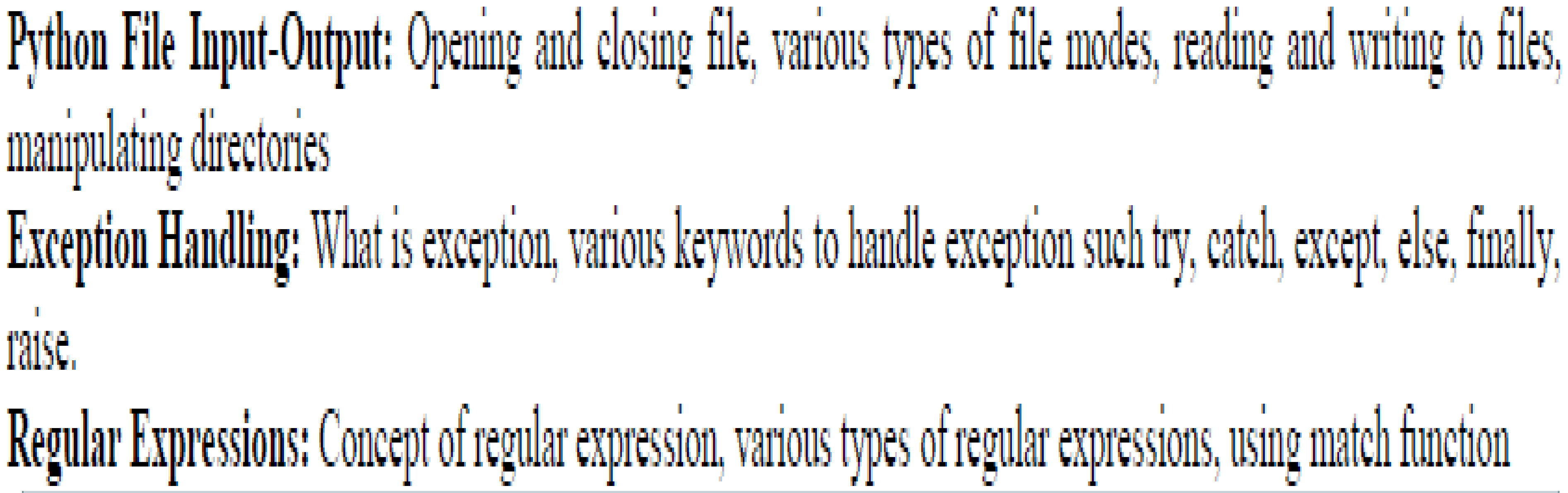
Python Programming (16ITE01)

UNIT - IV

T. PRATHIMA, DEPT. OF IT, CBIT

Syllabus



Python File Input-Output: Opening and closing file, various types of file modes, reading and writing to files, manipulating directories

Exception Handling: What is exception, various keywords to handle exception such try, catch, except, else, finally, raise.

Regular Expressions: Concept of regular expression, various types of regular expressions, using match function

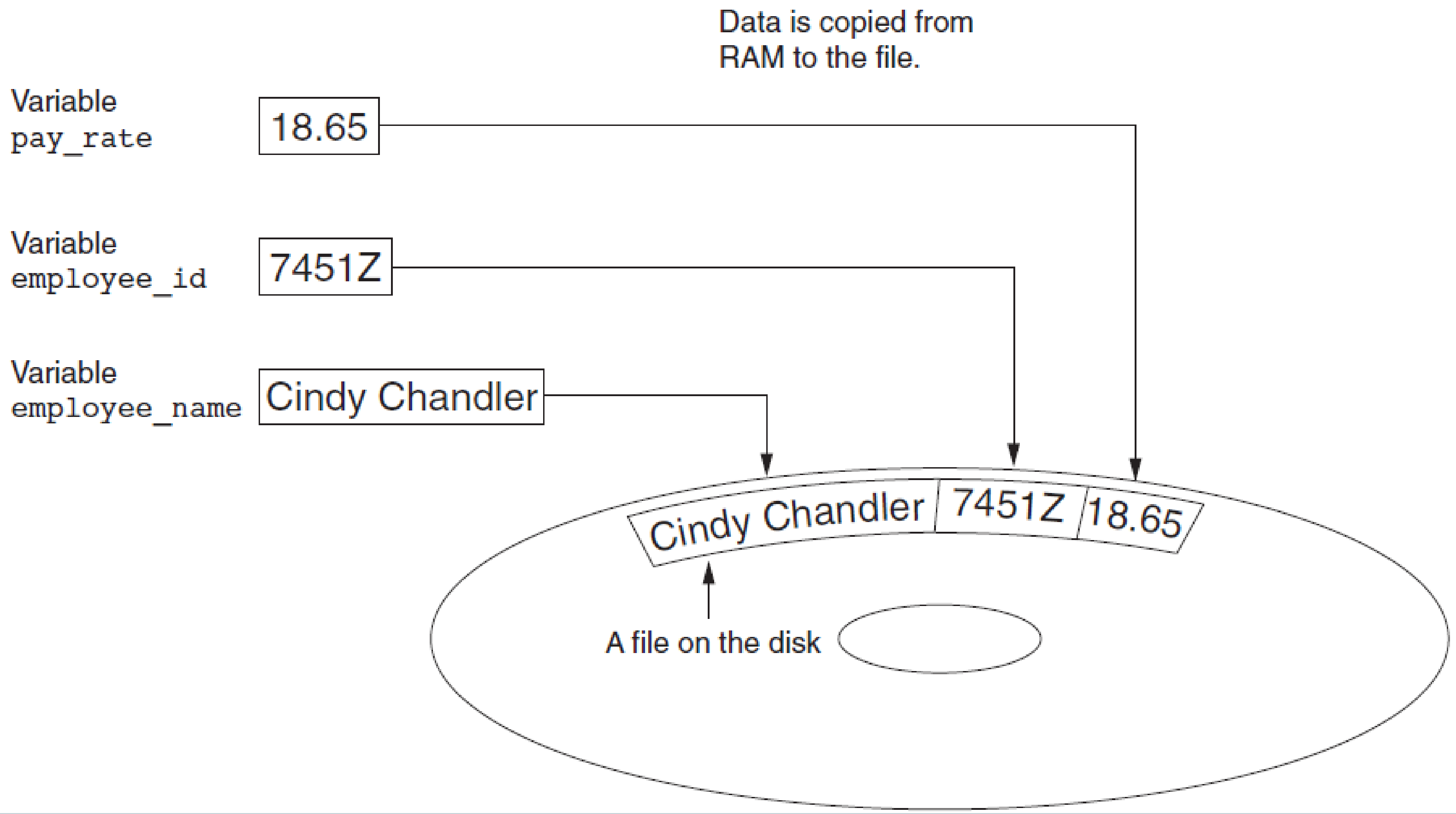
Topics

- Introduction to File Input and Output
- Using Loops to Process Files
- Processing Records
- Exceptions

Introduction to File Input and Output

- For program to retain data between the times it is run, you must save the data
    - Data is saved to a file, typically on computer disk
    - Saved data can be retrieved and used at a later time
- “Writing data to”: saving data on a file
- Output file: a file that data is written to

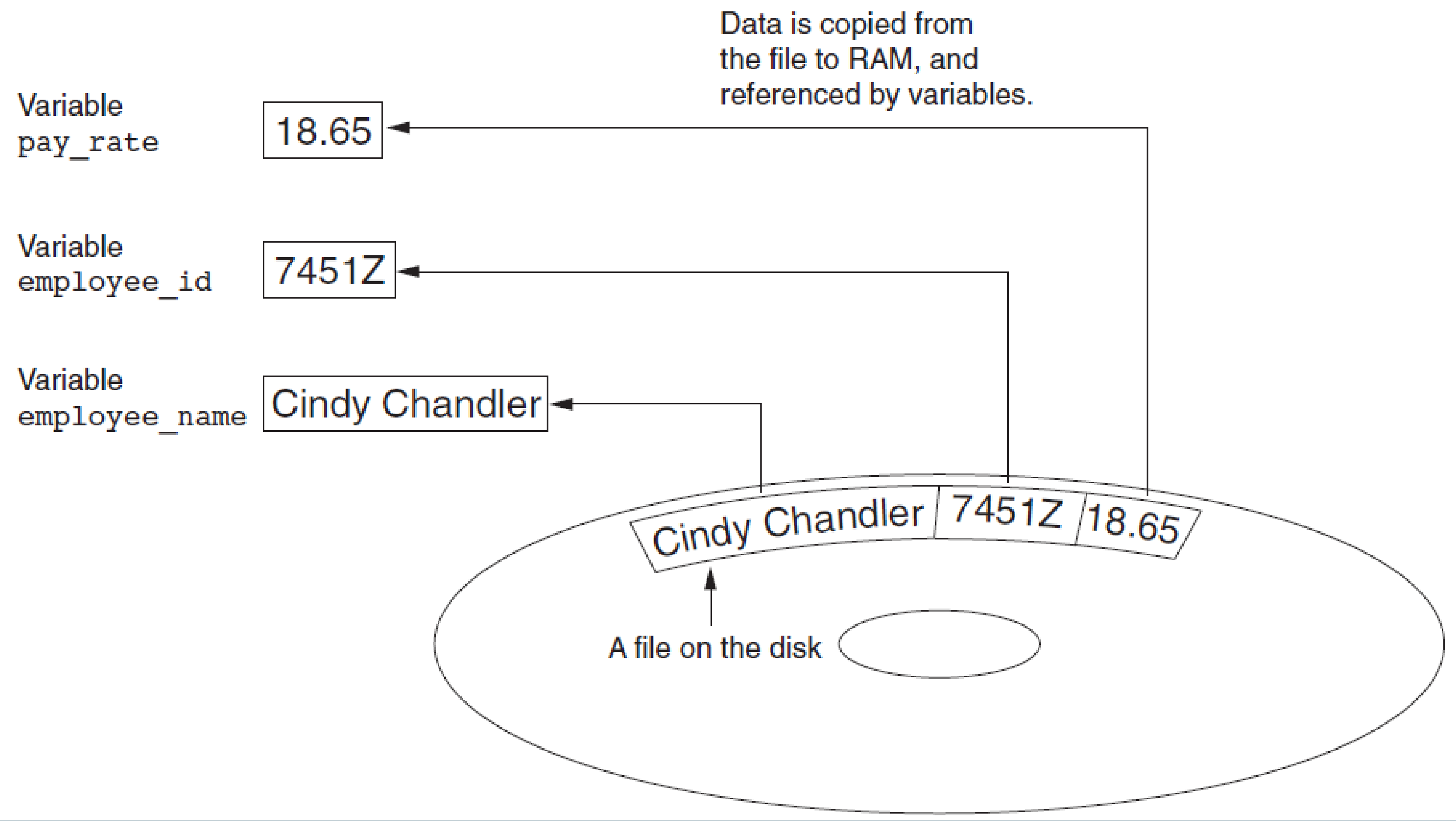
Figure 6-1 Writing data to a file



Introduction to File Input and Output (cont'd.)

- "<u>Reading data from</u>": process of retrieving data from a file
- <u>Input file</u>: a file from which data is read
- Three steps when a program uses a file
  - Open the file
  - Process the file
  - Close the file

Figure 6-2 Reading data from a file



Types of Files and File Access Methods

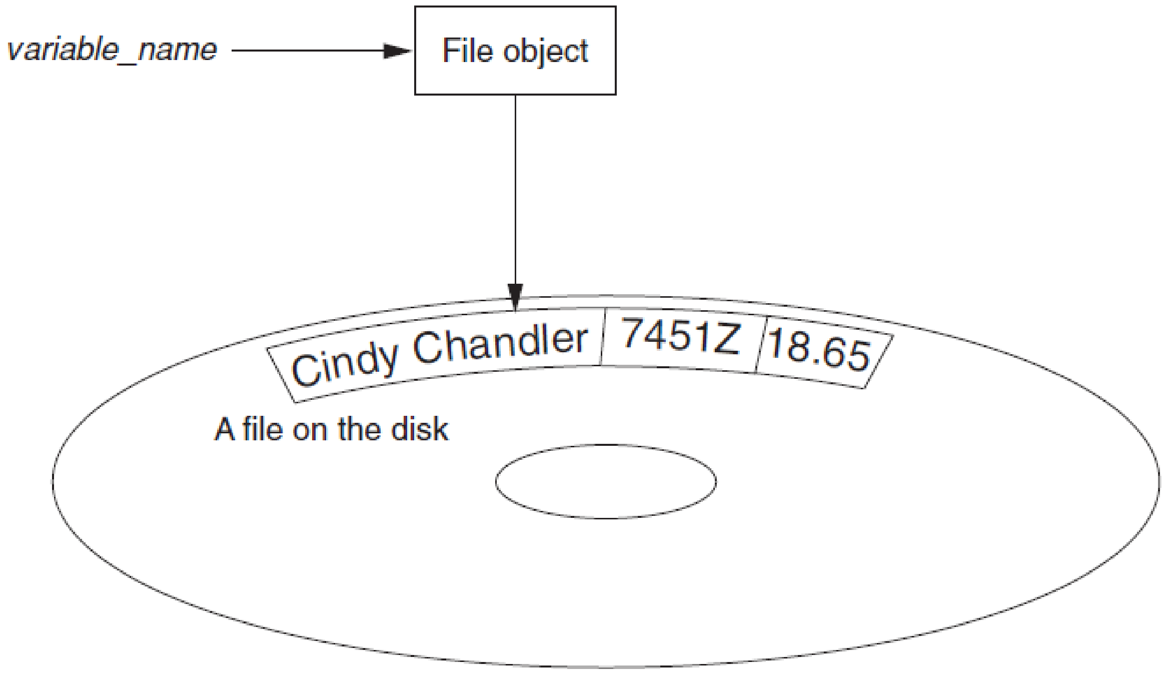
- In general, two types of files
  - Text file: contains data that has been encoded as text
  - Binary file: contains data that has not been converted to text
- Two ways to access data stored in file
  - Sequential access: file read sequentially from beginning to end, can't skip ahead
  - Direct access: can jump directly to any piece of data in the file

Filenames and File Objects

- Filename extensions: short sequences of characters that appear at the end of a filename preceded by a period
  - Extension indicates type of data stored in the file
- File object: object associated with a specific file
  - Provides a way for a program to work with the file: file object referenced by a variable

Filenames and File Objects (cont'd.)

Figure 6-4 A variable name references a file object that is associated with a file



Opening a File

- `open` function: used to open a file
  - Creates a file object and associates it with a file on the disk
  - General format:

    `file_object` `=` `open(`*`filename`*`,` *`mode`*`)`
- Mode: string specifying how the file will be opened
  - Example: reading only (`'r'`), writing (`'w'`), and appending (`'a'`)

Specifying the Location of a File

- If `open` function receives a filename that does not contain a path, assumes that file is in same directory as program
- If program is running and file is created, it is created in the same directory as the program
  - Can specify alternative path and file name in the `open` function argument
    - Prefix the path string literal with the letter `r`

Example:

```
test_file = open(r'C:\Users\Blake\temp\test.txt', 'w')
```

Writing Data to a File

- Method: a function that belongs to an object
  - Performs operations using that object
- File object's `write` method used to write data to the file
  - Format: `file_variable.write(string)`
- File should be closed using file object `close` method
  - Format: `file_variable.close()`

Reading Data From a File

- `read` method: file object method that reads entire file contents into memory
  - Only works if file has been opened for reading
  - Contents returned as a string
- `readline` method: file object method that reads a line from the file
  - Line returned as a string, including `'\n'`
- Read position: marks the location of the next item to be read from a file

Concatenating a Newline to and Stripping it From a String

- In most cases, data items written to a file are values referenced by variables
  - Usually necessary to concatenate a `'\n'` to data before writing it
    - Carried out using the + operator in the argument of the `write` method
- In many cases need to remove `'\n'` from string after it is read from a file
  - `rstrip` method: string method that strips specific characters from end of the string

Appending Data to an Existing File

- When open file with `'w'` mode, if the file already exists it is overwritten
- To append data to a file use the `'a'` mode
  - If file exists, it is not erased, and if it does not exist it is created
  - Data is written to the file at the end of the current contents

Writing and Reading Numeric Data

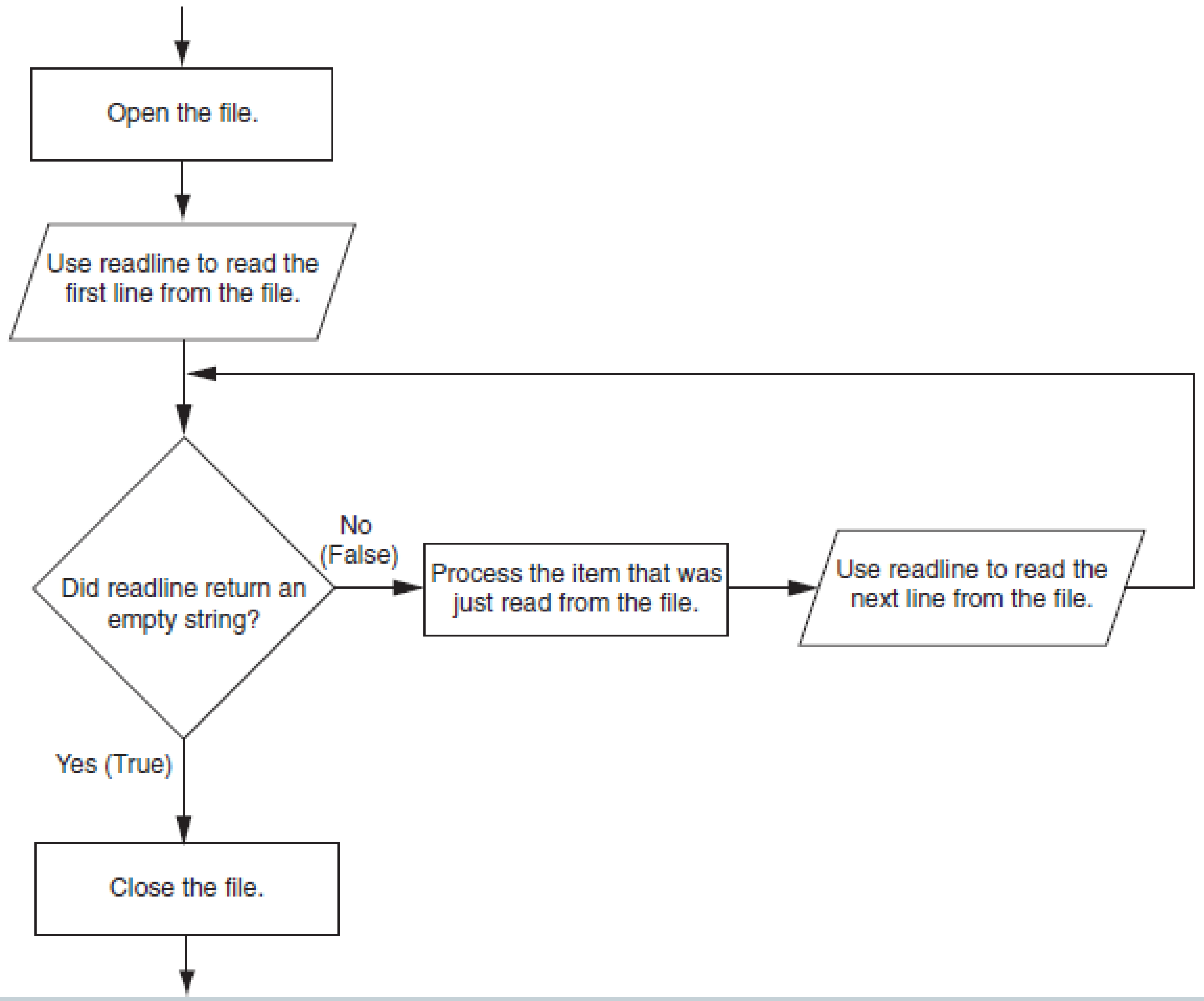
- Numbers must be converted to strings before they are written to a file
- `str` function: converts value to string
- Number are read from a text file as strings
  - Must be converted to numeric type in order to perform mathematical operations
  - Use `int` and `float` functions to convert string to numeric value

Using Loops to Process Files

- Files typically used to hold large amounts of data
  - Loop typically involved in reading from and writing to a file
- Often the number of items stored in file is unknown
  - The `readline` method uses an empty string as a sentinel when end of file is reached
    - Can write a while loop with the condition

      `while` *`line`* `!= ''`

Figure 6-17 General logic for detecting the end of a file



Open the file.
Use readline to read the first line from the file.
Did readline return an empty string?
No (False)
Process the item that was just read from the file.
Use readline to read the next line from the file.
Yes (True)
Close the file.

Using Python's `for` Loop to Read Lines

- Python allows the programmer to write a `for` loop that automatically reads lines in a file and stops when end of file is reached
  - Format:
    ```
    for line in file_object:
                statements
    ```
  - The loop iterates once over each line in the file

Processing Records

- Record: set of data that describes one item
- Field: single piece of data within a record
- Write record to sequential access file by writing the fields one after the other
- Read record from sequential access file by reading each field until record complete

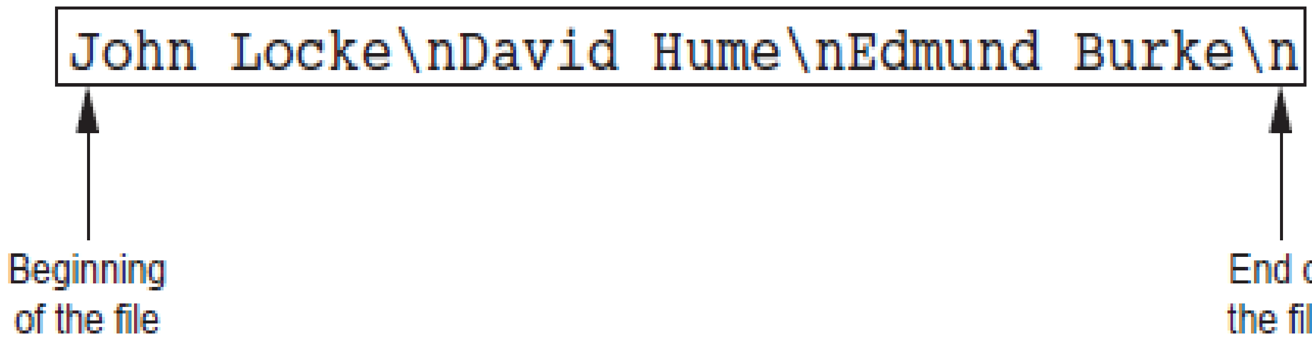
Processing Records (cont'd.)

- When working with records, it is also important to be able to:
  - Add records
  - Display records
  - Search for a specific record
  - Modify records
  - Delete records

Program 6-1 (file_write.py)

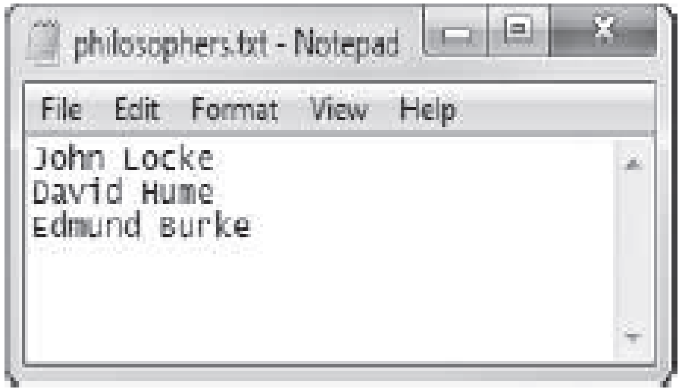
```python
# This program writes three lines of data
# to a file.
def main():
    # Open a file named philosophers.txt.
    outfile = open('philosophers.txt', 'w')

    # Write the names of three philosphers
    # to the file.
    outfile.write('John Locke\n')
    outfile.write('David Hume\n')
    outfile.write('Edmund Burke\n')

    # Close the file.
    outfile.close()

# Call the main function.
main()
```

Figure 6-5 Contents of the file `philosophers.txt`



Notice that each of the strings written to the file end with `\n`, which you will recall is the newline escape sequence. The `\n` not only separates the items that are in the file, but also causes each of them to appear in a separate line when viewed in a text editor. For example Figure 6-6 shows the `philosophers.txt` file as it appears in Notepad.

Figure 6-6 Contents of `philosophers.txt` in Notepad



```
John Locke
David Hume
Edmund Burke
```

```python
# This program reads and displays the contents
# of the philosophers.txt file.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read the file's contents.
    file contents = infile.read()
    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print(file_contents)

# Call the main function.
main()
```

Program Output

```
John Locke
David Hume
Edmund Burke
```

```
# This program gets three names from the user
# and writes them to a file.

def main():
    # Get three names.
    print('Enter the names of three friends.')
    name1 = input('Friend #1: ')
    name2 = input('Friend #2: ')
    name3 = input('Friend #3: ')

    # Open a file named friends.txt.
    myfile = open('friends.txt', 'w')

    # Write the names to the file.
    myfile.write(name1 + '\n')
    myfile.write(name2 + '\n')
    myfile.write(name3 + '\n')

    # Close the file.
    myfile.close()
    print('The names were written to friends.txt.')

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter the names of three friends.
Friend #1: Joe Enter
Friend #2: Rose Enter
Friend #3: Geri Enter
The names were written to friends.txt.
```

```python
# This program reads the contents of the
# philosophers.txt file one line at a time.
def main():
    # Open a file named philosophers.txt.
    infile = open('philosophers.txt', 'r')

    # Read three lines from the file.
    line1 = infile.readline()
    line2 = infile.readline()
    line3 = infile.readline()
    # Close the file.
    infile.close()

    # Print the data that was read into
    # memory.
    print(line1)
    print(line2)
    print(line3)

# Call the main function.
main()
```

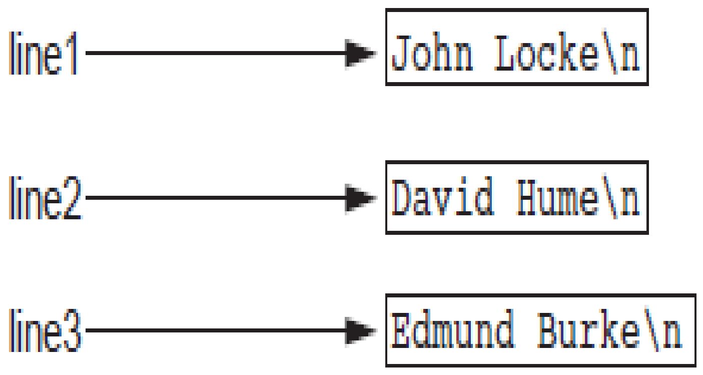
Program Output

```
John Locke

David Hume

Edmund Burke
```

Read position

- When a file is opened for reading, a special value known as a *read position is internally maintained for that file.*
- *A* file’s read position marks the location of the next item that will be read from the file.
- Initially, the read position is set to the beginning of the file

Figure 6-12 The strings referenced by the `line1`, `line2`, and `line3` variables



The statement in line 13 closes the file. The statements in lines 17 through 19 display the contents of the `line1`, `line2`, and `line3` variables.

NOTE: If the last line in a file is not terminated with a `\n`, the `readline` method will return the line without a `\n`.

Exceptions

- Exception: error that occurs while a program is running
  - Usually causes program to abruptly halt
- Traceback: error message that gives information regarding line numbers that caused the exception
  - Indicates the type of exception and brief description of the error that caused exception to be raised

Exceptions (cont'd.)

- Many exceptions can be prevented by careful coding
  - Example: input validation
  - Usually involve a simple decision construct
- Some exceptions cannot be avoided by careful coding
  - Examples
    - Trying to convert non-numeric string to an integer
    - Trying to open for reading a file that doesn't exist

Exceptions (cont'd.)

- Exception handler: code that responds when exceptions are raised and prevents program from crashing
  - In Python, written as `try/except` statement
    - General format:

      ```
      try:
          statements
      except exceptionName:
          statements
      ```

    - Try suite: statements that can potentially raise an exception
    - Handler: statements contained in `except` block

Exceptions (cont'd.)

- If statement in try suite raises exception:
  - Exception specified in except clause:
    - Handler immediately following except clause executes
    - Continue program after try/except statement
  - Other exceptions:
    - Program halts with traceback error message
- If no exception is raised, handlers are skipped

Handling Multiple Exceptions

- Often code in try suite can throw more than one type of exception
  - Need to write `except` clause for each type of exception that needs to be handled
- An `except` clause that does not list a specific exception will handle any exception that is raised in the try suite
  - Should always be last in a series of `except` clauses

Displaying an Exception's Default Error Message

- Exception object: object created in memory when an exception is thrown
  - Usually contains default error message pertaining to the exception
  - Can assign the exception object to a variable in an `except` clause
    - Example: `except ValueError as err:`
  - Can pass exception object variable to `print` function to display the default error message

The `else` Clause

- `try/except` statement may include an optional `else` clause, which appears after all the `except` clauses
  - Aligned with `try` and `except` clauses
  - Syntax similar to `else` clause in decision structure
  - Else suite: block of statements executed after statements in try suite, only if no exceptions were raised
    - If exception was raised, the else suite is skipped

The `finally` Clause

- `try/except` statement may include an optional `finally` clause, which appears after all the `except` clauses
  - Aligned with `try` and `except` clauses
  - General format: `finally:`

    ```
    finally:
        statements
    ```
  - Finally suite: block of statements after the `finally` clause
    - Execute whether an exception occurs or not
    - Purpose is to perform cleanup before exiting

What If an Exception Is Not Handled?

- Two ways for exception to go unhandled:
  - No except clause specifying exception of the right type
  - Exception raised outside a try suite
- In both cases, exception will cause the program to halt
  - Python documentation provides information about exceptions that can be raised by different functions

First the key word `try` appears, followed by a colon. Next, a code block appears which we will refer to as the *try suite*. The *try suite* is one or more statements that can potentially raise an exception.

After the try suite, an `except` *clause* appears. The `except` clause begins with the key word `except`, optionally followed by the name of an exception, and ending with a colon. Beginning on the next line is a block of statements that we will refer to as a *handler*.

When the `try/except` statement executes, the statements in the try suite begin to execute. The following describes what happens next:

- If a statement in the try suite raises an exception that is specified by the *`ExceptionName`* in an `except` clause, then the handler that immediately follows the `except` clause executes. Then, the program resumes execution with the statement immediately following the `try/except` statement.
- If a statement in the try suite raises an exception that is *not* specified by the `ExceptionName` in an `except` clause, then the program will halt with a traceback error message.
- If the statements in the try suite execute without raising an exception, then any `except` clauses and handlers in the statement are skipped and the program resumes execution with the statement immediately following the `try/except` statement.

```
# This program calculates gross pay.

def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))

        # Get the hourly pay rate.
        pay_rate = float(input('Enter your hourly pay rate: '))

        # Calculate the gross pay.
        gross_pay = hours * pay_rate

        # Display the gross pay.
        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
    except ValueError:
        print('ERROR: Hours worked and hourly pay rate must')

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
How many hours did you work? forty Enter
ERROR: Hours worked and hourly pay rate must
be valid numbers.
```

Figure 6-20 Handling an exception

If this statement raises a `ValueError` exception...

The program jumps to the `except ValueError` clause and executes its handler.

```
# This program calculates gross pay.

def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))

        # Get the hourly pay rate.
        pay_rate = float(input('Enter your hourly pay rate: '))

        # Calculate the gross pay.
        gross_pay = hours * pay_rate

        # Display the gross pay.
        print('Gross pay: $', format(gross_pay, ',.2f'), sep='')
    except ValueError:
        print('ERROR: Hours worked and hourly pay rate must')
        print('be valid integers.')

# Call the main function.
main()
```

```python
# This program displays the contents
# of a file.

def main():
    # Get the name of a file.
    filename = input('Enter a filename: ')
    try:
        # Open the file.
        infile = open(filename, 'r')

        # Read the file's contents.
        contents = infile.read()

        # Display the file's contents.
        print(contents)

        # Close the file.
        infile.close()
    except IOError:
        print('An error occurred trying to read')
        print('the file', filename)

# Call the main function.
main()
```

Program Output (with input shown in bold)

```
Enter a filename: bad_file.txt Enter
An error occurred trying to read the file bad_file.txt
```

```
# This program displays the total of the
# amounts in the sales_data.txt file.

def main():
    # Initialize an accumulator.
    total = 0.0

    try:
        # Open the sales_data.txt file.
        infile = open('sales_data.txt', 'r')

        # Read the values from the file and
        # accumulate them.
        for line in infile:
            amount = float(line)
            total += amount

        # Close the file.
        infile.close()

        # Print the total.
        print(format(total, ',.2f'))

    except IOError:
        print('An error occured trying to read the file.')

    except ValueError:
        print('Non-numeric data found in the file.')

    except:
        print('An error occured.')

# Call the main function.
main()
```

Regular Expressions

- Regular expressions are a powerful string manipulation tool
- All modern languages have similar library packages for regular expressions
- Use regular expressions to:
  - Search a string (`search` and `match`)
  - Replace parts of a string (`sub`)
  - Break strings into smaller pieces (`split`)

Python’s Regular Expression Syntax

- Most characters match themselves

  The regular expression “test” matches the string `‘test’`, and only that string

- [x] matches any *one* of a list of characters

  “[abc]” matches `‘a’`,`‘b’`,or `‘c’`

- [^x] matches any *one* character that is not included in *x*

  “[^abc]” matches any single character *except* `‘a’`,`’b’`,or `‘c’`

Python’s Regular Expression Syntax

- “.” matches any single character
- Parentheses can be used for grouping

  “(abc)+” matches `’abc’`, `‘abcabc’`, `‘abcabcabc’`, etc.

- *x*|*y* matches *x* or *y*

  “this|that” matches `‘this’ and ‘that’`, but not `‘thisthat’.`

Python’sRegular Expression Syntax

- *x** matches zero or more *x*’s

  “a*” matches `''`, `'a'`, `'aa'`, etc.

- *x*+ matches one or more *x*’s

  “a+” matches `'a'`,`'aa'`,`'aaa'`, etc.

- *x*? matches zero or one *x*’s

  “a?” matches `''` or `'a'`

- *x{m, n}* matches *i x*‘s, where $m \leq i \leq n$

  “a{2,3}” matches `'aa'` or `'aaa'`

Regular Expression Syntax

- “\d” matches any digit; “\D” any non-digit
- “\s” matches any whitespace character; “\S” any non-whitespace character
- “\w” matches any alphanumeric character; “\W” any non-alphanumeric character
- “^” matches the beginning of the string; “$” the end of the string
- “\b” matches a word boundary; “\B” matches a character that is not a word boundary

Search and Match

- The two basic functions are **re.search** and **re.match**
  - Search looks for a pattern anywhere in a string
  - Match looks for a match staring at the beginning
- Both return *None* (logical false) if the pattern isn't found and a "match object" instance if it is

```
>>> import re
>>> pat = "a*b"
>>> re.search(pat,"fooaaabcde")
<_sre.SRE_Match object at 0x809c0>
>>> re.match(pat,"fooaaabcde")
>>>
```

Q: What's a match object?

- A: an instance of the match class with the details of the match result

```
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group returns string matched
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```

What got matched?

- Here's a pattern to match simple email addresses

  \w+@(\w+\.)+(com|org|net|edu)

```
>>> pat1 = "\w+@(\w+\.)+(com|org|net|edu)"
>>> r1 = re.match(pat,"finin@cs.umbc.edu")
>>> r1.group()
'finin@cs.umbc.edu'
```

- We might want to extract the pattern parts, like the email name and host

What got matched?

- We can put parentheses around groups we want to be able to reference

```
>>> pat2 = "(\w+)@((\w+\.)+(com|org|net|edu))"
>>> r2 = re.match(pat2,"finin@cs.umbc.edu")
>>> r2.group(1)
'finin'
>>> r2.group(2)
'cs.umbc.edu'
>>> r2.groups()
r2.groups()
('finin', 'cs.umbc.edu', 'umbc.', 'edu’)
```

- Note that the ‘groups’ are numbered in a preorder traversal of the forest

What got matched?

- We can ‘label’ the groups as well…

```
>>> pat3
   ="(?P<name>\w+)@(?P<host>(\w+\.)+(com|org|net|edu))
   "

>>> r3 = re.match(pat3,"finin@cs.umbc.edu")

>>> r3.group('name')

'finin'

>>> r3.group('host')

'cs.umbc.edu’
```

- And reference the matching parts by the labels

More re functions

- re.split() is like split but can use patterns

```
>>> re.split("\W+", “This... is a test,
  short and sweet, of split().”)
['This', 'is', 'a', 'test', 'short’,
  'and', 'sweet', 'of', 'split’, ‘’]
```

- re.sub substitutes one string for a pattern

```
>>> re.sub('(blue|white|red)', 'black', 'blue socks and red
    shoes')
'black socks and black shoes’
```

- re.findall() finds al matches

```
>>> re.findall("\d+”,"12 dogs,11 cats, 1 egg")
['12', '11', ’1’]
```

Compiling regular expressions

- If you plan to use a re pattern more than once, compile it to a re object
- Python produces a special data structure that speeds up matching

```
>>> capt3 = re.compile(pat3)
>>> cpat3
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = cpat3.search("finin@cs.umbc.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'finin@cs.umbc.edu'
```

Pattern object methods

Pattern objects have methods that parallel the re functions (e.g., match, search, split, findall, sub), e.g.:

```
>>> p1 = re.compile("\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search(”Email steve@apple.com today.").group(0)
'steve@apple.com’
>>> p1.findall("Email steve@apple.com and bill@msft.com now.")
['steve@apple.com', 'bill@msft.com’]
>>> p2 = re.compile("[.?!]+\s+")
>>> p2.split("Tired? Go to bed!   Now!! ")
['Tired', 'Go to bed', 'Now', ’ ']
```

email address

sentence boundary

Example: pig latin

- Rules
  - If word starts with consonant(s)
    - Move them to the end, append “ay”
  - Else word starts with vowel(s)
    - Keep as is, but add “zay”
  - How might we do this?

The pattern

```
([bcdfghjklmnpqrstvwxyz]+)(\w+)
```

piglatin.py

```
import re
pat = ‘([bcdfghjklmnpqrstvwxyz]+)(\w+)’
cpat = re.compile(pat)

def piglatin(string):
  return " ".join( [piglatin1(w) for w in
   string.split()] )
```

piglatin.py

```
def piglatin1(word):
    """Returns the pig latin form of a word. e.g.:
     piglatin1("dog”) => "ogday". """
    match = cpat.match(word)
    if match:
        consonants = match.group(1)
        rest = match.group(2)
        return rest + consonents + “ay”
    else:
        return word + "zay"
```

Summary

- This chapter covered:
  - Types of files and file access methods
  - Filenames and file objects
  - Writing data to a file
  - Reading data from a file and determining when the end of the file is reached
  - Processing records
  - Exceptions, including:
    - Traceback messages
    - Handling exceptions

References

- Tony Gaddis, “Starting out with Python”, 3rd Edition, Global Edition, Pearson Education