Lecture #6

Text manipulation
File access

Lecture week 6 – 1

String formatting

Why string manipulation?

- String is very frequently used data type
  > Representing data: names, sentences, etc.
  > Outputs of program execution
  > Common format for data exchange or storage
    - Even numbers are often represented by corresponding string
- You can learn good programming techniques
  > You can learn many simple, but useful in data handling

What we will cover for string manipulation

- String formatting

  > Adjust string length, space, precisions of numbers, etc.

- Functions for string class

String formatting

- Evaluate a string with given arguments according to the specification in format string
- Evaluated strings can be
  > Assigned to a variable
  > Passed to a function as an argument
    - Especially to print() function

Types of string formatting

- Python provides several ways for string formatting
  > printf-style string formatting (all python versions)
    - Support format string similar to that of printf() function in C

  > format() function with format string syntax (python 3, python 2.6+)
    - Provide more flexible format string

  > Literal string interpolation (python 3.6+)
    - Support embedded expressions in format string

  > Template string (python 2+): similar to printf-style

printf-style string formatting (% or interpolation operator)

- Usage: *format % values*
  - > *format*: a format string that contains conversion specifiers
  - > *values*: one of the following
    - One value
    - Tuple
    - A mapping object (e.g., a dict object)
- Behavior
  - > Convert *values* into a string with formats given in *format*

Conversion specifier

1. Begins with '%'
2. (Optional) mapping key
   > Parenthesized sequence of characters (e.g., (name))
3. (Optional) conversion flag
4. (Optional) Minimum field width
5. (Optional) Precision
   > Specifies the number of decimal points after '.'
6. (Optional) length modifier: not used
7. Conversion type: a letter specifies the formats

Conversion type

Conversion	Meaning
'd' or 'i'	Signed integer decimal.
'o'	Signed octal value.
'u'	Obsolete type – it is identical to 'd'.
'x' or 'X'	Signed hexadecimal (lowercase ('x') or uppercase('X').
'e' or 'E'	Floating point exponential format (lowercase ('e') or uppercase('E').
'f' or 'F'	Floating point decimal format.
'g' or 'G'	Floating point format. Uses lowercase ('g') or uppercase ('G') exponential format if exponent is less than -4 or not less than precision, decimal format otherwise.
'c'	Single character (accepts integer or single character string).
'r', 's', 'a'	String (converts any Python object using repr(), str(), and ascii(), respectively)
'%'	No argument is converted, results in a '%' character in the result.

Example: printf-style string formatting

```
var = 42
string = 'Answer'
s1 = 'The %s is %d.' % (string, var)
print(s1)
print('0123456789' * 2)
print('%10d' % var)
print('%010d' % var)
print('%-10d' % var)
```

```
The Answer is 42.
01234567890123456789
        42
0000000042
42
```

Example: printf-style string formatting

```
pi = 3.141592653589793
print('%f' % pi)
print('%.2f' % pi)
print('%.10f' % pi)
print('%10.2f' % pi)
print('%g, %g, %g' % (var, pi,
10000000000))
print('%(name)s: %(average)g' %
        {'name': 'Alice',
         'average': 99.9})
```

```
3.141593
3.14
3.1415926536
      3.14
42, 3.14159, 1e+10
Alice: 99.9
```

String formatting with format() function

- Usage: *string*.format*(arguments)*
  - > *string*: a format string with replacement fields
  - > *arguments*: one of the following
- Replacement fields: {*fieldname*:*format_spec*}
  - > *fieldname*: specifies fields to be converted
  - > *format_spec*: specifies conversion formats (formatting specifier)
- Note
  - > formatting specifiers are similar to conversion specifier of printf-style formatting, but with more features and enhanced usability

Example: format()

```
d = 42
f = 3.14
s = 'apple'
print('{} {} {}'.format(d, f, s))
print('{2} {0} {1}'.format(d, f, s))
print('{0} is {0}'.format(s))
print('{d} {f} {s}'.format(d=6,
                           f=1.618,
                           s='pineapple'))
print('{0:d} {0:f} {0:g}'.format(42))
```

```
42 3.14 apple
apple 42 3.14
apple is apple
6 1.618 pineapple
42 42.000000 42
```

Example: format() (cont.)

```
print('0123456789' * 2)
print('{:10d}'.format(42))
print('{:>10d}'.format(42))
print('{:<10d}'.format(42))
print('{:^10d}'.format(42))
print('{:10.2f}'.format(3.1415926535))
print('{:1d}'.format(42))
print('{:5.5f}'.format(3.1415926535))
```

```
01234567890123456789
        42
        42
42
    42
      3.14
42
3.14159
```

Formatted string literals

- Also called as literal string interpolation
- A formatted string literal or f-string

  > a string literal that is prefixed with 'f' or 'F'

  > replacement fields within {}

  - Formatting syntax is similar to that of format()
  - You can put any expression within replacement fields

Example: formatted string literals

```
pi = 3.14159265
width = 10
precision = 2
print('0123456789' * 3)
print(f'{pi}')
print(f'{pi:10.2f}')
print(f'{pi:{width}.{precision}f}')
print(f'{max(width, precision)}')
t = [42, 1024, 23]
for i in range(len(t)):
    print(f'{i}: {t[i]}')
```

```
012345678901234567890123456789
3.14159265
      3.14
      3.14
10
0: 42
1: 1024
2: 23
```

Lecture week 6 – 2

Functions on strings

String functions (partial)

- Usage: *string.function(arguments)*
  > Formatting: rjust(), ljust(), center(), format()
  > Stripping: strip(), lstrip(), rstrip()
  > Join/split: join(), split(), splitlines()
  > Find/count: find(), rfind(), count()
- What you can have as *string*
  > string, a variable referring to a string, a function returning a string
- Note: most functions return a string
  > Exceptions: find(), rfind() return an integer
  > Exceptions: split(), splitlines() return a list with strings

챆Formatting: ljust(), rjust(), center()

- Function prototypes
  - > str.ljust(*width[, fillchar]*)
  - > str.rjust(*width[, fillchar]*)
  - > str.center(*width[, fillchar]*)
- Arguments
  - > *width*: integer value
  - > *fillchar*: a character for padding
- Return value
  - > left/right/center-justified string of a length *width*, padded with *fillchar*

Example: ljust(), rjust(), center()

```
s = '123'
print(s.ljust(5))
print(s.rjust(5))
print(s.center(5))
print(s.ljust(5, '*'))
print(s.rjust(5, '-'))
print(s.center(5, '.'))
print(s)  # no change on s
```

```
123␣␣
␣␣123
␣123␣
123**
--123
.123.
123
```

strip(), lstrip(), rstrip() functions

- **str.strip(*[chars]*)**
  - > Return a copy of the string with the leading and trailing characters removed
  - > The *chars* argument specifies the set of characters to be removed, and when omitted, whitespaces are removes
  - > Frequently used after reading a line from file/console/network
- Examples

```
print('  a string  '.strip())          a string
print('  a string  '.lstrip())        a string␣␣
print('  a string  '.rstrip())        ␣␣a string
```

join() function

- **str.join(*iterable*)**

  > Return a string which is the concatenation of the strings in *iterable*

- Examples

```
print('.'.join('abc'))                  a.b.c
print('.'.join(['foo', 'bar']))         foo.bar
print('*-*'.join(['foo', 'bar']))       foo*-*bar
print('₩n'.join(['foo', 'bar']))        foo
                                         bar
```

split() function

- **str.split(*sep*=None, *maxsplit*=-1)**
  - > Return a list of the words in the string, using *sep* as the delimiter string.
  - > If *maxsplit* is given, at most *maxsplit* splits are done (thus, the list will have at most maxsplit+1 elements).
  - > If *maxsplit* is not specified or -1, then there is no limit on the number of splits (all possible splits are made).
  - > If *sep* is given, consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, '1,,2'.split(',') returns ['1', '', '2']).
  - > The *sep* argument may consist of multiple characters (for example, '1<>2<>3'.split('<>') returns ['1', '2','3']).
  - > Splitting an empty string with a specified separator returns [''].

splitlines() function

- ***str.splitlines([keepends])***
  - > Return a list of the lines in the string, breaking at line boundaries.
  - > Line breaks are not included in the resulting list unless *keepends* is given and true.
  - > This method splits on the following line boundaries.
  - > In particular, the boundaries are a superset of universal newlines

Example: split()

```
colors = 'red,green,blue'
tokens = colors.split(',')
print(tokens)
words = 'hello world'
print(words.split())
```

```
['red', 'green', 'blue']
['hello', 'world']
```

Example: splitlines()

```
quote_text = '''First, solve the problem. Then, write the code. - John Johnson
Talk is cheap. Show me the code. - Linus Torvalds
Good design adds value faster than it adds cost. - Thomas C. Gale
Computers are good at following instructions, but not at reading your mind. - Donald Knuth
Without requirements or design, programming is the art of adding bugs to an empty text file. - Louis Srygley'''
quotes = quote_text.splitlines()
for quote in quotes:
    print(quote)
```

find(), rfind(), count() functions

- **str.find(*sub[, start[, end]]*)**
  - > Return the lowest index in the string where substring *sub* is found within the slice *sub[start:end]*
  - > Return -1 if *sub* is not found
- **str.rfind(*sub[, start[, end]]*)**
  - > Return the highest index in the string where substring *sub* is found within the slice *sub[start:end]*
  - > Return -1 if *sub* is not found
- **str.count(*sub[, start[, end]]*)**
  - > Return the number of non-overlapping occurrences of substring *sub* within the slice *sub[start:end]*

Example: find(), rfind()

```
t = '01234567890123456789012345678901234567890'
s = 'A horse is a horse, of course, of course'
print(s.find('rse'))                                    4
print(s.find('rse', 10))                                15
print(s.rfind('rse'))                                   37
print(s.rfind('rse', 0, 10))                            4
print(s.find('mule'))                                   -1
print(s.count('rse'))                                   4
print(s.count('rse', 10))                               3
print(s.count('rse', 10, 20))                           1
```

Lecture week 6 – 3

File access (text files)

What is a file on computer systems?

- What is a file?
  - > File is a collection of data composed of numbers or texts
  - > Stored at HDD, SSD, USB drive, or on clouds
  - > Usually distinguished by the name and the place where a file resides (i.e., file path)
  - > Typically file extension (strings after the last '.') specifies the type of files (e.g., .py, .html, .txt, .mp3, .jpg)

Text files

- Text files

  > Files with human readable characters such as alphabets, numbers, special characters (e.g., “, ’, ?), typically composed of multiple lines

  > Can represent different characters by text encodings

  > There are many types of text files, e.g.,

  - python files (.py)
  - html files (.html) / cascading style sheet (.css)
  - text files (.txt)

Binary files

- Binary files

  > Files that are NOT text files

  > Most multimedia files such as pictures, videos, music are binary files, e.g.,

  - Graphic files (.jpg, .png)
  - Videos files (.avi, .mp4)
  - music files (.wav, .mp3)

  > You may use python packages to handle these types of files

File access

- To access a file, you need to follow some steps

  > Open a file: open()

  > Access

  - Read data from a file: read()
  - Write data to a file: print()

  > Close a file: close()

- Note: the above steps are typical to access any resources on computer such as file, memory, I/O

Reading a text file

- Open a file named ‘quotes.txt’ and read the content of the file

```
fileobj = open('quotes.txt', 'r')
file_contents = fileobj.read()
fileobj.close()
```

- Explanation
  - > open(*filename*, *mode*): open a file and returns a file object
    - *filename* may include path and *‘r’* means open a text file for reading
  - > read(): return the content of a file as a string
    - Lines are separated by ‘₩n’
  - > close(): close a file

Reading a text file (cont.)

- Another way of opening a file (a recommended way)

```
with open('quotes.txt', 'r') as fileobj:
    file_contents = fileobj.read()
```

- Explanation
  - > With with keyword, you can open a file and assign the return value (file object) to a variable after as keyword (*fileobj* in this example)
  - > *fileobj* is valid within the code block of with statement
  - > Properly close a file after the code block, even though an exception occurs

Writing a text file

- Write contents to a file named *'test.txt'*

```
fileobj = open('test.txt', 'wt')
print('Hello, world!', file=fileobj)
print('Hi, again.', file=fileobj)
fileobj.close()
```

- Explanation
  - > *'wt'* in the argument of open(): open a text file for writing
  - > file keyword argument in print(): specify the output file

Writing a text file (cont.)

- A recommended way of writing a file

```
with open('test.txt', 'wt') as fileobj:
    print('Hello, world!', file=fileobj)
```

- Explanation

  > Automatically close the file after the code block

Lecture week 6 – 4

Errors and Exception Handling

Types of Errors

- While programming, you may generate errors
  > You need to figure out why such errors occur and how to fix them
- Types of errors (in general)
  > Syntax Errors
  > Runtime Errors
  > Logical Errors
- In python, most runtime errors can be handled with exceptions

Example: syntax error

```
while True
    number = input('Enter a number: ')
    print('If I double the number, it is', number * 2)
```

```
 File "errors.py", line 1
   while True
              ^
SyntaxError: invalid syntax
```

Example: logic error

- Example of code

```
number = input('Enter a number: ')
print('If I double the number, it is', number * 2)
```

- Example of code execution

```
Enter a number: 42
If I double the number, it is 4242
```

Example: exception

- Example of code

```
number = int(input('Enter a number: '))
print('If I double the number, it is', number * 2)
```

- Example of code execution

```
Enter a number: abc
Traceback (most recent call last):
  File "errors.py", line 1, in <module>
    number = int(input('Enter a number: '))
ValueError: invalid literal for int() with base 10: 'abc'
```

Examples of exceptions in python

- **ArithmeticError**
  - > **OverflowError**: raised when the result of an arithmetic operation is too large to be represented
  - > **ZeroDivisionError**: raised when the second argument of a division or modulo operation is zero
- **KeyError**: raised when a dictionary key is not found
- **IndexError**: raised when a sequence subscript is out of range
- **NameError**: raised when a local or global name is not found
- **FileExistsError**: raised when trying to create a file or directory which already exists
- **FileNotFoundError**: raised when a file or directory is requested but doesn't exist

Ideas of exception handling in python

- Put statements of normal flow in ‘try’ block
- Exceptions are raised in the try block, they are handled by ‘except’ blocks
  > You can have more than one except blocks by specifying the name of exceptions you want to handle
- You may also raise exception with ‘raise’ statement if necessary

Example: exception handling

- Example of code

```
try:
    number = int(input('Enter a number: '))
    print('If I double the number, it is', number * 2)
except:
    print('Please enter an integer number!!!')
```

- Example of code execution

```
Enter a number: abc
Please enter an integer number!!!
```

Else clause for try/exception statement

- Else clause for try/exception statement
  - > Executed only when no exception is raised in ‘try’ block
  - > Used to avoid accidentally catching exception that isn’t NOT generated by try/except statement
- Example

```
try:
    number = int(input('Enter a number: '))
except:
    print('Please enter an integer number!!!')
else:
    print('I got got', number)
```

Argument for exceptions

- You may get detailed information from Exception

```
try:
    a = 3 / 0
except Exception as err:
    print(type(err))
    print(err)
    print(err.args)
```

- Output

```
<class 'NameError'>
name 'a' is not defined
("name 'a' is not defined",)
```

Raising exception

- You may raise an exception with ‘raise’ statement

  > Note: you can define your own exception by inheriting Exception classes

```
try:
    raise NameError('Foo', 'Bar')
except NameError as err:
    print(err.args)
```

- Output

```
('Foo', 'Bar')
```

More realistic code: user input

- Repeat until you get an expected input

```
while True:
    try:
        number = int(input('Enter a positive integer number: '))
        if number <= 0:
            print('Please enter a positive integer number!!!')
            continue
        break
    except:
        print('Please enter a positive integer number!!!')
```

More realistic code: user input (cont.)

- Repeat until you get an expected input

```
while True:
    try:
        number = int(input('Enter a positive integer number: '))
        if number <= 0:
            raise ValueError
        break
    except:
        print('Please enter a positive integer number!!!')
```

More realistic code: file handling

```
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
```

ANY QUESTIONS?