Course Objectives

At the conclusion of this course, students will be able to:

- Execute Python code in a variety of environments
- Use correct Python syntax in Python programs
- Use the correct Python control flow construct
- Write Python programs using various collection data types
- Write home-grown Python functions
- Use many of the standard Python modules such as os, sys, math, and time
- Trap various errors via the Python Exception Handling model
- Use the IO model in Python to read and write disk files
- Create their own classes and use existing Python classes
- Understand and use the Object Oriented paradigm in Python programs
- Use the Python Regular Expression capabilities for data verification
# Table of Contents

## CHAPTER 1: AN INTRODUCTION TO PYTHON
- Introduction ................................................................................................... 1-2
- A Brief History of Python ............................................................................... 1-3
- Python Versions ............................................................................................ 1-4
- Installing Python ........................................................................................... 1-5
- Environment Variables .................................................................................. 1-6
- Executing Python from the Command Line .................................................. 1-7
- IDLE ............................................................................................................ 1-10
- Editing Python Files .................................................................................... 1-11
- Python Documentation ............................................................................... 1-12
- Getting Help ................................................................................................ 1-14
- Dynamic Types ........................................................................................... 1-16
- Python Reserved Words ............................................................................. 1-17
- Naming Conventions .................................................................................. 1-18

## CHAPTER 2: BASIC PYTHON SYNTAX
- Basic Syntax ................................................................................................. 2-2
- Comments .................................................................................................... 2-4
- String Values ................................................................................................ 2-5
- String Methods .............................................................................................. 2-6
- The `format` Method ................................................................................... 2-10
- String Operators ......................................................................................... 2-11
- Numeric Data Types ................................................................................... 2-13
- Conversion Functions ................................................................................. 2-14
- Simple Output ............................................................................................. 2-15
- Simple Input ................................................................................................ 2-16
- The `%` Method .......................................................................................... 2-18
- The `print` Function ................................................................................... 2-19

## CHAPTER 3: LANGUAGE COMPONENTS
- Indenting Requirements ................................................................................ 3-2
- The `if` Statement ......................................................................................... 3-3
- Relational and Logical Operators ................................................................. 3-4
- Bit Wise Operators ....................................................................................... 3-6
- The `while` Loop ......................................................................................... 3-7
- break and continue ....................................................................................... 3-10
- The `for` Loop ............................................................................................ 3-11

## CHAPTER 4: COLLECTIONS
- Introduction ................................................................................................. 4-2
- Lists .............................................................................................................. 4-3
- Tuples .......................................................................................................... 4-6
- Sets .............................................................................................................. 4-7
- Dictionaries .................................................................................................. 4-10
- Sorting Dictionaries .................................................................................... 4-14
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Topics</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Functions</td>
</tr>
<tr>
<td>6</td>
<td>Modules</td>
</tr>
<tr>
<td>7</td>
<td>Exceptions</td>
</tr>
<tr>
<td>8</td>
<td>Input and Output</td>
</tr>
</tbody>
</table>

### Chapter 5: Functions
- Introduction
- Defining Your Own Functions
- Parameters
- Function Documentation
- Keyword and Optional Parameters
- Passing Collections to a Function
- Variable Number of Arguments
- Scope
- Functions - "First Class Citizens"
- Passing Functions to a Function
- map
- filter
- Mapping Functions in a Dictionary
- Lambda
- Inner Functions
- Closures

### Chapter 6: Modules
- Modules
- Standard Modules - sys
- Standard Modules - math
- Standard Modules - time
- The dir Function

### Chapter 7: Exceptions
- Errors
- Runtime Errors
- The Exception Model
- Exception Hierarchy
- Handling Multiple Exceptions
- raise
- assert

### Chapter 8: Input and Output
- Introduction
- Data Streams
- Creating Your Own Data Streams
- Access Modes
- Writing Data to a File
- Reading Data From a File
- Additional File Methods
- Using Pipes as Data Streams
- Handling IO Exceptions
- Working with Directories
### Metadata

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The pickle Module</td>
<td>8-23</td>
</tr>
</tbody>
</table>

### CHAPTER 9: CLASSES IN PYTHON

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes in Python</td>
<td>9-2</td>
</tr>
<tr>
<td>Principles of Object Orientation</td>
<td>9-3</td>
</tr>
<tr>
<td>Creating Classes</td>
<td>9-4</td>
</tr>
<tr>
<td>Instance Methods</td>
<td>9-7</td>
</tr>
<tr>
<td>File Organization</td>
<td>9-8</td>
</tr>
<tr>
<td>Special Methods</td>
<td>9-9</td>
</tr>
<tr>
<td>Class Variables</td>
<td>9-13</td>
</tr>
<tr>
<td>Inheritance</td>
<td>9-14</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>9-17</td>
</tr>
<tr>
<td>Type Identification</td>
<td>9-22</td>
</tr>
<tr>
<td>Custom Exception Classes</td>
<td>9-23</td>
</tr>
</tbody>
</table>

### CHAPTER 10: REGULAR EXPRESSIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>10-2</td>
</tr>
<tr>
<td>Simple Character Matches</td>
<td>10-4</td>
</tr>
<tr>
<td>Special Characters</td>
<td>10-6</td>
</tr>
<tr>
<td>Character Classes</td>
<td>10-7</td>
</tr>
<tr>
<td>Quantifiers</td>
<td>10-9</td>
</tr>
<tr>
<td>The Dot Character</td>
<td>10-10</td>
</tr>
<tr>
<td>Greedy Matches</td>
<td>10-11</td>
</tr>
<tr>
<td>Grouping</td>
<td>10-12</td>
</tr>
<tr>
<td>Matching at Beginning or End</td>
<td>10-14</td>
</tr>
<tr>
<td>Match Objects</td>
<td>10-15</td>
</tr>
<tr>
<td>Substituting</td>
<td>10-17</td>
</tr>
<tr>
<td>Splitting a String</td>
<td>10-18</td>
</tr>
<tr>
<td>Compiling Regular Expressions</td>
<td>10-19</td>
</tr>
<tr>
<td>Flags</td>
<td>10-20</td>
</tr>
</tbody>
</table>
# Chapter 1:
## An Introduction to Python

1) Introduction .............................................................................................................. 1-2
2) A Brief History of Python ........................................................................................ 1-3
3) Python Versions ....................................................................................................... 1-4
4) Installing Python ...................................................................................................... 1-5
5) Environment Variables ........................................................................................... 1-6
6) Executing Python from the Command Line .......................................................... 1-7
7) IDLE ........................................................................................................................ 1-10
8) Editing Python Files ............................................................................................... 1-11
9) Python Documentation .......................................................................................... 1-12
10) Getting Help ......................................................................................................... 1-14
11) Dynamic Types .................................................................................................... 1-16
12) Python Reserved Words ....................................................................................... 1-17
13) Naming Conventions ............................................................................................ 1-18
Introduction

• In this section, we will provide a high-level overview of the environment in which Python programs are created and executed.

• By the end of this chapter, you will understand the basic philosophy of and gain an appreciation for the capabilities of the Python language.
  ▸ You will learn various ways of executing a Python application and learn how to navigate the Python help system.

• There are many ways of classifying programming languages.
  ▸ In the case of Python, it can be described as:
    • interpreted as opposed to compiled;
    • object oriented as opposed to procedure oriented; and
    • dynamically typed, as opposed to statically typed.

• Some of Python's strengths include the following.
  ▸ It is easy to learn.
  ▸ It is efficient at processing text data.
  ▸ It is easily extensible via Python Modules.

• Like most modern languages, Python facilitates modular programming and code reuse.
  ▸ Python also supports object oriented programming.
  ▸ Python is often compared to Perl and Ruby, two other scripting languages.
A Brief History of Python

- Python was created by Guido Van Rossum in 1990 and released to the public domain in 1991.

- In 1994, `comp.lang.python` was formed.
  - This is a dedicated Usenet newsgroup for general discussions and questions about Python.

- In 1996, the first edition of O'Reilly’s *Programming Python* was released.

- Python was originally developed to aid in the creation of test scripts.
  - However, it quickly became a widely used, general-purpose programming language.

- Python runs on Windows, Linux/Unix, and Mac OS X, and has been ported to the Java and .NET virtual machines.
  - Python is also available in source code format to allow it to be ported to any operating system that supplies an ANSI-compliant C compiler.
Python Versions

• At the time of this writing, Python version 3.3 has been released.

• Python 3 includes significant changes to the language that make it incompatible with Python 2.
  - Some of the changes are listed below.
    - Print function
    - Bytes vs. strings
    - Iterators vs. list

• Some of the less disruptive improvements in Python 3 have been backported to versions 2.6 and 2.7.
  - Therefore, Python 2.x will correctly interpret Python 3 code in some, but not all, cases.
    - As we proceed through this course, we will point out various differences in the two versions of the language.
Installing Python

• Various versions of Python can be downloaded at http://www.python.org/download for each of the following operating systems, among others.
  - Linux
  - Windows
  - Mac

• A Microsoft Windows user can download a standard Windows MSI installer to install Python on their machine.

• Most Linux distributions come with Python pre-installed.
  - However, the pre-installed version is often an old version.
  - Several links provide access to the Python source code, as well as instructions for downloading, compiling, and installing.

• Python comes pre-installed on the Mac OS X operating system.
  - However, similar to Linux distributions, it is often an old version.
  - Several links provide access to the Python source code, as well as a Mac .dmg file that can be used to install Python on a Mac computer.

• For this course, your machines have a working version of Python 3.x, ready for use.
Environment Variables

Once Python is installed, the following environment variables can make working with Python files easier.

- The PATH environment variable should include the directory where the Python executable resides.
- The PYTHONPATH variable can be set to the directories where Python searches for the names of modules or libraries.
- The PYTHONSTARTUP variable stores the name of a Python script to execute when Python is started in interactive mode.
  - This script can be used to customize certain modes in which Python is executed.
Executing Python from the Command Line

There are various ways of executing Python code.

- From the command line
- From the Graphical Tool named IDLE
- From an Integrated Development Environment (IDE)

To execute a Python program from the command line, do the following.

- First create a `.py` file with your favorite text editor.
  - Note that the industry standard file extension for files containing Python code is `.py`.
- Once this file has been created, you can execute the file using the `python` command.

On a Linux system, the `python` command may refer to some version of Python 2 or some version of Python 3.

- The following convention has been recommended to ensure that Python scripts can continue to be portable across Linux systems, regardless of the version of the Python interpreter.
  - `python2` will refer to some version of Python 2.
  - `python3` will refer to some version of Python 3.
  - `python` may refer to either version of Python for a given Linux distribution.
- The recommendation comes in the form of a Python Enhancement Proposal (PEP).
  - The specific PEP for the above recommendation is PEP 394.
  - More information about PEPs can be found at the following URL.
    http://www.python.org/dev/peps/
Executing Python from the Command Line

- All examples in this course will rely on the python3 command to execute the Python code.
  - All code for this course will include the following shebang as the first line of each source file.
    ```
    #!/usr/bin/env python3
    ```
  - The simple example script is shown below.

```
hello.py

1. #!/usr/bin/env python3
2. print("Hello World")
```

- The script would then be executed as follows.

```
student@localhost:~/pythonlabs/examples/1$ python3 hello.py
Hello World
$
```

- A more Linux-like approach is to first make the file executable and simply execute it.

```
student@localhost:~/pythonlabs/examples/1$ chmod 755 hello.py
$ hello.py
Hello World
$
```

- A variation on the above is shown below.
  - It uses a+x instead of 755
  - It also uses ./hello.py instead of hello.py if the current directory (.) is not on the PATH environment variable.

```
student@localhost:~/pythonlabs/examples/1$ chmod a+x hello.py
$ ./hello.py
Hello World
$
```
Executing Python from the Command Line

• This first program is offered merely to demonstrate the execution of a Python program from the command line.
  
  ▶ The `print` function sends data to the standard output.
    
    ▶ In Python 3, parentheses are required for function arguments.
    ▶ This is not the case in previous Python versions.

• You can also open up a Python shell.
  
  ▶ The Python shell is an interactive Python environment where you can enter Python commands and have them executed automatically.

  ▶ Here is a small example session.

```
student@localhost:~/pythonlabs/examples/1 $ python3
Python 3.3.0 (default, Apr 2 2013, 14:03:44)
[GCC 4.5.1 20100924 (Red Hat 4.5.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> print('Hello World')
Hello World
>>> 2 + 2
4
>>> 2 ** 10
1024
>>> x = 5
>>> x = x * 5
>>> print(x)
25
>>> exit()

$```

IDLE

• Python also provides a graphical version of the Python Command Line called IDLE.
  
  ▶ A Linux environment might require an additional installation of the python-tools package if Python was not installed from source.
  
  ▶ As with the python3 command discussed earlier, the IDLE command for Python 3 should be idle3.

• IDLE may be used as a simple editor and debugger for Python applications.

• You can start IDLE from the command line, as shown.

  student@localhost:~/pythonlabs/examples/1
  $ idle3
  $
  
  ▶ You will be presented with the following window and interactive environment.
Editing Python Files

• The choice of an editor for editing Python files is largely the personal choice of the developer.
  
  ▶ A long list of Python editors can be found at the following URL.
    
    http://wiki.python.org/moin/PythonEditors

• As we will see later, Python relies heavily on indentation of the source code to define blocks of code.
  
  ▶ This is in sharp contrast to the more traditional approach of using keywords or curly braces.
    
    • Since there are many different Integrated Development Environments available for Python, you might consider using one of them for this course.
    
    • Python IDEs can handle the spacing issues for you.

• For those developers of you who do not necessarily have a favorite editor, you may find one of the following useful.
  
  ▶ A simple graphical editor, named gedit, is available.

  ▶ A lightweight IDE, named Geany, has also been made available for this course.

  • Launching each program can be done from the applications menu as shown below.
Python Documentation

● There are various ways in which the Python programmer can get help.

  ▶ A good starting point is the following URL.

    http://www.python.org/doc/versions/

● From there, you can navigate to the specific version of Python being used.

  ▶ The documentation page for version 3.3.0 is shown on the following page.
The "Library Reference" and "Language Reference" links above are often useful to both new and seasoned Python developers.

The "Python Setup and Usage" provides additional information for using Python on a specific operating system.
Getting Help

- In addition to the online Python documentation, you can also use the Python shell to get additional help.
  - Recall the Python shell can be started with either of the following.
    - The `python3` command for a text based environment
    - The `idle3` command for a graphical environment
  - Once the Python shell is available, typing `help()` will get you to the Python help utility.
    - If you wish to see the math functions, type `math`.
    - If you wish to see the string functions, type `str`.
    - If you wish to see all documented topics, type `topics`.

- Here is an example of using the help utility.

```
$ python3
Python 3.3.0 (default, Apr 2 2013, 14:03:44)
[GCC 4.5.1 20100924 (Red Hat 4.5.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> help()
Welcome to Python 3.3! This is the interactive help utility.
If this is your first time using Python, you should definitely check out the tutorial on the Internet at http://docs.python.org/3.3/tutorial/.
Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".
To get a list of available modules, keywords, or topics, type "modules", "keywords", or "topics". Each module also comes with a one-line summary of what it does; to list the modules whose summaries contain a given word such as "spam", type "modules spam".

help> math
```

- Upon typing `math` at the `help>` prompt, the documentation will be displayed as shown on the following page.
Getting Help

$ Help on module math:

NAME
    math

MODULE REFERENCE
    http://docs.python.org/3.3/library/math

The following documentation is automatically generated from the Python source files. It may be incomplete, incorrect or include features that are considered implementation detail and may vary between Python implementations. When in doubt, consult the module reference at the location listed above.

DESCRIPTION
    This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
    acos(x)

        Return the arc cosine (measured in radians) of x.

• The up and down arrow keys on the keyboard can be used to scroll through the help screen shown above.
  • Linux users might recognize the above view as "man page."

• Typing the letter q will quit out of the help screen above.

• Typing quit at the help> prompt will exit the help utility and bring you back to the interactive Python shell prompt >>>
  • From there, typing quit() will exit the Python shell.

• Alternatively, at the interactive Python prompt >>>>, help can be obtained by passing information to the help method as shown below.

  • help('math'), help('str'), etc.
Dynamic Types

• Now that you know a little about Python and the Python environment, we will demonstrate a small program to emphasize the dynamic type system of Python.

```python
# datatypes.py
1. #!/usr/bin/env python3
2. x = 10
3. print (type(x))
4. x = 25.7
5. print (type(x))
6. x = "Hello"
7. print (type(x))
```

• Notice that the `type` function returns the type of what is referred to by a particular variable.

  ▶ Notice further that the type of the variable is bound dynamically as the program is executed.

  ▶ This is different from languages such as C, where the type of a variable is statically bound at compile time.
Python Reserved Words

• Several words we have seen are used for special purposes in the Python language.
  
  ▸ These words cannot be used as the names of variables or functions.

• We present them here so that you will know early on which ones they are.

<table>
<thead>
<tr>
<th>Python Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>True</td>
</tr>
<tr>
<td>and</td>
</tr>
<tr>
<td>as</td>
</tr>
<tr>
<td>assert</td>
</tr>
<tr>
<td>break</td>
</tr>
<tr>
<td>class</td>
</tr>
<tr>
<td>continue</td>
</tr>
</tbody>
</table>

• We have only seen a few of these keywords, but as we proceed through the course, we will use all of them.
  
  ▸ Each will be explained when introduced.

• The list above can be generated by executing:
  
  ▸ `help('keywords')` at the >>> prompt; or
  
  ▸ `keywords` at the help prompt.
Naming Conventions

- An identifier must start with either a letter of the alphabet or the underlining (_) character.
  - This can be followed by any number of letters and/or digits and/or the _ character.
  - Identifier names cannot consist of blanks, or punctuation symbols.

- There are a lot of different naming styles within the Python language.
  - PEP 8 is designed to give coding conventions for Python code.
  - A complete list of naming conventions can be found within PEP 8 at the following URL:
    www.python.org/dev/peps/pep-0008/#naming-conventions
Exercises

1. If you have not already done so, run the `python3` command to open a Python Shell at the command line and experiment with some Python expressions.

2. If you have not already done so, start the `idle3` command and experiment some more with some Python expressions.

3. Create a file named `first.py`.
   - In that file, assign values to variables and then perform a few operations with them.
   - Print the values of those variables.

4. Test the `PYTHONSTARTUP` environment variable by doing the following.
   - Start the Python shell and try to execute the following.
     ```
     >>> square(5)
     • Exit the Python shell by typing `exit()`
     ```
   - Now, create a file called `startup.py` in your home directory.
     - Place the following lines in `startup.py`:
       ```
       def square(p):
           return p * p
       ```
     - Set the `PYTHONSTARTUP` variable to point to your `startup.py` file.
     - If on a Linux system, be sure to `export` this variable.
   - Start the Python shell again and execute the following.
     ```
     >>> square(5)
     • Now `exit()` from the Python shell.
Chapter 8:
Input and Output

1) Introduction.............................................................................................................. 8-2
2) Data Streams ............................................................................................................ 8-3
3) Creating Your Own Data Streams ......................................................................... 8-4
4) Access Modes ........................................................................................................ 8-5
5) Writing Data to a File ............................................................................................ 8-6
6) Reading Data From a File ....................................................................................... 8-9
7) Additional File Methods ......................................................................................... 8-13
8) Using Pipes as Data Streams ............................................................................... 8-16
9) Handling IO Exceptions ......................................................................................... 8-18
10) Working with Directories .................................................................................... 8-19
11) Metadata ............................................................................................................... 8-20
12) The pickle Module ............................................................................................... 8-23
Introduction

• This section discusses various ways to perform input and output in Python.
  
  ▶ Up to this point in the course, we have limited our discussion of input and output to the following functions.
  
    • input
    • print
  
• This chapter discusses the various ways in which Python programs can read and write to and from various sources other than the standard input and output files.
  
  ▶ For example, you may want your program to read and write the following.
  
    • Disk files on your local disk
    • To and from processes
    • Binary files
Data Streams

• An input data stream is an object that provides data to the application as a sequence of bytes.
  
  - `sys.stdin` is the data stream that represents the operating system's standard input device - usually the keyboard.

• An output data stream is an object used by an application for writing data out as a sequence of bytes.
  
  - `sys.stdout` is the data stream that represents the standard output device - usually the system console.
  - `sys.stderr` is the data stream that represents the standard error device - also usually the system console.

• Although `stdin`, `stdout`, and `stderr` are opened by Python automatically, the `sys` module must be imported to access them directly.

    `sys_output.py`

    ```python
    1. #!/usr/bin/env python3
    2. import sys
    3. 
    4. sys.stdout.write("Standard Output\n")
    5. sys.stderr.write("Error Output\n")
    ```

  - The output of the above program is shown below.

    ```
    student@localhost:~/pythonlabs/examples/8
    $ python3 sys_output.py
    Standard Output
    Error Output
    $
    ```
Creating Your Own Data Streams

- A data stream must be declared as an input data stream or an output data stream at the time the stream is opened.
  - The `open()` function opens a file and returns a data stream object.
    - The data stream object may be used to read from the file or write to the file, depending upon how the file is opened.
    - If the `open()` function fails, a subclass of `OSError` is raised.

- An example of using the `open()` function is shown below.
  ```python
  f = open("datafile", "r")
  ```
  - When a file is opened as shown above, the file is assumed to be in the current directory.
  - Full path information can be used as well.
    ```python
    f = open("../datafile", "r")   # Linux
    f = open("..\datafile", "r")  # Windows
    ```
  - The `open` function has the following two parameters.
    - A file name
    - An access mode
Access Modes

The following table describes the access modes.

<table>
<thead>
<tr>
<th>Mode</th>
<th>File Opened for:</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>• Reading</td>
</tr>
<tr>
<td></td>
<td>• Assumes file already exists</td>
</tr>
<tr>
<td>w</td>
<td>• Writing</td>
</tr>
<tr>
<td></td>
<td>• Assumes file does not exist</td>
</tr>
<tr>
<td></td>
<td>• If it does exist, file is truncated</td>
</tr>
<tr>
<td>a</td>
<td>• Appending</td>
</tr>
<tr>
<td></td>
<td>• If file does not exist, it is created</td>
</tr>
<tr>
<td>b</td>
<td>• Binary</td>
</tr>
<tr>
<td>r+</td>
<td>• Read and Write</td>
</tr>
<tr>
<td></td>
<td>• Assumes file already exists</td>
</tr>
<tr>
<td>w+</td>
<td>• Read and Write</td>
</tr>
<tr>
<td></td>
<td>• Assumes the file does not exist</td>
</tr>
<tr>
<td>a+</td>
<td>• Read and Write at the end of the file</td>
</tr>
<tr>
<td></td>
<td>• If file does not exist, it is created</td>
</tr>
</tbody>
</table>

Once the stream is opened, you can use various methods to read and write files.

First, we will look at how you can write data to a disk file.
Writing Data to a File

- Here is a simple program that writes data to a file.
  - When the program is run, no output appears on the display, since it has been routed to the file named `output`.

```python
write1.py
1. #!/usr/bin/env python3
2. f = open('output', 'w')
3. f.write('This is a test.
')
4. f.write('This is another test.
')
5. f.close()
```

- The program above uses the `open` function to create a stream.
- Then, the `write` function places data on the stream.
  - Each `write` places data sequentially on the stream.
- Finally, the `close` function closes the stream.
  - It is always a good idea to close a stream you opened after you are finished processing it.

- In the above program, the file name is hard coded into the program.
  - To add robustness to the program, you could just as easily have given the file name as a command line argument.

```python
write2.py
1. #!/usr/bin/env python3
2. import sys
3.
4. f = open(sys.argv[1], 'w')
5. f.write('This is a test.
')
6. f.write('This is another test.
')
7. f.close()
```
Writing Data to a File

• Alternatively, you could get the file name interactively from the user of the program.

write3.py

```python
#!/usr/bin/env python3
filename = input("Enter file name: ")
f = open(filename, 'w')
f.write('This is a test.\n')
f.write('This is another test.\n')
f.close()
```

• Keep in mind that the `write` function can only write character data.

  ▶ Therefore, if you need to write data that is numerical, you will have to convert it first.

write4.py

```python
#!/usr/bin/env python3
f = open("numeric", "w")
x = 58
f.write(str(x) + "\n")
f.write(str(43.5))
f.close()
```

• The next example shows how to append data to a file.

append.py

```python
#!/usr/bin/env python3
f = open('test.txt', 'a')
f.write('Appended to the bottom')
f.write(' of the file
')
f.write('More at the bottom
')
f.close()
```

▶ Run multiple times and check the contents of the file `test.txt`.
Writing Data to a File

• There is also a `writelines` function, which writes all the elements of a list to a file.
  
  Each of these elements must be a string.

```python
writelines.py

#!/usr/bin/env python3
f = open("output", "w")
lis = []
while True:
    data = input("Enter data ('q' to exit): ")
    if data == "q":
        break
    lis.append(data)
f.writelines(lis)
f.close()
```

• You can also use the `print` function to write data to a file.
  
  The named parameter `file` in the `print` function can be used to specify an output file.

  In the following program, lines are written to the file associated with the stream `f`.

```python
print1.py

#!/usr/bin/env python3
f = open("output", "w")
while True:
    data = input("Enter data ('q' to exit): ")
    if data == "q":
        break
    print(data, file=f)
f.close()
```
Reading Data From a File

- The following functions can be used to read data once a file has been opened for reading.

- `read()`
  - For reading an entire file into a string
  - Or, the number of characters to be read can be specified by passing it as a parameter to the read function.
    - `read(10)` would read 10 chars from the stream to which it is connected.

- `readline()`
  - For reading a single line into a string
  - Retains the newline character(s)

- `readlines()`
  - For reading an entire file into a list, where each element of the list contains a line from the file
Reading Data From a File

• The program below uses the `readline` method to read lines from a file.
  
  - It counts the number of lines and characters in the file.
  - When the end of the file is reached, the `readline` method returns the value `None`.

```
read1.py

1. #!/usr/bin/env python3
2. import sys
3. ct = lines = 0
4. file = open(sys.argv[1], "r")
5. 
6. while True:
7.     line = file.readline()
8.     if not line:
9.         break
10.    ct += len(line)
11.    lines += 1
12. 
13. file.close()
14. print("Characters:", ct, " Lines:", lines)
```

• The next example uses a slightly different technique.

```
read2.py

1. #!/usr/bin/env python3
2. import sys
3. 
4. lines = 0
5. file = open(sys.argv[1], "r")
6. 
7. line = file.readline()
8. while line:
9.     lines += 1
10.    line = file.readline()
11. file.close()
12. print(lines)
```
Reading Data From a File

- The `readlines` method reads all the lines of a file into a list that you can then process one line at a time.

**read3.py**

```python
#!/usr/bin/env python3
import sys

file = open(sys.argv[1], "r")
lines = file.readlines()
file.close()

for line in lines:
    print(line, end="")
```

- Python also allows the following idiom.

**read4.py**

```python
#!/usr/bin/env python3
import sys

file = open(sys.argv[1], "r")

for line in file:
    print(line, end="")

file.close()
```
Reading Data From a File

- The `read` method reads data from a file into a string.
  - Passing no arguments will cause it to read the whole file.
  - Passing in a number as an argument indicates the quantity of data to be read.

```python
# read5.py
1. #!/usr/bin/env python3
2. import sys
3. 
4. file = open(sys.argv[1], "r")
5. 
6. data = file.read(10)
7. while data:
8.     print(data, end="")
9.     data = file.read(10)
10.
11. file.close()
```
Additional File Methods

- The `seek()` method positions the internal file pointer to a given offset within the file, relative to a reference point within the file so that the next read or write occurs at that new position.

- The `os` module defines three constants to represent the following three reference points.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>os.SEEK_SET</td>
<td>0</td>
<td>Reference point is the beginning of the file</td>
</tr>
<tr>
<td>os.SEEK_CUR</td>
<td>1</td>
<td>Reference point is the current position</td>
</tr>
<tr>
<td>os.SEEK_END</td>
<td>2</td>
<td>Reference point is the end of the file</td>
</tr>
</tbody>
</table>

- Seeking relative to the current position and end position requires an offset of 0 when working with files in text mode.
- Seeking in binary files allows positive and negative offsets.

- The `tell()` method returns the byte offset from the beginning of the file to the current position.

- Here is a small program that demonstrates these functions.

```
seek1.py
1. #!/usr/bin/env python3
2. import os
3. # Reading from file in binary mode
4. file = open("seekdata.txt", "rb")
5. print(file.read())
6. 
7. file.seek(3, os.SEEK_SET)
8. print(file.read(5))
9. print("Position:", file.tell())
10. 
```

- Continued on the following page.
Additional File Methods

seek1.py (continued)

11. file.seek(12, os.SEEK_CUR)
12. print(file.read(7))
13. print("Position:", file.tell())
14.
15. file.seek(-20, os.SEEK_CUR)
16. print(file.read(5))
17. print("Position:", file.tell())
18. file.close()
19.
20. print("#" * 30)
21.
22. # Reading from file in text mode
23. file = open("seekdata.txt", "r")
24. print(file.read())
25.
26. file.seek(12, os.SEEK_SET)
27. print(file.read(5))
28. print("Position:", file.tell())
29.
30. file.seek(0, os.SEEK_CUR)
31. print(file.read(7))
32. print("Position:", file.tell())
33.
34. file.seek(0, os.SEEK_CUR)
35. print(file.read(5))
36. print("Position:", file.tell())
37. file.close()

• The seek method is often used when a file is opened for both reading and writing.
  ▶ Additionally, it is used when a file must be read several times from beginning to end.

• For example, suppose we wish to search a file and list those lines that begin with either of several strings.
  ▶ Suppose further that the file name and the strings are given on the command line.
Additional File Methods

• The interface for such a program would be:

  program filename strings...

• Here is such a program.

  seek2.py

```python
#!/usr/bin/env python3
from sys import argv

file = open(argv[1], "r+")

for word in argv[2:]:
    for line in file:
        if line.startswith(word):
            print(line, end="")

file.seek(0, 0)

file.close()
```

The output of the above program is shown below.

```
$ python3 seek2.py seek2.py file for
file = open(argv[1], "r+")
file.close()
for word in argv[2:]:
$
```
Using Pipes as Data Streams

- Sometimes it is necessary for a Python application to communicate with an operating system command line command.
  - For example, you might wish to have the output from the Linux `ls -l` command sent to your Python program.
  - Alternately, you might want the Python program to send some data to the `sort` command line command.

- The `popen()` function allows you to read and/or write from and to an operating system command.
  - This function requires the importing of the `platform` module.

```python
import platform
input = platform.popen("dir", "r");
output = platform.popen("sort", "w")
```

- If the pipe is an input data stream, the Python application will read data from the standard output of the executing command.
- If the pipe is an output data stream, the Python application will write data to the standard input for the executing command.

- Below is a program that reads data from a process.

```python
pipe1.py
```

```python
1. #!/usr/bin/env python3
2. import platform
3. p = platform.popen("dir", "r")
4. data = p.readlines()
5. p.close()
6. for i in data:
   7.     print(i, end="")
```
Using Pipes as Data Streams

The output of the previous program is shown below.

```
$ python3 pipe1.py
allchars.py    pickle1.py  print1.py  readline.py   sys_output.py
append.py      pickle2.py  read1.py   readlines.py  write1.py
chars.py       pickra.py   read2.py   seek1.py     write2.py
countlines.py  pickwa.py   read3.py   seek2.py     write3.py
DATA           pipe1.py    read4.py   seekdata.txt  write4.py
ioerror.py     pipe2.py    read5.py   stat.py      writelines.py
$
```

The following program writes data to a process.

**pipe2.py**

1. `#!/usr/bin/env python3`
2. `import platform`
3. 
4. `data = [ "mike", "jane", "alice", "ruth" ]`
5. 
6. `p = platform.popen("sort", "w")`
7. `for i in data:`
8. `print(i, file=p)`
9. 
10. `p.close()`

The output of the above program is shown below.

```
$ python3 pipe2.py
alice
jane
mike
ruth
$
```
Handling IO Exceptions

• None of the programs so far has been concerned with file open failures.
  • The program below demonstrates one way to handle this problem.

```python
ioerror.py

1. #!/usr/bin/env python3
2. import sys
3.
4. try:
5.     file = open(sys.argv[1], "r")
6. except OSError as err:
7.     print(type(err))
8.     print(err)
9.     exit(1)
10. finally:
11.     print("IN FINALLY")
12.
13. num = 1
14. line = file.readline()
15. while line:
16.     print(str(num) + "\t" + line, end="")
17.     num += 1
18.     line = file.readline()
19.
20. file.close()
```
Working with Directories

- The `os` module contains operating system specifics.
  - The items shown in this section work for both Windows and Unix/Linux operating systems.
  - `os.name` yields "nt" or "posix" depending upon whether you are running Windows or Linux.
  - `os.environ` is a dictionary having access to environmental variables such as:
    ```
    os.environ['PATH']
    os.environ['LOGNAME']
    ```
  - `os.chdir` allows you to change directories.
  - `os.getcwd` allows you to determine the current directory.

- Here is an example of counting the lines over all files in a directory.

```python
countlines.py
1. #!/usr/bin/env python3
2. import os
3.
4. files = os.listdir(".")
5. ct = 0
6.
7. for file in files:
8.     f = open(file, "r")
9.     lines = f.readlines()
10.    ct += len(lines)
11.    f.close()
12.
13. print(ct)
```
Metadata

- It is often necessary to determine information about files.
  - The `os.stat(filename)` method takes a filename and returns a tuple containing information about a file `filename`.
    - Its values can be accessed by index and by attribute name.
  - The following table lists the values that are available from the `stat()` function.

<table>
<thead>
<tr>
<th>tuple Index</th>
<th>Attribute Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><code>st_mode</code></td>
<td>File mode (type and permissions)</td>
</tr>
<tr>
<td>1</td>
<td><code>st_ino</code></td>
<td>The inode number</td>
</tr>
<tr>
<td>2</td>
<td><code>st_dev</code></td>
<td>Device number of filesystem</td>
</tr>
<tr>
<td>3</td>
<td><code>st_nlink</code></td>
<td>Number of hard links to the file</td>
</tr>
<tr>
<td>4</td>
<td><code>st_uid</code></td>
<td>Numeric user id of file’s owner</td>
</tr>
<tr>
<td>5</td>
<td><code>st_gid</code></td>
<td>Numerical group id of file’s owner</td>
</tr>
<tr>
<td>6</td>
<td><code>st_size</code></td>
<td>Size of file in bytes</td>
</tr>
<tr>
<td>7</td>
<td><code>st_atime</code></td>
<td>Last access time (seconds since epoch)</td>
</tr>
<tr>
<td>8</td>
<td><code>st_mtime</code></td>
<td>Last modify time (seconds since epoch)</td>
</tr>
<tr>
<td>9</td>
<td><code>st_ctime</code></td>
<td>Linux: Last inode change time (seconds since epoch)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Windows: Creation time (seconds since epoch)</td>
</tr>
</tbody>
</table>

- The program on the following page takes any number of files on the command line and displays the `stat()` information for each file.
  - The `stat()` values are shown twice for each file.
    - First by index
    - Then by attribute name
Metadata

filestats.py

```python
#!/usr/bin/env python3
import sys, os, time

tag = [
    "tmode", "tinode#", "tdevice#",
    "t#links", "tuser", "tgroup", "tbytes",
    "tlast access", "tlast modified",
    "tchange/creation time"
]

def printstats01(file, stat):
    print("File Stats for:", file)
    print(tag[0], ":", oct(stat.st_mode))
    print(tag[1], ":", stat.st_ino)
    print(tag[2], ":", stat.st_dev)
    print(tag[3], ":", stat.st_nlink)
    print(tag[4], ":", stat.st_uid)
    print(tag[5], ":", stat.st_gid)
    print(tag[6], ":", stat.st_size)
    print(tag[7], ":", time.ctime(stat.st_atime))
    print(tag[8], ":", time.ctime(stat.st_mtime))
    print(tag[9], ":", time.ctime(stat.st_ctime))
    print("\n")

def printstats02(file, stats):
    print("File Stats for:", file)
    for i, a_stat in enumerate(stats):
        print(tag[i], ":", a_stat)
    print("\n")

for file in sys.argv[1:]:
    info = os.stat(file)
    if os.path.isfile(file):
        print("*" * 30)
        printstats01(file, info)
        printstats02(file, info)
```

- The output of the above program is shown on the following page.
You can also determine information about files by using the following methods.

- Each of the following methods returns True or False.
  
  - `os.path.isdir("dirname")`
  - `os.path.isfile("filename")`
  - `os.path.exists("filename")`
The pickle Module

- The pickle module implements an algorithm for serializing and de-serializing a Python object structure.
  - Pickling is the process whereby a Python object is converted into a byte stream.
    - Pickling is also known as serialization or marshalling.
  - Unpickling is the inverse operation, whereby a byte stream is converted back into an object.
    - Unpickling is also known as deserialization or unmarshalling.

- The pickle module is extensive and powerful and can handle data of any complexity.
  - However, in this section we limit the pickle examples to those data types that we have seen to this point in the course.
  - In the examples that we offer here, we will typically show a pair of programs.
    - The first program will dump data to a file.
    - The second program will load that data from a file.

- The first example demonstrates the dumping of an integer, a floating point number, and a string.

pickle1.py

```python
1. #!/usr/bin/env python3
2. import pickle
3. values = [50, 32.29, "Michael"]
4. f = open("output", "wb")
5. for value in values:
6.     pickle.dump(value, f)
7. f.close()
```
The pickle Module

- When the previous program is run, the data is sent to the file named output, which will be a binary file, possibly containing bytes that print as unusual characters.
  - Any file produced with pickle.dump() would normally be read with pickle.load(), as the next example illustrates.

- Here is a program that reads the "pickled" data produced by the previous program.

```python
pickle2.py
1. #!/usr/bin/env python3
2. import pickle
3. f = open("output", "rb")
4. value = pickle.load(f)
5. cost = pickle.load(f)
6. name = pickle.load(f)
7. f.close()
8. print(name, cost, value)
```

- You can see that the load function from the pickle module correctly loaded the data that was created with the dump function from the previous program.

- The following pair of programs will demonstrate writing several collections and then reading them.
  - The program that reads them back in will incorporate exception handling to determine when to stop loading objects from the file.

The **pickle Module**

### pickle_dump.py

```python
#!/usr/bin/env python3
import pickle

a_list = [10, 11, 12, 13]
a_tuple = ("A", "B", "C", "D")
a_set = {1, 2, 3, 4, 5, 6}
a_map = {'a':5,'b':6,'c':7}
elements = [a_list, a_tuple, a_set, a_map]
f = open("output", "wb")
for element in elements:
pickle.dump(element, f)
f.close()
```

### pickle_load.py

```python
#!/usr/bin/env python3
import pickle

f = open("output", "rb")
try:
    while True:
        obj = pickle.load(f)
        print(obj)
except EOFError:
    pass # Quietly ignoring since this is expected
f.close()
```

- The output of the above `pickle_load.py` program is shown below.

```
student@localhost:~/pythonlabs/examples/8$ python3 pickle_load.py
[10, 11, 12, 13]
('A', 'B', 'C', 'D')
{1, 2, 3, 4, 5, 6}
{'a': 5, 'b': 6, 'c': 7}
```

©2013 /training/etc Inc. REPRODUCTION OF THESE MATERIALS IS PROHIBITED.
8-25
Exercises

1. Write a program that asks the user for the names of an input and an output file.
   - Open both of these files and then have the program read from the input file (use `readline`) and write to the output file (use `write`).
   - In effect, this is a copy program.
   - The interface to the program might look like:
     Enter the name of the input file: myinput
     Enter the name of the output file: myoutput

2. Rewrite Exercise 1 but this time get the file names from the command line.
   - The interface would look as shown below.
     - Make sure the correct number of command line arguments is provided.
     - Otherwise, print an error message and terminate the program.
     python3 program_name inputfile outputfile

3. Add exception handling to Exercise 2 so that if a file open fails, an `OSError` is handled and the program is halted.

4. Write a program that displays the file name, size, and modification date for all those files in a directory that are greater than a certain size.
   - The directory name and the size criteria are given as command line arguments.
   - If the number of command line arguments is incorrect, the program should print an error message and terminate.
Exercises

5. Create two data files, each with a set of names, one per line.

- Now, write a program that reads both files and lists only those names that are in both files.
- The two file names should be supplied on the command line.

6. Now, create a few more files file with one name per line.

- The program in this exercise should read all these files and print the number of times each line occurs over all of the files.
- The file names should be supplied on the command line.

- For example:

<table>
<thead>
<tr>
<th>file1</th>
<th>file2</th>
<th>file3</th>
<th>file4</th>
</tr>
</thead>
<tbody>
<tr>
<td>jane</td>
<td>susan</td>
<td>jane</td>
<td>jake</td>
</tr>
<tr>
<td>john</td>
<td>mary</td>
<td>john</td>
<td>chris</td>
</tr>
<tr>
<td>peter</td>
<td>dave</td>
<td>peter</td>
<td>peter</td>
</tr>
<tr>
<td>bill</td>
<td>mike</td>
<td>bill</td>
<td>bill</td>
</tr>
<tr>
<td>mike</td>
<td>alice</td>
<td>mike</td>
<td>mike</td>
</tr>
<tr>
<td>alice</td>
<td>chris</td>
<td>alice</td>
<td>alice</td>
</tr>
<tr>
<td>frank</td>
<td>beverly</td>
<td>frank</td>
<td>frank</td>
</tr>
<tr>
<td>bart</td>
<td>bill</td>
<td>bart</td>
<td>susan</td>
</tr>
</tbody>
</table>

Output:

- alice  4
- bart  2
- beverly  1
- bill  4
- chris  2
- dave  1
- frank  3
- jane  3
- john  2
- mary  1
- mike  4
- peter  3
- susan  2
Exercises

7. Write a program that counts the number of lines, words, and characters in each file named on the command line.

8. Revise your solution to the previous exercise so that if you specify the "-t" option on the command line (before the list of files), your program also prints total number of lines, words, and characters in all the files.