FILE HANDLING AND EXCEPTIONS

INPUT

- We've already seen how to use the input function for grabbing input from a user:
- input()
  - Asks the user for a string of input, and returns the string.
  - If you provide an argument, it will be used as a prompt.
- raw_input() – Python 2
  - in Python 2, input() is available, but it will evaluate the expression.
  - Considered dangerous – try and avoid it.

```
>>> print(raw_input('What is your name? '))
What is your name? Spongebob
Spongebob
>>>
```

Note: reading an EOF will raise an EOFError.

FILES

Python includes a file object that we can use to manipulate files. There are two ways to create file objects.

- Use the file() constructor – Python 2
  - The second argument accepts a few special characters: 'r' for reading (default), 'w' for writing, 'a' for appending, 'r+' for reading and writing, 'b' for binary mode.

```
>>> f = file("filename.txt", 'r')
```

- Use the open() method- Python 3
  - The first argument is the filename, the second is the mode.

```
>>> f = open("filename.txt", 'rb')
```

Note: when a file operation fails, an `IOError` exception is raised.

FILE INPUT

- There are a few ways to grab input from a file.
- f.read()
  - Returns the entire contents of a file as a string.
  - Provide an argument to limit the number of characters you pick up.
- f.readline()
  - One by one, returns each line of a file as a string (ends with a newline).
  - End-of-file reached when return string is empty.
- Loop over the file object.
  - Most common, just use a for loop!

```
>>> f = open("somefile.txt",'r')
>>> f.read()
"Here's a line.\nHere's another line.\n"
>>> f.close()
>>> f = open("somefile.txt",'r')
>>> f.readline()
"Here's a line.\n"
>>> f.readline()
"Here's another line.\n"
>>> f.readline()
''
>>> f.close()
>>> f = open("somefile.txt",'r')
>>> for line in f:
...     print(line)
...
Here's a line.

Here's another line.
```

FILE INPUT

- Close the file with f.close()
  - Close it up and free up resources.

```
>>> f = open("somefile.txt", 'r')
>>> f.readline()
"Here's line in the file! \n"
>>> f.close()
```

- Another way to open and read:
  - No need to close, file objects automatically close when they go out of scope.

```
with open("text.txt", "r") as txt:
    for line in txt:
        print line
```

STANDARD FILE OBJECTS

- Just as C++ has cin, cout, and cerr, Python has standard file objects for input, output, and error in the sys module.
  - Treat them like a regular file object.

```
import sys
for line in sys.stdin:
    print line
```

- You can also receive command line arguments from sys.argv[ ].

```
for arg in sys.argv:
    print arg
```

```
$ python program.py here are some arguments
program.py
here
are
some
arguments
```

OUTPUT

- print()
  - Use the print() function to print to the user.
  - Use comma-separated arguments (separates with space) or concatenate strings.
  - Each argument will be evaluated and converted to a string for output.
  - print() has two optional keyword args, end and sep.

```
>>> print ('Hello,', 'World', 2018)
Hello, World 2018
>>> print ("Hello, " + "World " + "2018" )
Hello, World 2018
>>> for i in range(10):
...     print (i, end = '') # Do not include trailing
newline
...
0 1 2 3 4 5 6 7 8 9
```

PRINT FUNCTION

- ```
  print(*objects, sep=' ', end='\n', file=sys.stdout)
  ```
- Specify the separation string using the *sep* argument. This is the string printed between comma-separated objects.
- Specify the last string printed with the *end* argument.
- Specify the file object to which to print with the *file* argument.
  ```

PRINT FUNCTION

```
>>> print(555, 867, 5309, sep="-")
555-867-5309
>>> print("Winter", "is", "coming", end="...\n")
Winter is coming...
>>>
```

FILE OUTPUT

- f.write(*str*)
  - Writes the string argument *str* to the file object and returns None.
  - Make sure to pass strings, using the str() constructor if necessary.

```
>>> f = open("filename.txt", 'w')
>>> f.write("Heres a string that ends with " + str(2018))
```

- print >> f
  - Print to objects that implement write() (i.e. file objects).

```
f = open("filename.txt","w")
for i in range(1, 11):
    print >> f, "i is:", i
f.close()
```

MORE ON FILES

- File objects have additional built-in methods. Say I have the file object f:
- `f.tell()` gives current position in the file.
- `f.seek(offset[, from])` offsets the position by *offset* bytes from *from* position.
- `f.flush()` flushes the internal buffer.

Python looks for files in the current directory by default. You can also either provide the absolute path of the file or use the os.chdir() function to change the current working directory.

MODIFYING FILES AND DIRECTORIES

- Use the os module to perform some file-processing operations.
- `os.rename(`*`current_name`*`, `*`new_name`*`)` renames the file *current_name* to *new_name.*
- `os.remove(`*`filename`*`)` deletes an existing file named filename.
- `os.mkdir(`*`newdirname`*`)` creates a directory called *newdirname.*
- `os.chdir(`*`newcwd`*`)` changes the cwd to *newcwd.*
- `os.getcwd()` returns the current working directory.
- `os.rmdir(`*`dirname`*`)` deletes the empty directory *dirname.*

EXCEPTIONS

- Errors that are encountered during the execution of a Python program are *exceptions.*

```
>>> print (spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined

>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

There are a number of built-in exceptions, which are listed here.

HANDLING EXCEPTIONS

- Explicitly handling exceptions allows us to control otherwise undefined behavior in our program, as well as alert users to errors. Use try/except blocks to catch and recover from exceptions.

```
>>> while True:
...     try:
...         x = int(raw_input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...
Enter a number: two
Ooops !! That was not a valid number. Try again.
Enter a number: 100
```

HANDLING EXCEPTIONS

- First, the try block is executed. If there are no errors, except is skipped.
- If there are errors, the rest of the try block is skipped.
  - Proceeds to except block with the matching exception type.
- Execution proceeds as normal.

```
>>> while True:
...     try:
...         x = int(input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...
Enter a number: two
Ooops !! That was not a valid number. Try again.
Enter a number: 100
```

HANDLING EXCEPTIONS

- The try/except clause options are as follows:

Clause form	**Interpretation**
`except:`	Catch all (or all other) exception types
`except name:`	Catch a specific exception only
`except name as value:`	Catch the listed exception and its instance
`except (name1, name2):`	Catch any of the listed exceptions
`except (name1, name2) as value:`	Catch any of the listed exceptions and its instance
`else:`	Run if no exception is raised
`finally:`	Always perform this block

HANDLING EXCEPTIONS

- There are a number of ways to form a try/except block.

```
>>> while True:
...     try:
...         x = int(input("Enter a number: "))
...     except ValueError:
...         print("Ooops !! That was not a valid number. Try again.")
...     except (TypeError, IOError) as e:
...         print(e)
...     else:
...         print("No errors encountered!")
...     finally:
...         print("We may or may not have encountered errors…")
...
```

RAISING AN EXCEPTION

- Use the raise statement to force an exception to occur. Useful for diverting a program or for raising custom exceptions.

```
try:
    raise IndexError("Index out of range")
except IndexError as ie:
    print("Index Error occurred: ", ie)
```

Output:
Index Error occurred: Index out of range

CREATING AN EXCEPTION

- Make your own exception by creating a new exception class derived from the *Exception* class (we will be covering classes soon).

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print ('My exception occurred, value:', e)
...
My exception occurred, value: 4
```

ASSERTIONS

- Use the assert statement to test a condition and raise an error if the condition is false.

```
>>> assert a == 2
```

is equivalent to

```
>>> if not a == 2:
...     raise AssertionError()
```