Compact Course Python

Michaela Regneri & Andreas Eisele

Lecture 4

Overview

- More on Strings
- Modules
- Exceptions
- Input and Output in Python
- Encodings

Strings: Methods

http://docs.python.org/3.1/library/stdtypes.html # string-methods

- `s1.count(s2)`: count occurrences of `s2` in `s1`
- Index of the first (last) occurrence of `s2` in `s1`:
  - `s1.index(s2 [, start [, end]])`   `(rindex)`
  - `s1.find(s2 [, start [, end]])`   `(rfind)`

    (Error if `s1` is not in `s2`)
- Properties of `s1` (`False` for empty `s1`):
  - Digits? `s1.isdigit()`
  - Letters? `s1.isalpha()`
  - Digits or letters (+'_'): `s1.isalnum()`
  - whitespaces: `s1.isspace()`

Strings: Methods

http://docs.python.org/3.1/library/stdtypes.html # string-methods

- Methods for case sensitivity:
  - `s1.isupper()` / `s1.islower()`: all upper / lower case? (`False` for strings without case)

  `s1.upper()` / `s1.lower()`: a copy of `s1` with all charachters upper / lower case
  - `s1.capitalize()`: copy of `s1` with first character in upper case
  - `s1.swapcase()`: copy of `s1`, upper and lower case exchanged
  - `s1.title()` (also: `s1.istitle()`): A copy of `s1` each letter after a whitespace or punctuation is upper case

Strings: Methods

http://docs.python.org/3.1/library/stdtypes.html # string-methods

- strip whitespaces [characters of `s2`] on the left and right: `s1.strip([s2])` (`lstrip`, `rstrip`)
- Splitting strings: `s1.split([sep1, sep2,...])`
  - Return: an array of strings that are left when one cuts `s1` around all occurrences of `sepx`
  - If no delimiters are specified, whitespaces are assumed as delimiters
  - consecutive delimiters separate the empty String

```
>>> 'aa,,a.b'.split([','])
['Aa','', a.b ']
```

Modules

- Modules are collections of classes / functions, or code in general (= *. py files)
- Modules are reusable, one can access code from other modules
- Python has (besides "builtins") some standard modules, which one can resort to when necessary (such as `sys`)
- In order to use a module and their elements, you have to import it (with `import <modulname>`)

```
import sys
[...]
a = sys.argv[0]
```

module name

Modules

- To use a file foo.py a module, you import the module "foo"
- One can also import single slasses or functions of a module with `from`

```
from math import sqrt
[...]
a = sqrt(25)
```

Module

Function

- Python finds a module (without any additional information) only if
  - they are in the same folder as the current module
  - they are in the Python library directory (e.g. under UNIX often `/usr/local/lib/python/`)

Modules

- You can import modules by specifiying the path to a subdirectory explicitly:

```
import foo.bar.module
```

if `module` is in the subfolder `foo/baar` of the current directory

- using the keyword **`as`** one can bind variables to module name and use them later instead of the full name (handy for long names)

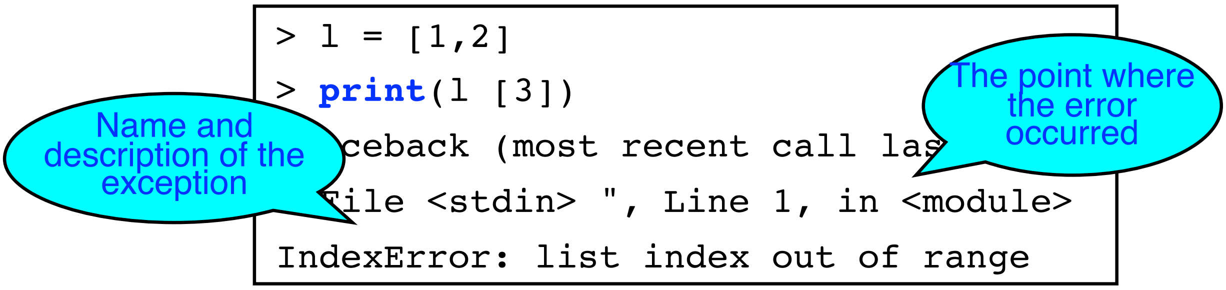
```
import foo.bar.blah.blubb.module as fb
i = fb.method()
```

Exceptions

- Exceptions are errors that occur during a program run
- up to know we simply tried avoid exceptions
- There are ways to handle exceptions, so the program will continue after the exception
- It may also be useful to raise exceptions (in contrast to empty return values, etc.)

Exceptions

- There are a number of exceptions that can occur in Python's standard modules (http://docs.python.org/3.1/library/exceptions.html)
- An Example: accessing a nonexistent list index



```
> l = [1,2]
> print(l [3])
  ceback (most recent call las
  File <stdin> ", Line 1, in <module>
IndexError: list index out of range
```

Name and description of the exception

The point where the error occurred

Catching exceptions

- Exceptions can be caught with "try ... except"
- If an exception occurs in block1, the execution of block1 is canceled and block2 is executed
- afterwards, the program flow is resumed after the try construct
- there can be a else statement after except; block3 will be executed if there was no exception in block1

```
try:
  block1
except:
  block2
```

```
try:
  block1
except:
  block2
else:
  block3
```

Catching exceptions

- except:catches everything
- To react to specific exceptions, you write their class names after except (except IndexError: ...)
- If you expect several exceptions and want to treat each of them in a different way, can you can define more except blocks
- else always comes after the last except block

```
try:
  block1
except <Error1>:
  block2
except <Error2>:
  block3
[...]
else:
  blockx
```

Exceptions: `finally`

```
try:
 block1
except <Exc>:
 block2
finally:
 block3
```

- `finally` guarantees that the following code will be execute in any (!) case
- if an exception is caught, first `block2` will be executed, after that `block3`
- If an unhandled exception occurs, first `block3 is` executed and then the exception is raised again
- `else` comes before `finally` (in notation and in the execution)

Exceptions as classes

- All built-in Python exceptions are derived from `Exception` (or `BaseException`)
- ie `except Exception` (`except BaseException`) catches all exceptions (equivalent to `except` without argument)
- If we need to access the specific instance of an exception, wee need to them to a variable using `as`

```
try:
 block1
except Exception as e:
 print(e)
```

Defining and throwing exceptions

- we can define our own exceptions
- Exceptions should inherit from `Exception` (and have to inherit `BaseException`)
- The default message is defined in the `__str__` method

```
class MyIndexError(Exception):
  def __init__(self, length, index):
    self.length = length
    self.index = index

  def __str__(self):
    ret = 'Only ' + str(self.length)
    ret += ' items in the list, '
    ret += 'index ' + str(self.index)
    ret += ' is invalid."
    return ret
```

Defining and throwing exceptions

- Exceptions are „thrown“ with `raise<Exception>`
- `<Exception>` is an instance of an Exception class

```
> raise MyIndexError(2, 5)
Traceback (most recent call last):
  File <stdin> ", Line 1, in <module>
__main__.MyIndexError: Only 2 items in the
list, index 5 is invalid.
```

- if the `__init__` Method of the Exception class does not need any additional arguments, you can simply write the class name

Defining and throwing exceptions

```
> raise Exception
Traceback (most recent call last):
  File <stdin> , Line 1, in <module>
Exception
```

- The base class has an Exception *optional* String argument

```
> raise Exception('Moep.')
Traceback (most recent call last):
  File <stdin> , Line 1, in <module>
Exception: Moep.
```

Defining and throwing exceptions

- If one wants to re-throw an exception but needs to do something beforehand one can use `raise` without parameters
- `raise` is looking for "active" exceptions and raises the most recent one

```
try:
 block1
except:
 # Do something
raise
```

- after the try-except block, the exception is no longer active (not even in finally)

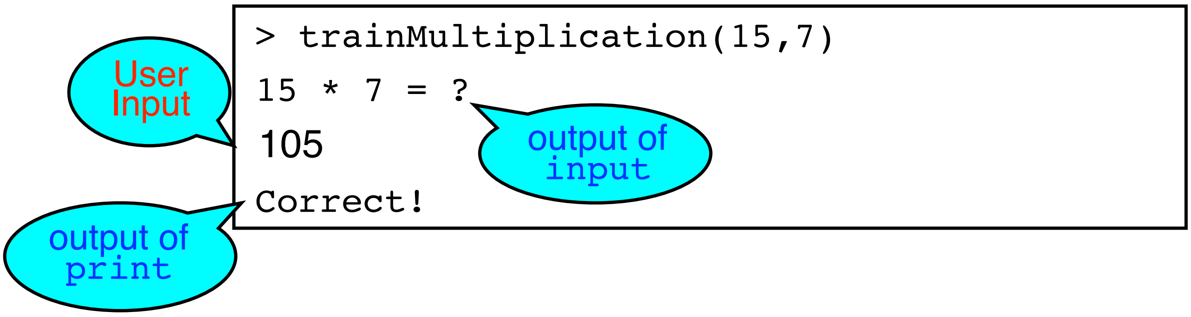
Input and output: console

- output: already seen (`print`)
- Command line arguments (input): `sys.argv[i]`
- Interaction during the program run: `input([string])`
  - `string` is printed right before the user input is read
  - the return value contains the user input that followed after the method execution (sent by pressing Return)
  - `input` returns the entered string

Input and output: Console

- an example:

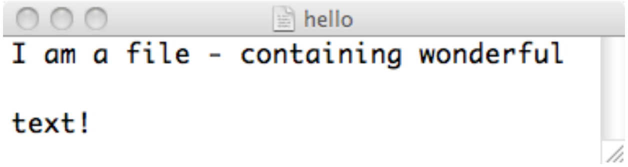
```
def trainMultiplication(x, y):
    i = input(str(x) + '*' + str(y) + '= ? \n')
    if int(i) == (x * y):
        print('Correct!')
    else:
        print('Wrong.')
```



```
> trainMultiplication(15,7)
15 * 7 = ?
105
Correct!
```

Input and output: files

- Working with files in Python means works with `file` objects
- you get them e.g. `open(string)`



```
I am a file - containing wonderful

text!
```

```
f = open('hello')
print(f.read())
f.close()
```

⇒

```
'I am a file - containing wonderful \n text!'
```

Input and output:
File Handling

- all operations on files start at the current "position" in the file
- The position changes when reading / writing. Right after opening the file it is 0
- print the current position: `f.tell()`
- set the current position: `f.seek(index)`
- To avoid errors, you have to close opened files if they are no longer needed: `f.close()`

Short interlude: the with statement

```
with foo as var:
  block
```

- with ensures (among other things) that objects follow a certain "life cycle"
- for file objects, this means that they are closed right before the `with` block ends
- internally: when starting the `with` block, the `__enter__` method (of foo) is called, and at its end the `__exit__` method is called

```
with open('hello.txt') as f:
   f.read()
```

Input and output: reading files

- `f.read()`: returns the (text) content of `f`
- `f.readline()`: returns `f` line by line (new call - next line)
- `f.readlines()`: returns the list of lines in `f`
- iterating over the lines in `f` directly:

```
with open(file) as f:
  i = 1
  for l in f:
    print(i + '. line:' + l)
```

The position after the last read character in the file is stored, read all the reading methods from the current position!

Input and output: writing files

- Writing access to files can be obtained with additional parameters (*flags*) In `open`:
  - `open(f, 'w')`: writing access
  - `open(f, 'a')`: writing access, text is appended
  - `open(f, 'r+')`: reading and writing access
  - `open(f, 'r+a')`: reading and writing (text appended)
  - `open(f, 'r')`: reading
- Without the second parameter: read only
- you can read the variable `f.mode` to retrieve those rights again

Input and output: writing files

- `f.write(string)`: writes `string` to `f`
- `f.writelines(seq)`
  writes all the elements in `seq` (some sequence type) to `f` (no automatic line break!)
- `f.flush()`:
  writes everything that was previously passed to write actually to file. This is executed automatically when calling `f.close()` (and before exiting a with statement)

Input and output: URLs

- `urllib.request` allows to open URLs
- reading web pages (their source code) works like reading files:

```
import urllib.request as url
hp = 'http://www.coli.uni-saarland.de'
for line in url.urlopen(hp):
    print(line)
```

- objects returned by `urlopen` support the reading methods `read()` and `readlines()`
- Copy a web page to a local file:

```
[...]
url.urlretrieve(hp, 'filename.html')
```

unicode strings vs. byte strings

- Python knows two types of strings: unicode and byte strings (`str` and `bytes`)
  - Standard string literals (`"x", 'y'`) Are Unicode strings
  - `b"word"` creates a byte string
- byte strings are internally encoded as a sequence of bytes (restriction to a maximum of 255 different characters)
- unicode strings are internally represented as a sequence of 2 or 4 bytes (they cover virtually all alphabets)

unicode strings vs. byte strings

- One must not mix the two (with concatenation, etc.), but has to convert:
  - String → Byte: `str.encode(unicodeString)`
  - Byte → String: `bytes.decode(byteString)`
- `urlopen` returns byte strings, `open` by default unicode strings (!!!) - in consequence you may only write those then, too!
- If no encoding is specified explicitly, ASCII is assumed

unicode strings vs. byte strings

- if you know nothing about the file you‘re processing, it might be easier to work with byte strings only (as long as you don‘t need a readable output)

- (Reading and writing) file content as byte strings: `open(f, 'br')`
  - **b** can be put right before the other *Flags* are in `open`
  - if you use **b** as a flag, you need a second parameter indicating whether you need reading or writing access (etc.) to the file

Encodings

- Strings are sequences of characters
- computers don‘t know characters: internally, strings are represented as sequences of numbers
- we need a mapping from numbers to characters
- such mappings are called *encodings*

Encodings

- ASCII is a simple (7-bit) encoding, which maps latin characters to numbers from 32 to 127 (numbers ≤ 31 are control characters).

```
for c in 'python':
  print(ord(c), end =" ")
```

```
112 121 116 104 111 110
```

- ASCII does not cover umlauts etc.
- Some extensions of ASCII
  - ISO-8859-1 ("latin1") - Western European languages
  - ISO-8859-2 ("latin2") - Eastern European Languages

Declaring encodings

- If the source code contains non-ascii characters, the encoding for string literals has to be defined explicitly:

```
# -*- coding: latin1 -*-
print("Hällo, Wörld!")
```

- Without explicitly specifying the encoding, the example above won‘t compile. However, the same result is achieved like this:

```
print("H\xe4llo, W\xf6rld!")
```

Unicode

- How do we handle several texts with different encodings at the same time?
- Or languages with more than 256 characters?
- Unicode!
  - discards the restriction that characters must be represented as exactly one byte
  - includes all (most) characters of most languages

Unicode and encodings

- unicode defines how characters are represented as code points
  - The code points 0-256 are identical to latin-1
- code points are numbers (hex numbers here)

```
0061 'a'; LATIN SMALL LETTER A
0062 'b'; LATIN SMALL LETTER B
0063 'c'; LATIN SMALL LETTER C
...
007B '{'; LEFT CURLY BRACKET
```

Unicode and encodings

- an encoding defines how unicode characters are represented in memory.
- encodings can be incomplete (eg, ASCII).
- A "naive" complete encoding would represent each character as a sequence of 32-bit numbers (4 bytes).
  - but: os dependendy (byte order), uses a lot of memory, representations contain zeros

Unicode Transformation Format

- UTF-8 is a commonly used, compact (8-bit) encoding for Unicode:
  - can represent all Unicode code points
  - most characters (ASCII) are represented by a single byte.
- encoding:
  - code-Point <128 ⇒ 1 byte
  - code-Point ≥ 128 ⇒ 2-4 bytes
- Note: UTF-8 is not Unicode!

Unicode & files

- Stream objects (files, URLs) always do have *some* encoding
- If you do not know which one, you can simply work with byte strings, as long as possible
- If you need a string, you have to decode it again, either like this:

```
with open(file, encoding="UTF-8") as f:
```

... or like that:

```
with open(file, 'br') as f:
  for line in f.readlines():
    astring = str(line, encoding="UTF-8")
```

Summary

- More Basics: Modules & Exceptions
- Input / output: console, files, URLs
- String Handling: Byte-vs. Unicdoe strings, encodings