File I/O

A Python `file` object serves as a link to a file residing on your computer
Data can be read to or from a file using an open `file` object
Both binary and text files are supported
The `file` object has methods for many common operations you might want to perform on files:

```
input = open('my_file.txt', 'r')
line = input.readline()
lines = input.readlines()
```

open for reading
read one line as a string
read all lines to a list of strings

```
output = open('my_file.txt', 'w')
print("Hello, World", file=output)
output.close()
```

open for writing
output a line to the file
close the file

File I/O

Output Example

writepowers.py

```
fo = open('powers.txt', 'w')
for x in range(1,101):
    x2 = x * x
    x3 = x2 * x
    x4 = x2 * x2
    print(x, x2, x3, x4, file=fo)
fo.close()
```

- The variable `fo` is assigned to the file object
- `fo.close()` isn't strictly necessary - Python closes the file object for us when the program exits

File I/O

Output Example

```
$ cat powers.txt
1 1 1 1
2 4 8 16
3 9 27 81
4 16 64 256
5 25 125 625
6 36 216 1296
...
99 9801 970299 96059601
100 10000 1000000 100000000
```

- `print` has automatically separated our fields by spaces

File I/O

Input Example

readpowers1.py

```
fi = open('powers.txt', 'r')
for line in fi.readlines():
    print(line)
fi.close()
```

```
$ python readpowers1.py
1 1 1 1

2 4 8 16

3 9 27 81
...
100 10000 1000000 100000000
```

⚠ note the extra blank line between each output - `readlines()` doesn't strip the line ending, and `print` adds another.

File I/O

Input Example

readpowers2.py

```
fi = open('powers.txt', 'r')
for line in fi.readlines():
    print(line.rstrip('\n'))
fi.close()
```

```
$ python readpowers2.py
1 1 1 1
2 4 8 16
3 9 27 81
...
100 10000 1000000 100000000
```

File I/O

readpowers3.py Input Example

```
fi = open('powers.txt', 'r')
for line in fi.readlines():
    line = line.rstrip('\n')
    fields = line.split(' ')
    print(fields)
fi.close()
```

```
$ python readpowers3.py
['1', '1', '1', '1']
['2', '4', '8', '16']
['3', '9', '27', '81']
...
['100', '10000', '1000000', '100000000']
```

each line has been parsed into a *list* of *strings*

File I/O

readpowers4.py

```
fi = open('powers.txt', 'r')
for line in fi.readlines():
    line = line.rstrip('\n')
    fields = line.split(' ')
    for power in fields:
       print(int(power), end='')
    print()  # force a new line
fi.close()
```

```
$ python readpowers4.py
1 1 1 1
2 4 8 16
3 9 27 81
...
100 10000 1000000 100000000
```

Type conversion (casting)

builtin methods exist for converting between data types e.g.:

```
int(), str(), float(), bool()
```

```
>>> int('5')
5
>>> float('4.56')
4.56
>>> int(7.6)
7
>>> int('7.6')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with
base 10: '7.6'
```

OK, rounds down

oops - not OK

Type conversion to `bool`

Almost any object can be interpreted as a boolean value. Most evaluate to `True`, but the following, are `False`:

`0, 0.`	integer 0, float 0
`''`	empty string
`[], ()`	empty list, empty tuple
`None`	the single value of the special `None` type

```
>>> bool(4)
True
>>> bool(0)
False
>>> bool('hello')
True
>>> bool('')
False
```

More Type conversions

Most objects have a string representation, accessed with `str()`

```
>>> str([1, 2, 3])
'[1, 2, 3]'
```

Iterable objects (such as lists, tuples, and strings can be interconverted:

```
>>> tuple([1, 2, 3])
(1, 2, 3)
```

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

ℹ strings are iterable

```
>>> list(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

⚠ oops - integers aren't iterable

`print` formatting

Python supports C-style formatting of strings

`%[n]d`	integer in `n` chars
`%[n][.m]f`	float in `n` chars with `m` decimal places
`%[n][.m]e`	scientific-notation float
`%[n]s`	string in `n` chars
`%%`	the `%` sign

These are used by placing the format specifier within the string and following it with `% (a, b, c, ...)` where a, b, c, are the values to insert into the string. For example,

```
>>> a = 6
>>> '%4d + %3.1f = %9.3e' % (a, 7.3, a+7.3)
'   6 + 7.3 = 1.330e+01'
```

ℹ 1.33×10^{1}, *i.e.* 13.3

`print` formatting

Examples

```
>>> a = 1.07; b = 2.34; c = 0.9; d=-3.77
>>> print(a,b,c,d)
1.07 2.34 0.9 -3.77
```

```
>>> print('%6.2f'*4 % (a,b,c,d))
  1.07  2.34  0.90 -3.77
```

```
>>> print('%6.2f%6.2f\n%6.2f%6.2f' % (a,b,c,d))
  1.07  2.34
  0.90 -3.77
```

Syntactic Sugar

... and other cool Python stuff:

Multiple variables may be assigned in one statement:

```
>>> a, b, c = 10, -8, 'foo'
>>> x = y = z = -1
```

Swapping the values of two variables:

```
>>> a, b = b, a
>>> a
-8
>>> b
10
```

note that the righthand side gets evaluated first

Chaining comparison operators:

```
>>> 5 < b <= 10
True
```

Syntactic Sugar

... and other cool Python stuff:

Using `enumerate` to get an item and its index:

elements2.py

```
elements = ['H', 'He', 'Li', 'Be', 'B']
print(' Z  element')
for i, elm in enumerate(elements):
    print('%2d %5s' % (i+1, elm))
```

```
$ python elements2.py
 Z  element
 1      H
 2     He
 3     Li
 4     Be
 5      B
```

Syntactic Sugar

... and other cool Python stuff:

List *comprehensions*:

```
>>> [x**2 for x in range(6)]
[0, 1, 4, 9, 25]

>>> [-x for x in [1,2,3]]

[-1, -2, -3]

>>> line = '1 2 3 4'
>>> [int(f) for f in line.split()]
[1, 2, 3, 4]

>>> [i for i in range(10) if not i%2]
[0, 2, 4, 6, 8]
```

Errors and Exceptions

We distinguish two kinds of error: *syntax errors* and *exceptions*
Syntax Errors occur when the *grammar* of a Python statement is incorrect, *e.g.*

```
>>> while x > 0
              ^
SyntaxError: invalid syntax
```

Python expects a colon here!

Even if a statement is syntactically correct, an error may occur when Python attempts to execute it, *e.g.*

```
>>> 1 + zz * 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'zz' is not defined
```

we don't have to *declare* the variable `zz`, but it must have been assigned a value to execute the statement `1 + zz * 2`

Errors and Exceptions

`IOError`	e.g. file not found
`IndexError`	an attempt to access a sequence (such as a list with an index out of range)
`TypeError`	an operation or function applied to an object of inappropriate type
`NameError`	a variable name is not recognised
`ValueError`	an operation or function receives an argument of the right type but an inappropriate value
`ZeroDivisionError`	a specific type of ValueError raised when an attempt is made to divide by zero.

Errors and Exceptions

```
>>> a = [1,2,3]
>>> a[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```


oops - maximum index of a is 2

```
>>> x = 1
>>> 4./ (x - 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

Errors and Exceptions

```
>>> float((1,2,3,4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: float() argument must be a
string or a number
```


oops - the float() builtin doesn't accept a tuple as its argument

```
>>> float('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to
float: hello
```


oops - the float() builtin accepts a string as an argument, but it must evaluate to a number

Exception Handling

Exceptions can be 'caught' and handled gracefully by a Python script. In fact, this is the Pythonic way of doing things:

EAFP

- it is Easier to Ask Forgiveness than Permission

This is what the `try ... except ... finally` clause is for:

```
try:
    <do something that might fail>
except (<exception1>, <exception2>, ...):
    <something went wrong: deal with it>
finally:
    <statements here are always executed>
```

Exception Handling

Valid, but not *Pythonic*:

```
if x != 0:
    rx = 1./x
    print('the reciprocal of x is', rx)
else:
    print('you can\'t divide by zero!')
```

The Pythonic way:

```
try:
    rx = 1./x
    print('the reciprocal of x is', rx)
except ZeroDivisionError:
    print('you can\'t divide by zero!')
```

Raising Exceptions

Your code can raise its own exceptions (either builtin exceptions such as `ValueError` or ones you define yourself).

```
det = a*d - b*c
if det == 0.:
    raise ValueError(
       'Oops: that matrix is not invertible')
```

Exception Handling

Another example: suppose an input file contains two columns of numbers, but also some blank lines and comments (which start with '#':

datafile.txt

```
0.5      1.23

1.5      2.46
# the next point is an outlier
2.5    200.34
3.5      3.77
4.5     12.5

# not sure about this point, either
55.0    -0.22
6.5      8.78
```

Exception Handling

readdata.py

```
x = []
y = []
fi = open('datafile.txt', 'r')
for line in fi.readlines():
    try:
        fields = line.split()
        this_x = float(fields[0])
        this_y = float(fields[1])
        x.append(this_x)
        y.append(this_y)
    except (IndexError, ValueError):
        pass
```

- Two sorts of `Exception` could be raised: the blank lines cause an `IndexError` and the comments a `ValueError`.
- The `pass` statement does nothing, so we return to the top of the loop; `continue` would have the same effect here.