Files

File I/O

Files

- Files need to be opened in Python before they can be read from or written into
- Files are opened in Python using the `open()` built-in function

```
open(file, mode='r', buffering=-1, encoding=None,...
```

Opening Files

open()

```
open(file, mode='r', buffering=-1, encoding=None,...
```

Parameters

`file`: Relative or absolute pathname to a file

`mode`: Combination of `'rwxa'` and `'bt+'`

- `'r'` Read file in text mode (default)
- `'w'` Write to a file in text mode. Truncates first
- `'a'` Open in append mode, text mode. No truncation
- `'rb'`,`'wb'`,`'ab'` Binary mode
- `'+'` Update mode, read and write

Opening Files

- `open()`ing a file returns a stream object

```
>>> a = open('rfc-968.txt', 'rt', encoding='utf-8')
>>> type(a)
<class '_io.TextIOWrapper'>
```

Unicode Support

Character Encoding

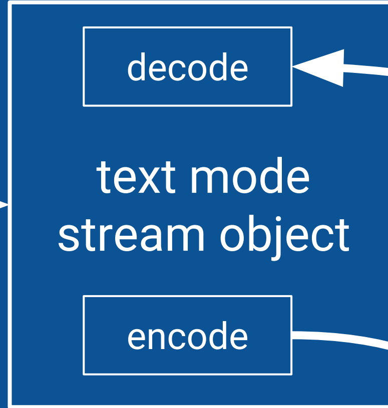
And the `encoding` parameter

- At lowest level, all files are stored as a sequence of bytes
- Encoding determines how byte sequences in a file are mapped to characters representing a script
  - `encoding` parameter only makes sense in text mode
- ASCII is an encoding where a single byte can represent any character in the set (which is primarily the english alphabet)
- Unicode has more symbols that can fit in a single byte
- The default encoding is *platform dependent*
  - i.e depends on where you run your program. Not good
- Always specify an encoding when working with text files
  - `'utf-8'` will be the usual choice

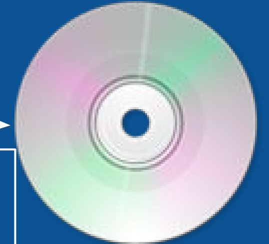
Unicode Support

```
Twas the night b
efore start-up a
nd all through t
he net,.  not a
packet was movin
g; no bit nor oc
tet..The enginee
rs rattled their
 cards in despai
r,.  hoping a ba
d chip would blo
w with a flare..
```

str



bytes



```
54 77 61 73 20 74 68 65 20 6e 69 67 68 74 20 62
65 66 6f 72 65 20 73 74 61 72 74 2d 75 70 20 61
6e 64 20 61 6c 6c 20 74 68 72 6f 75 67 68 20 74
68 65 20 6e 65 74 2c 0a 20 20 6e 6f 74 20 61 20
70 61 63 6b 65 74 20 77 61 73 20 6d 6f 76 69 6e
67 3b 20 6e 6f 20 62 69 74 20 6e 6f 72 20 6f 63
74 65 74 2e 0a 54 68 65 20 65 6e 67 69 6e 65 65
72 73 20 72 61 74 74 6c 65 64 20 74 68 65 69 72
20 63 61 72 64 73 20 69 6e 20 64 65 73 70 61 69
72 2c 0a 20 20 68 6f 70 69 6e 67 20 61 20 62 61
64 20 63 68 69 70 20 77 6f 75 6c 64 20 62 6c 6f
77 20 77 69 74 68 20 61 20 66 6c 61 72 65 2e 0a
```

Stream Methods

Reading

- `read()` method
  Read n characters from the stream and return it as an `str/bytes`

```
>>>> a = open('rfc-968.txt', 'rt', encoding='utf-8')

>>>> char = a.read(1) # Read one character from the stream
>>>> type(char)
<class 'str'>

>>>> print(char)
T
```

Stream Object

Stream position

- A stream object remembers the position where the last read or write ended
- The next read/write operation begins from that position

```
>>>> a.read(3)
'was'

>>>> a.read(1)
' '

>>>> a.read(4)
'the '

>>>> a.read(12)
'night before'
```

Stream Object

EOF

- Reading past the end of a file returns an empty `str`
- No exception is raised

```
>>> char = a.read(1)
>>> print(repr(char))
'5'

>>> char = a.read(1)
>>> print(repr(char))
'\n'

>>> char = a.read(1) # EOF Reached
>>> print(repr(char))
''

>>> char = a.read(1)
>>> print(repr(char))
''
```

Stream Methods

Reading

- `readline()` method

  Read a line and return as `str`/`bytes`

```
>>>> a = open('rfc.txt','rt',encoding='utf-8')
>>>> print(a.readline())
Twas the night before start-up and all through the net,

>>>> print(a.readline())
  not a packet was moving; no bit nor octet.

>>>> print(a.readline())
The engineers rattled their cards in despair,

>>> a.close()
```

Stream Methods

Reading

- Stream objects can be used as iterators
- Each iteration returns the next line from the file

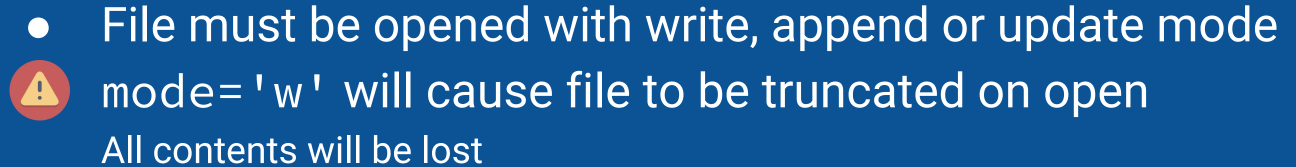
```
>>> a = open('../../week1/cities.txt','rt', encoding='utf-8')
>>> for line in a:
...     print(line)
...
New York

Phoenix

...

New York

New York

New York
>>> a.close()
```

Stream Methods

Writing

`write()` method

Write string to stream




```
$ ls -l rfc.txt
-rw-rw-r-- 1 ebasheer ebasheer 1574 Sep 14 20:42 rfc.txt
```

```
>>> a = open('rfc.txt','wt',encoding='utf-8')
>>> a.close()
```

```
$ ls -l rfc.txt
-rw-rw-r-- 1 ebasheer ebasheer 0 Sep 14 20:42 rfc.txt
```

Stream Methods

write()

```
$ cat cities.txt
New York
Los Angeles
Chicago
```

```
>>>> a = open('cities.txt','at',encoding='utf-8')
>>>> a.write('Philadelphia\n')
13
>>>> a.close()
```

```
$ cat cities.txt
New York
Los Angeles
Chicago
Philadelphia
```

Stream Methods

`close()`

Flush contents and free system resources (file descriptor)

- Open file descriptors take up memory
- Important to `close()` a file as soon as we're done using it
- Closing a file also flushes any buffered data

Python

`with` statement

- Execute a block of statements with predefined operations performed before and after execution of the block
- Entry and exit operations depend on context manager object

```
with <expression> as <var>:
    statement-1
    statement-2
    statement-3
```

- `<expression>` must return a context manager object
- Stream objects returned by `open()` are also context managers

with statement

Python

```
>>> with open('rfc.txt','rt', encoding='utf-8') as infile:
...     print("is the file closed:", infile.closed)
...     print(infile.readline())
...     print(infile.readline())
...
is the file closed: False
Twas the night before start-up and all through the net,

  not a packet was moving; no bit nor octet.

>>> print(infile.closed)
True
```

Stream Methods

Standard Streams

stdin, stdout, stderr

- Only functions we’ve seen so far for I/O from the standard streams
  `input()`, `print()`
- `read()`, `write()` etc. methods can be used directly
- Module sys contains already opened standard streams

Python

Standard Streams

stdin, stdout, stderr

```
import sys

for line in sys.stdin:
    sys.stdout.write(line.upper())
```

```
$ python3 stdin.py
one
ONE
two
TWO
a sentence
A SENTENCE
Ctrl-d
$

$ echo 'this is sent to stdout'|python3 stdin.py
THIS IS SENT TO STDOUT
```

Python

Exceptions

- Python uses exceptions to signal errors
  - Some languages rely on functions returning specific values or storing values in special variables to indicate an error
- An exception in Python is a signal that triggers the interpreter to interrupt the normal flow of the program
- Exceptions can be *raised* by a statement in the program or by the interpreter itself when it encounters an error
- *Unhandled* exceptions cause the interpreter to terminate the program

Exceptions

```
while True:
    a = int(input('Enter first integer:'))
    b = int(input('Enter second integer:'))
    q = a / b
    print('a/b is: ',q)
```

```
$ python3 zexcept.py
Enter first integer:1
Enter second integer:3
a/b is:  0.3333333333333333
Enter first integer:7
Enter second integer:0
Traceback (most recent call last):
  File "zexcept.py", line 4, in <module>
    q = a / b
ZeroDivisionError: division by zero
```

- Division by zero causes interpreter to raise an exception
- Exception is unhandled, therefore terminates program

Python

Exception handling

- Exceptions are handled using syntax called a `try...except` statement

```
try:
    statement-1
    statement-2
    statement-3
except <exception-1>:
    statement-e1
except <exception-2>:
    Statement-e2
...
```

Exceptions

```
while True:
    try:
        a = int(input('Enter first integer:'))
        b = int(input('Enter second integer:'))
        q = a / b
        print('a/b is: ',q)
    except ZeroDivisionError:
        print("Can't divide by zero. Try again")
```

```
$ python3 zexcept_handler.py
Enter first integer:99
Enter second integer:0
Can't divide by zero. Try again
Enter first integer:1
Enter second integer:2
a/b is:  0.5
Enter first integer:A
Traceback (most recent call last):
  File "zexcept_handler.py", line 3, in <module>
    a = int(input('Enter first integer:'))
ValueError: invalid literal for int() with base 10: 'A'
```

```python
def one():
    try:
        two()
    except ZeroDivisionError:
        print('ZeroDivisionError Handler in one()')

def two():
    three()

def three():
    four()

def four():
    try:
        1/0
    except ValueError:
        print('ValueError Handler in four()')

if __name__ == '__main__':
    one()
```

- Exceptions are propagated up the call stack
    - Until a handler is found
    - Or program is terminated
- `raise` statement can be used to create an exception event

```
$ python3 propagate.py
ZeroDivisionError Handler in one()
```

Exceptions

Exceptions

Example

- Depending on mode. `open()` raises different exceptions

```
>>>> a = open('noexist.dat', 'rb')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'noexist.dat'
```

Exceptions

```
>>>> try:
....     a = open('noexist.dat', 'rb')
.... except FileNotFoundError:
....     print('Would you like to create the file')
....     reply = input('y/n')
....     if reply[0].lower() == 'y':
....         a = open('noexist.dat','wb')
....
Would you like to create the file
y/ny
>>>> type(a)
<class '_io.BufferedWriter'>
```

String Formatting

- Allows us to generate strings from various combinations of literal text, objects and values

```
print('x has the value', x, 'given value', y)
```

- The `format` method of `str` objects provides much more flexibility in generating formatted strings than
    - `print()` function or concatenation operator

format()

Examples: Simple substitution

```
>>>> x = 10.11
>>>> msg = 'substitute here {} the value of x'.format(x)
>>>> print(msg)
substitute here 10.11 the value of x
```

File I/O

format()

Examples: substitution by position

```
>>>> x,y = 10.1, 22
>>>> msg = 'x has value {}, and y is {}'.format(x,y)
>>>> print(msg)
x has value 10.1, and y is 22
```

format()

Examples: specifying order

```
>>>> first, last = 100, 999
>>>> msg = 'last {1} comes first {0} in arguments'.format(last,first)
>>>> print(msg)
last 100 comes first 999 in argument
```

format()

Examples: using keywords

```
>>>> i = 1.22
>>>> V = 12200
>>>> msg = 'current is {curr} A, voltage is {volt}
V'.format(volt=V,curr=i)
>>>> print(msg)
current is 1.22 A, voltage is 12200 V
```

format()

Examples: format specifiers

```
>>>> lst1 = [1, 20000, 23, 323, 3, -16, -33.333]
>>>> lstval = ['value','weight','run_length','current','voltage','x','y','last']
>>>> for name,val in zip(lstval,lst1):
....     print(name,val)
....
value 1
weight 20000
run_length 23
current 323
voltage 3
x -16
y -33.333
```

format()

Examples: format specifiers

```
>>>> for name,val in zip(lstval,lst1):
....     msg = '{0:>15} : {1:<7}'.format(name,val)
....     print(msg)
....
          value : 1
         weight : 20000
     run_length : 23
        current : 323
        voltage : 3
              x : -16
              y : -33.333
```