FILES AND EXCEPTIONS

FILES

- To open a **file**, you specify its name and indicate whether you want to **read** or **write**

```
>>> f = open("test.txt","w")
>>> print f
<open file ’test.txt’, mode ’w’ at fe820>
```

  - First argument: Name of the file
  - Second argument: mode (“**r**” for Read/”**w**” for Write)

- If there is no file named **test.txt**, it will be created. If there already is one, it will be replaced by the file we are writing.

- If we try to open a file that doesn’t exist, we get an **error**:

```
>>> f = open("test.cat","r")
IOError: [Errno 2] No such file or directory: ’test.cat’
```

WRITING DATA

- To put data in the file we invoke the write method on the file object with **w**

```
>>> f = open("test.txt","w")
>>> f.write("Now is the time")
>>> f.write("to close the file")
```

- Closing the file tells the system that we are done writing and makes the file available for reading

```
>>> f.close()
```

READING DATA

- To read the data from the file we invoke the read method on the file object with **r**

```
>>> f = open("test.txt",“r")
>>> text = f.read()
>>> print text
Now is the timeto close the file
```

- Read can also take an argument that indicates how many characters to read

```
>>> f = open("test.txt","r")
>>> print f.read(5)
Now i
```

READING CONTINUED

- If not enough characters are left in the file, **read** returns the remaining characters. When we get to the **end of the file, read** returns the empty **string:**

```
>>> print f.read(1000006)
s the timeto close the file
>>> print f.read()

>>>
```

EXAMPLE 1

- The following function copies a **file,** reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file

```
def copyFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.read(50)
    if text == "":
      break
    f2.write(text)
  f1.close()
  f2.close()
  return
```

TEXT FILES

- A text file with three lines of text separated by newlines

```
>>> f = open("test.dat","w")
>>> f.write("line one\nline two\nline three\n")
>>> f.close()
```

- The **readline** method reads all the characters up to and including the next newline character:

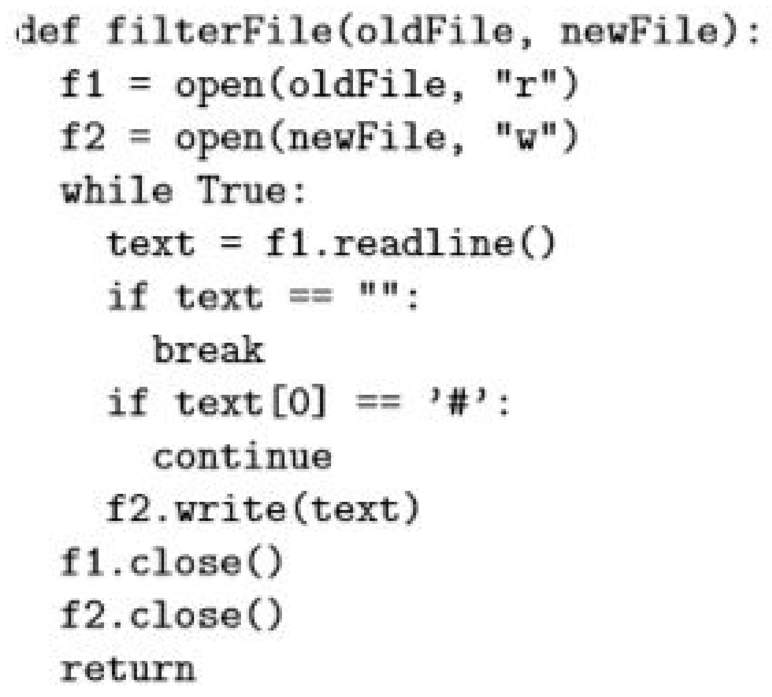
```
>>> f = open("test.dat","r")
>>> print f.readline()
line one

>>>
```

- **readlines** returns all of the remaining lines as a list of strings:

```
>>> print f.readlines()
['line two\012', 'line three\012']
```

EXAMPLE 2

- The following is an example of a line-processing program. filterFile makes a copy of oldFile, omitting any lines that begin with #:



```
def filterFile(oldFile, newFile):
  f1 = open(oldFile, "r")
  f2 = open(newFile, "w")
  while True:
    text = f1.readline()
    if text == "":
      break
    if text[0] == '#':
      continue
    f2.write(text)
  f1.close()
  f2.close()
  return
```

WRITING VARIABLES

- To write data types other than **string**, do

```
>>> x = 52
>>> f.write (str(x))
```

or use the **Format Operator**

```
>>> cars = 52
>>> "%d" % cars
'52'
```

Another example:

```
>>> cars = 52
>>> "In July we sold %d cars.“ % cars
'In July we sold 52 cars.'
```

WRITING VARIABLES – FORMAT OPERATOR

- **%f** for floats, **%s** for strings

```
>>>
```

- **"In %d days we made %f million %s." % (34,6.1,'dollars')**

```
'In 34 days we made 6.100000 million dollars.'

>>> "%d %d %d" % (1,2)
TypeError: not enough arguments for format string
>>> "%d" % 'dollars'
TypeError: illegal argument type for built-in operation
```

- For more control over the format of numbers, we can specify the number of digits as part of the format sequence:

```
>>> "%6d" % 62
'      62'
>>> "%12f" % 6.1
'      6.100000'
```

The number after the percent sign is the minimum number of spaces the number will take up.

WRITING VARIABLES – FORMAT OPERATOR

- If the value provided takes fewer digits, leading spaces are added. If the number of spaces is negative, trailing spaces are added:

```
>>> "%-6d" % 62
'62    '
```

- For floating-point numbers, we can also specify the number of digits after the decimal point:

```
>>> "%12.2f" % 6.1
'        6.10'
```

DIRECTORIES

- If you want to open a file somewhere else, you have to specify the path to the file, which is the name of the directory (or folder) where the file is located:

```
>>> f = open("/usr/share/dict/words","r")
>>> print f.readline()
Aarhus
```

- This example opens a file named words that resides in a directory named **dict**, which resides in share, which resides in **usr**, which resides in the top-level directory of the system, called /.

  You cannot use / as part of a filename; it is reserved as a delimiter between directory and filenames.

PICKLING

- Things are written as **strings** in files. To get the original data structures (like **lists**, **dictionaries**) back, use the concept of **Pickling**. To use it, import **pickle** and then open the file in the usual way:

```
>>> import pickle
>>> f = open("test.pck","w")
```

- To store a data structure, use the dump method and then close the file in the usual way:

```
>>> pickle.dump(12.3, f)
>>> pickle.dump([1,2,3], f)
>>> f.close()
```

PICKLING - CONTINUED

- Then we can open the file for reading and load the data structures we dumped:

```
>>> f = open("test.pck","r")
>>> x = pickle.load(f)
>>> x 12.3
>>> type(x)
<type 'float'>
>>> y = pickle.load(f)
>>> y [1, 2, 3]
>>> type(y)
<type 'list'>
```

EXCEPTIONS

- Whenever a runtime error occurs, it creates an exception. Usually, the program stops and Python prints an error message.

For example,

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

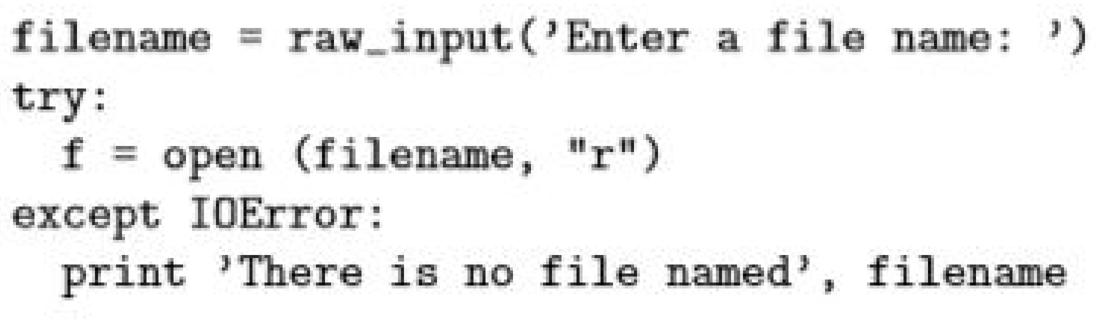
```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

```
>>> b = {}
>>> print b['what']
KeyError: what
```

```
>>> f = open("Idontexist", "r")
IOError: [Errno 2] No such file or directory: 'Idontexist'
```

EXCEPTIONS

- The error message has two parts: the type of error before the colon, and specifics about the error after the colon. Sometimes we want to execute an operation that could cause an exception, but we don't want the program to stop. We can handle the exception using the try and except statements.

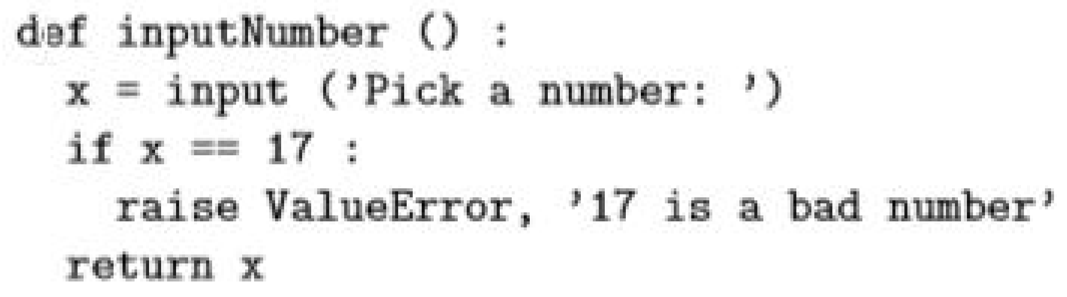


```
filename = raw_input('Enter a file name: ')
try:
  f = open (filename, "r")
except IOError:
  print 'There is no file named', filename
```

- The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If an exception of type IOError occurs, it executes the statements in the except branch and then continues.

EXCEPTIONS

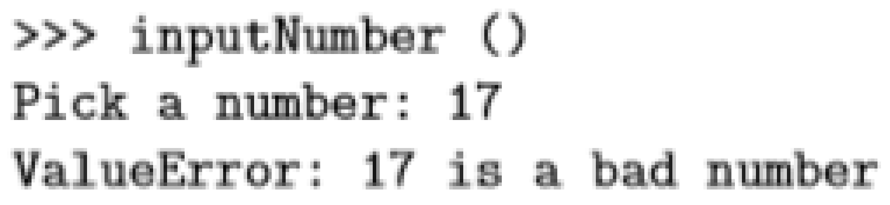
- You can use multiple except blocks to handle different kinds of exceptions.
- If your program detects an error condition, you can make it raise an exception. Here is an example that gets input from the user and checks for the value 17.



```
def inputNumber () :
  x = input (’Pick a number: ’)
  if x == 17 :
    raise ValueError, ’17 is a bad number’
  return x
```

EXCEPTIONS

- The raise statement takes two arguments: the exception type and specific information about the error. ValueError is one of the exception types Python provides for a variety of occasions. Other examples include TypeError, KeyError, and my favorite, NotImplementedError.
- If the function that called inputNumber handles the error, then the program can continue; otherwise, Python prints the error message and exits:



```
>>> inputNumber ()
Pick a number: 17
ValueError: 17 is a bad number
```