
7      if line == '':
8        break
9      count += len(line.split())
10   print(count, 'words')
11
12 if __name__ == '__main__':
13   main()
```

Resource Management

- Always close a file object when you are done with it!
- Why? Resources are limited!
  - upper limit on open files
  - conflicts when multiple processes try to change the same file

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 are done with it!
- The `with` statement is a convenient way to do this automatically

```
1  def grep(filename, word):
2      '''Returns True if filename contains word'''
3      with open(filename) as f:
4          while True:
5              line = f.readline()
6              if line == '':  # no input -> stop
7                  break
8              if word in line:
9                  return True
10     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 type of “context manager”
  - Context managers are useful to automatically trigger certain actions upon entering or leaving the `with`-block
  - How are `with`-blocks similar to / different from namespaces?

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      '''Returns True if filename contains word'''
3      with open(filename) as f:
4          for line in f:
5              if word in line:
6                  return True
7      return False
```

Reading input with `for`-loops

Which one is better?

```
1 with open(filename) as f:
2     for line in f:
3         ...
```

```
1 with open(filename) as f:
2     for line in f.readlines():
3         ...
```

Hint: You can find the answer at

`https://docs.python.org/3/tutorial/inputoutput.html`

Reading input with `for`-loops

Which one is better?

```
1 with open(filename) as f:
2     for line in f:
3         ...
```

```
1 with open(filename) as f:
2     for line in f.readlines():
3         ...
```

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

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()
```

```
$ python3 wordcount.py exampl.txt
...
FileNotFoundError:  [Errno 2] No such file or directory:
'exampl.txt'
```

Exceptions

- Exceptions are errors that occur at runtime
  - $\rightarrow$ usually result in an error message
  - $\rightarrow$ Python stops executing the program :(
- Some errors are fatal
  - $\rightarrow$ stopping the program is the only thing we can do
- However, not all errors are fatal. If possible, they should be explicitly handled within the program itself.

An Example

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

Catching Exceptions

```
try:
    # statements that can cause exceptions
except Exception_1:
    # handle exceptions of type Exception_1
...
except Exception_k:
    # handle exceptions of type Exception_k
except:
    # handle any other exception (NOT RECOMMENDED!)
else:
    # executed if try-block caused no exceptions
finally:
    # always executed, clean-up code
```

Catching Exceptions

```
1  try:
2      ...
3  except Exception_1:
4      ...
5  ...
6  except Exception_k:
7      ...
8  except:
9      ...
10 else:
11     ...
12 finally:
13     ...
```

- Exception types are not mutually exclusive
  - e.g. `FileNotFoundError` is a subclass of `IOError`
  - Every exception is a subclass of `Exception`
  - $\rightarrow$ the first applicable exception handler is executed

Another Example

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

Some Builtin Exceptions

- `ArithmeticError`
  - for instance: 1/0 ($\rightarrow$ `ZeroDivisionError`)
- `IOError`
  - file not found, disk full, etc.
- `IndexError`
  - access to a list with too large (or too small) index
- `KeyError`
  - access to a dict with key not found in the dict
- ...

Exceptions

- Use the `raise` statement to throw a specific 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')
8  # prints 2
9  print(find([('a',1), ('b',2), ('c',3)], 'f')
10 KeyError: 'f'
```

Encodings

- Strings are internally represented as sequences of bytes (numbers)
- File contents are stored in a specific *encoding*
  - Encoding = mapping from numbers to characters
- Default value of `encoding` keyword arg of `open()` is system-dependent
  - → better to specify

```
1 with open('example.txt') as f:
2     content = f.read()
```

```
UnicodeDecodeError:
'utf-8' codec can't decode
bytes in position 118-123
```

```
1 with open('example.txt', encoding='latin-1') as f:
2     content = f.read()
```

- More info:
  - `https://docs.python.org/3/tutorial/inputoutput.html`
  - `https://realpython.com/python-encodings-guide/`

Doctests

- Exceptions are great for dealing with “expected things that might go wrong”
- Before it comes to that, we should make sure that everything else goes right!
  - → Test your code!
  - → Write docstrings explaining what a function does and how to use it!

```
1  def gcd(m, n):
2      '''
3      Computes the greatest common denominator of m and n.
4      '''
5      return m % n
6  # This implementation is clearly wrong.
7  # If we use this gcd() function somewhere else,
8  # expecting it to work correctly, we have a problem.
```

Doctests

- Doctests are a convenient way of combining these two things in a formal way

```
1  def gcd(m, n):
2      '''
3      Computes the greatest common denominator of m and n.
4
5      >>> gcd(8, 12)
6      4
7      >>> gcd(54, 24)
8      6
9      '''
10     return m % n
11
12 if __name__ == '__main__':
13     import doctest
14     doctest.testmod(verbose=True)
```

Doctests

- It's important to write multiple tests, in case one of them gets the correct answer by accident!

```
File "<input>", line 5, in __main__.gcd
Failed example:
    gcd(8, 12)
Expected:
    4
Got:
    8
Trying:
    gcd(54, 24)
Expecting:
    6
ok!
```

Doctests

- Expected output behavior must be exactly as if you were running the interactive console (like in IDLE)
- Careful with strings and complex types

```
1  def hello():
2      '''
3      >>> hello()
4      hello world
5      >>> hello()
6      'hello world'
7      >>> print(hello())
8      hello world
9      '''
10     return "hello world"
```

```
Failed example:
    hello()
Expected:
    hello world
Got:
    'hello world'
Trying:
    hello()
Expecting:
    'hello world'
ok
Trying:
    print(hello())
Expecting:
    hello world
ok
```

The following slides are for your reference.

Command Line Arguments and Parsing

Executing a Python program on the command line:

```
python3 myProgram.py filename1.txt 5
```

```
1  import sys
2  ...
3   # sys.argv is a list of strings
4  if __name__ == "__main__":
5    file1 = open(sys.argv[1])
6    count = int(sys.argv[2])
```

- Try out: what is the content of `sys.argv[0]`?
- Here, you need exceptions to handle invalid argument values!
- Basics: https://www.pythonforbeginners.com/system/python-sys-argv
- Advanced: https://docs.python.org/3/library/argparse.html

Filepath handling with `os`

`os.path` is a convenient module for handling file paths in an operating system-independent way (and to do other useful things).

```
python3 myProgram.py /home/anne/data grades.csv
```

```
1  import sys
2  ...
3   # sys.argv is a list of strings
4  if __name__ == "__main__":
5    basePath = sys.argv[1]
6    filename = sys.argv[2]
7    full_path = os.path.join(basePath, filename)
```

https://docs.python.org/3/library/os.path.html

- Check out: `os.listdir()`.
- Check out `shutil` module for file operations (e.g., moving files or removing a folder). https://docs.python.org/3/library/shutil.html