Text Data, Files, and Exceptions

String Representation

A string value is represented as a sequence of characters delimited by quotes

Quotes can be single (') or double (")

What if the string includes both ' and "?

Escape sequence \' or \" is used to indicate that a quote is not the string delimiter but is part of the string value

Function `print()` interprets the escape sequence

Another example:

- `\n` is an escape sequence that represents a new line

```
>>> excuse = 'I am sick'
>>> excuse = "I am sick"
>>> excuse = 'I'm sick'
SyntaxError: invalid syntax
>>> excuse = "I'm sick"
>>> excuse = "I'm "sick""
SyntaxError: invalid syntax
>>> excuse = 'I'm "sick"'
SyntaxError: invalid syntax
>>> excuse = 'I\'m "sick"'
>>> excuse
'I\'m "sick"'
>>> print(excuse)
I'm "sick"
>>> excuse = 'I\'m ...\n... "sick"'
>>> excuse
'I\'m ...\n... "sick"'
>>> print(excuse)
I'm ...
... "sick"
```

Indexing operator

- Indexing operator return the character in index i (as single character)
- Indexing operator also can be used to obtain a slice of string

`s[i:j]` : the slice of `s` starting at index `i` and ending before index `j`

`s[i:]` : the slice of `s` starting at index `i`

`s[:j]` : the slice of `s` ending before index `j`

Exercise

The indexing operator can also be used to obtain slices of a list as well. Let list lst refer to list

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

Write Python expressions using list lst and the indexing operator that evaluate to:

a) ['a', 'b', 'c', 'd']
b) ['d', 'e', 'f']
c) ['d']
d) ['f', 'g']
e) ['d', 'e', 'f', 'g', 'h']
f) ['f', 'g', 'h']

```
>>> lst[:4]
['a', 'b', 'c', 'd']
>>> lst[3:6]
['d', 'e', 'f']
>>> lst[3:4]
['d']
>>> lst[-3:-1]
['f', 'g']
>>> lst[3:]
['d', 'e', 'f', 'g', 'h']
>>> lst[-3:]
['f', 'g', 'h']
```

String Method

Usage	
s.capitali	
s.count(ta	
s.find(tar	
s.lower()	
s.replace(	
s.split(se	
s.strip()	
s.upper()	returns lowercase copy of s

```
>>> link = 'http://www.main.com/smith/index.html'
>>> link[:4]
'http'
>>> link[:4].upper()
'HTTP'
>>> link.find('smith')
20
>>> link[20:25]
'smith'
>>> link[20:25].capitalize()
'Smith'
>>> link.replace('smith', 'ferreira')
'http://www.main.com/ferreira/index.html'
>>> link
'http://www.main.com/smith/index.html'
>>> new = link.replace('smith', 'ferreira')
>>> new
'http://www.main.com/ferreira/index.html'
>>> link.count('/')
4
>>> link.split('/')
['http:', '', 'www.main.com', 'smith', 'index.html']
```

Strings are immutable; none of the string methods modify string `$ink`

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

String Method

Suppose we need to pick up the date and time components of string `event`

Puntuation makes it difficult to use method `split()`

```
>>> event = "Tuesday, Feb 29, 2012 -- 3:35 PM"
>>> table = str.maketrans(':,-', 3*' ')
>>> event.translate(table)
'Tuesday  Feb 29  2012    3 35 PM'
>>> event.translate(table).split()
['Tuesday', 'Feb', '29', '2012', '3', '35', 'PM']
>>>
```

Solution: replace punctuation with blank spaces

Usage	Explanation
`str.maketrans(old, new)`	returns a table mapping characters in string `old` to characters in string `new`
`s.translate(table)`	returns a copy of `s` in which the original characters are replaced using the mapping described by `table`

String formatting, better printing

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- So far, we just used the default print function
- We can do many more complicated things to make the output "prettier" and more readable
- The basic form of the format method is :

```
"<format string>".format (data1, data2, …)
```

Format Command

- Each {} can include format commands that provide directives about how a particular object is to be printed
- The four piece of information
  - align is optional (default left)
  - width is how many spaces (default just enough)
  - .precision is for floating point rounding (default no rounding)
  - type is expected type (error if the arguments is the wrong type)

Format Command

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- The general structure of the most commonly used parts of the format command is:

```
{:[align][minimum width][.precision][type]}
```

where the square bracket is indicated **optional** arguments

s	string
d	decimal integer
f	floating-point decimal
e	floating-point exponential
%	floating-point as percent

<	left
>	right
^	center

Format Command Example


```
print ('{:>10s} is {:<10d} years old'.format('Bill', 25))
```

String 10 spaces wide including the object, right justified

Decimal 10 spaces wide including the object, left justified

Output:

Bill is 25 years old

10 spaces 10 spaces

Example

```
>>> for i in range(1,8):
        print(i, i**2, 2**i)


1 1 2
2 4 4
3 9 8
4 16 16
5 25 32
6 36 64
7 49 128
>>> for i in range(1, 8):
        print('{} {:2} {:3}'.format(i, i**2,
2**i))


1  1   2
2  4   4
3  9   8
4 16  16
5 25  32
6 36  64
7 49 128
>>>
```

plus a blank space between the columns

reserves 3 spaces for `2**i`

Specifying field width

The `format()` method can be used to line up data in columns

Numbers are aligned to the right

Strings are aligned to the left

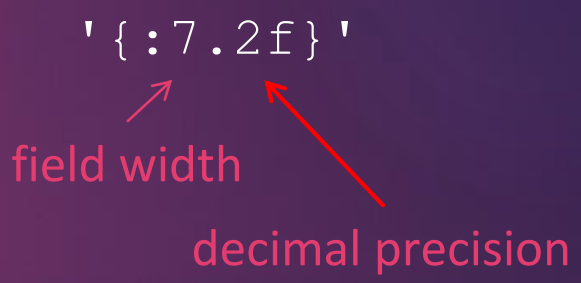
```
>>> lst = ['Alan Turing', 'Ken Thompson', 'Vint Cerf']
>>> for name in lst:
        fl = name.split()
        print(fl[0], fl[1])


Alan Turing
Ken Thompson
Vint Cerf
>>> for name in lst:
        fl = name.split()
        print('{:5} {:10}'.format(fl[0], fl[1]))


Alan  Turing
Ken   Thompson
Vint  Cerf
>>>
```

Output format type

Inside the curly braces of a placeholder, we can specify the field width, the type of the output, and the decimal precision

Type	Explanation
b	binary
c	character
d	decimal
X	hexadecimal
e	scientific
f	fixed-point

```
>>> n = 10
>>> '{:b}'.format(n)
'1010'
>>> '{:c}'.format(n)
'\n'
>>> '{:d}'.format(n)
'10'
>>> '{:X}'.format(n)
'A'
>>> '{:e}'.format(n)
'1.000000e+01'
>>> '{:7.2f}'.format(n)
'  10.00'
>>>
```



Floating point precision

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- Can round floating point numbers to specific number of decimal places

```
>>> import math
>>> print (math.pi)
3.141592653589793
>>> print ("Pi is {:.4f}".format(math.pi))
Pi is 3.1416
>>> print ("Pi is {:4.4f}".format(math.pi))
Pi is  3.1416
>>> print ("Pi is {:8.2f}".format(math.pi))
Pi is    3.14
>>> print ("{:8.2%}".format(2/3))
66.67%
```

What is a file?

- A file is a collection of data that is stored on a secondary storage like a harddisk or a flashdisk
- Accessing a file means establishing a connection between the file and the program and moving data between the two

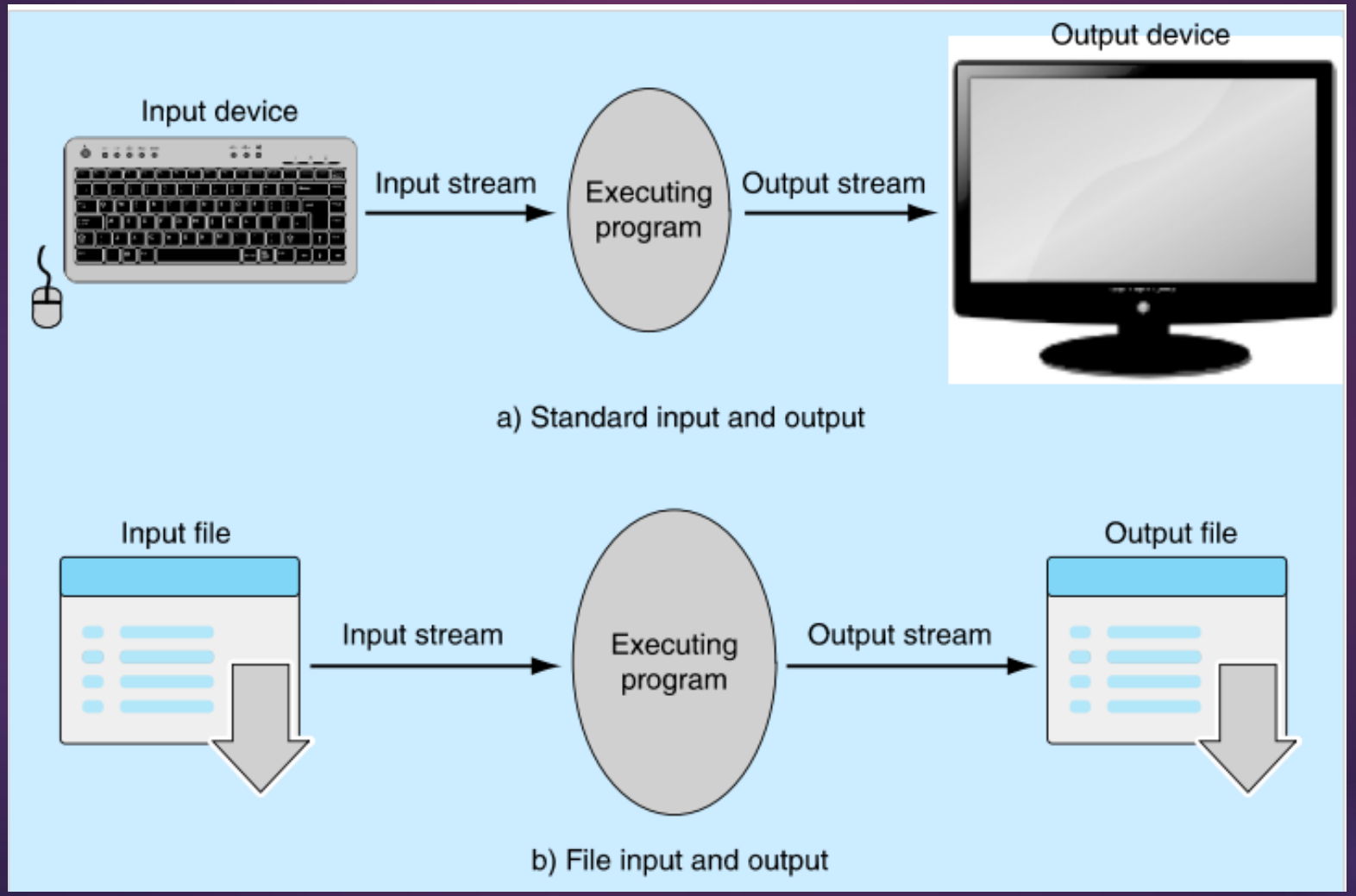
Two type of files

- Text file
  - organized as Unicode/ASCII data and generally human readable
- Binary file
  - Organized as a sequence of bytes and is generally not human readable

File Objects or Stream

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- When opening a file, you create a file object or file stream that is a connection between the file information on disk and the program
- The stream contains a buffer of the information from the file, and provides the information to the program
- If everything is working well, the whole process will be invisible to the programmer



Buffering

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- Reading from disk is very slow
  - Computer will read a lot of data from a file
    - It will buffered in a primary memory
  - It means that the file object contains a copy of information from the file

File Object

- `my_file` is the file object. It contains the buffer information.
- The open function creates the connection between the disk file and the file object.
  - The first parameter is the file name
  - The second parameter is the mode to open it

```
my_file = open("file.txt", "r")
```

Where is the disk file?

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- When opened, the filename is come in one of two forms:
  - "file.txt" assumes the file name is file.txt and it is located in the current folder
  - "c:\program\file.txt" is the fully qualified file name and includes directory information: c:\program

Open file mode

The file mode defines how the file will be accessed

Mode	How Opened	File Exists	File Does Not Exist
'r'	read-only	Opens that file	Error
'w'	write-only	Clears the file contents	Creates and opens a new file
'a'	write-only	File contents left intact and new data appended at file's end	Creates and opens a new file
'r+'	read and write	Reads and overwrites from the file's beginning	Error
'w+'	read and write	Clears the file contents	Creates and opens a new file
'a+'	read and write	File contents left intact and read and write at file's end	Creates and opens a new file

These are all equivalent →

```
>>> infile = open('example.txt', 'rt')
>>> infile = open('example.txt', 'r')
>>> infile = open('example.txt', 't')
>>> infile = open('example.txt')
```

File method

There are several “file” types; they all support similar “file” methods

- Methods `read()` and `readline()` return the characters read as a string
- Methods `readlines()` returns the characters read as a list of lines
- Method `write()` returns the number of characters written

Usage	Description
`infile.read(n)`	Read n characters starting from cursor; if fewer than n characters remain, read until the end of file
`infile.read()`	Read starting from cursor up to the end of the file
`infile.readline()`	Read starting from cursor up to, and including, the end of line character
`infile.readlines()`	Read starting from cursor up to the end of the file and return list of lines
`outfile.write(s)`	Write string s to file outfile starting from cursor
`infile.close(n)`	Close file infile

Reading a file

```
1 The 3 lines in this file end with the new line character.\n
2 \n
3 There is a blank line above this line.\n
```

example.txt

When the file is opened, a cursor is associated with the opened file

The initial position of the cursor is:

- at the beginning of the file, if file mode is `r`
- at the end of the file, if file mode is `a` or `w`

```
>>> infile = open('example.txt')
>>> infile.read(1)
'T'
>>> infile.read(5)
'he 3 '
>>> infile.readline()
'lines in this file end with the new line
character.\n'
>>> infile.read()
'\nThere is a blank line above this line.\n'
>>> infile.close()
>>>
```

Writing file

```
1 This is the first line. Still the first line…\n
2 Now we are in the second line.\n
3 Non string value like 5 must be converted first.\n
4 Non string value like 5 must be converted first.\n
```

test.txt

```
>>> outfile = open('test.txt', 'w')
>>> outfile.write('T')
1
>>> outfile.write('his is the first line.')
22
>>> outfile.write(' Still the first line...\n')
25
>>> outfile.write('Now we are in the second line.\n')
31
>>> outfile.write('Non string value like '+str(5)+' must be converted first.\n')
49
>>> outfile.write('Non string value like {} must be converted first.\n'.format(5))
49
>>> outfile.close()
```

Writing a file

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- Careful with write modes
  - Be careful if you open a file with ‘w’ mode. It sets an existing file’s content to be empty, destroying any existing data
  - The ‘a’ mode is nicer, allowing you to write to the end of existing file without changing the existing contents

Text file use strings

- Remember that in a text file, everything is a string
  - Everything we read is string
  - Writing a file only in a string

Writing a file

- Once you have created a file object, opened for writing, you can use the print command by adding the argument `file = <file object>`

```
filenya = open("hasil.txt", "w")
print ("Hallo, selamat menulis file", file = filenya)
print ('Having fun writing a file', file = filenya)
filenya.close()
```

Close a file

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- When the program is finished with a file, we close the file:
  - Flush the buffer content from computer to the file
  - Shutdown the connection to the fie
- `close` is a method of a file obj
  - `file_obj.close()`
- All files should be closed after used

Exception

- Type of error
  - Syntax of error
  - Erroneous state error

```
>>> excuse = 'I'm sick'
SyntaxError: invalid syntax
>>> print(hour+':'+minute+':'+second)
Traceback (most recent call last):
  File "<pyshell#113>", line 1, in <module>
    print(hour+':'+minute+':'+second)
TypeError: unsupported operand type(s) for +:
'int' and 'str'
>>> infile = open('sample.txt')
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    infile = open('sample.txt')
IOError: [Errno 2] No such file or directory:
'sample.txt'
```

Syntax Error

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF COMPUTER SCIENCE

- Error that are due to the incorrect format of Python statement
- It occur while the statement is being translated to machine language and before it is being executed

```
>>> (3+4]
SyntaxError: invalid syntax
>>> if x == 5
SyntaxError: invalid syntax
>>> print 'hello'
SyntaxError: invalid syntax
>>> lst = [4;5;6]
SyntaxError: invalid syntax
>>> for i in range(10):
print(i)
SyntaxError: expected an indented block
```

Erroneous State Errors

The program execution gets into an erroneous state

When an error occurs, an “error” object is created

- This object has a type that is related to the *type* of error
- The object contains information about the error
- The default behavior is to print this information and interrupt the execution of the statement.

The “error” object is called an exception; the creation of an exception due to an error is called the raising of an exception

```
>>> 3/0
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    3/0
ZeroDivisionError: division by zero
```

```
>>> lst
Traceback (most recent call last):
  File "<pyshell#57>", line 1, in <module>
    lst
NameError: name 'lst' is not defined
```

```
>>> lst = [12, 13, 14]
>>> lst[3]
```

```
>>> lst * lst
Traceback (most recent call last):
```

```
>>> int('4.5')
```

```
>>> int('4.5')
Traceback (most recent call last):
  File "<pyshell#61>", line 1, in <module>
    int('4.5')
ValueError: invalid literal for int() with
base 10: '4.5'
```

Exception Type

Some of the built-in exception classes:

Exception	Explanation
KeyboardInterrupt	Raised when user hits Ctrl-C, the interrupt key
OverflowError	Raised when a floating-point expression evaluates to a value that is too large
ZeroDivisionError	Raised when attempting to divide by 0
IOError	Raised when an I/O operation fails for an I/O-related reason
IndexError	Raised when a sequence index is outside the range of valid indexes
NameError	Raised when attempting to evaluate an unassigned identifier (name)
TypeError	Raised when an operation of function is applied to an object of the wrong type
ValueError	Raised when operation or function has an argument of the right type but incorrect value

Exception Handling

- Basic Idea
  - Keep watch on particular section of code
  - If we get an exception, throw that exception
  - Look for a catcher that can handle that kind of exception
  - If found, handle the exception, otherwise, let the python handle it (usually halts the program)

Handle the user input

- In general, we assume that the input we receive is correct (from a file or user)
- There is always a chance that the input could be wrong
  - The elegant and well written program should handle it
- Failure to enforce correct input is a common source of software security problem

Input is 'Evil'

- '*Writing Secure Code*' by Howard and LeBlanc
  - "All input is evil until proven otherwise"
- Most security problem holes in programs are based on programmer assumptions about the input
- Secure program can protect themselves from evil input

General form of exception handling

- `Try-except` statement

```
try:
    <body>
except <ErrorType>:
    <handler>
```

- When python encounter a `try` statement, it attempts to execute the statement inside the body
  - If no error, it will pass to the statement after the `try-except`

General form of exception handling

- If an error occur while executing the statement in body
  - Python look for an except clause with matching error type
  - If one is found, the handler code will be executed
- The `try-except` statement can be used to catch any kind of error and provide graceful exit
- Single `try` can have multiple `except` statement

Multiple except

- Multiple `except` clause act like `elif`s. If error occurs, Python will try each `except` clause looking for one that matches the type of error
- The bare `except` clause at the bottom acts like an else and catches any error without a specific match
- If there is no bare `except` clause at the end and none of the `except` clauses match, the program will crash and report the error

Try block

- The try block contains code that we want to monitor for errors during its execution
- If an error occurs in anywhere in that try block, Python will looks for handler that can deal with the error
- If no matched handler exists, Python handles it by halting the program with an error messages

Except block

- An (multiple) `except` block is associated with a `try` block
- Each exception names a type of exception it is monitoring for
- If the error occur in `try` block matches the exception type, then `except` block is executed


UNIVERSITAS
INDONESIA
Veritas, Probitas, Iustitia
FACULTY OF
COMPUTER
SCIENCE