INTRODUCTION TO PYTHON 3

INTRODUCTION TO PYTHON 3

This Page Intentionally Left Blank

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 classs
  - Understand and use the Object Oriented paradigm in Python programs
  - Use the Python Regular Expression capabilities for data verification

This Page Intentionally Left Blank

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

Copying Collections 4-17
Summary 4-19

CHAPTER 5: FUNCTIONS

Introduction 5-2
Defining Your Own Functions 5-3
Parameters 5-4
Function Documentation 5-5
Keyword and Optional Parameters 5-6
Passing Collections to a Function 5-7
Variable Number of Arguments 5-8
Scope 5-11
Functions - "First Class Citizens" 5-13
Passing Functions to a Function 5-14
`map` 5-15
`filter` 5-16
Mapping Functions in a Dictionary 5-17
Lambda 5-18
Inner Functions 5-19
Closures 5-20

CHAPTER 6: MODULES

Modules 6-2
Standard Modules - `sys` 6-5
Standard Modules - `math` 6-6
Standard Modules - `time` 6-7
The `dir` Function 6-9

CHAPTER 7: EXCEPTIONS

Errors 7-2
Runtime Errors 7-3
The Exception Model 7-4
Exception Hierarchy 7-7
Handling Multiple Exceptions 7-9
`raise` 7-11
`assert` 7-12

CHAPTER 8: INPUT AND OUTPUT

Introduction 8-2
Data Streams 8-3
Creating Your Own Data Streams 8-4
Access Modes 8-5
Writing Data to a File 8-6
Reading Data From a File 8-9
Additional File Methods 8-13
Using Pipes as Data Streams 8-16
Handling IO Exceptions 8-18
Working with Directories 8-19

Metadata 8-20
The `pickle` Module 8-23

CHAPTER 9: CLASSES IN PYTHON
Classes in Python 9-2
Principles of Object Orientation 9-3
Creating Classes 9-4
Instance Methods 9-7
File Organization 9-8
Special Methods 9-9
Class Variables 9-13
Inheritance 9-14
Polymorphism 9-17
Type Identification 9-22
Custom Exception Classes 9-23

CHAPTER 10: REGULAR EXPRESSIONS
Introduction 10-2
Simple Character Matches 10-4
Special Characters 10-6
Character Classes 10-7
Quantifiers 10-9
The Dot Character 10-10
Greedy Matches 10-11
Grouping 10-12
Matching at Beginning or End 10-14
Match Objects 10-15
Substituting 10-17
Splitting a String 10-18
Compiling Regular Expressions 10-19
Flags 10-20

This Page Intentionally Left Blank

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 **P**ython **E**nhancement **P**roposal (**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.

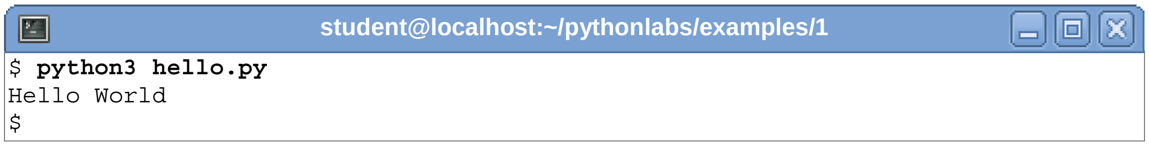
```
#!/usr/bin/env python3
```

- The simple example script is shown below.

hello.py

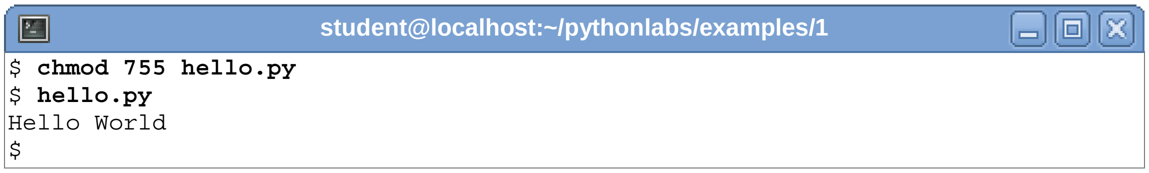
```
#!/usr/bin/env python3
print("Hello World")
```

  - The script would then be executed as follows.



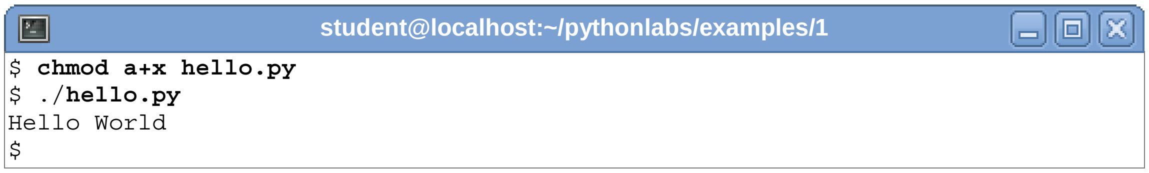
```
$ python3 hello.py
Hello World
$
```

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



```
$ chmod 755 hello.py
$ hello.py
Hello World
$
```

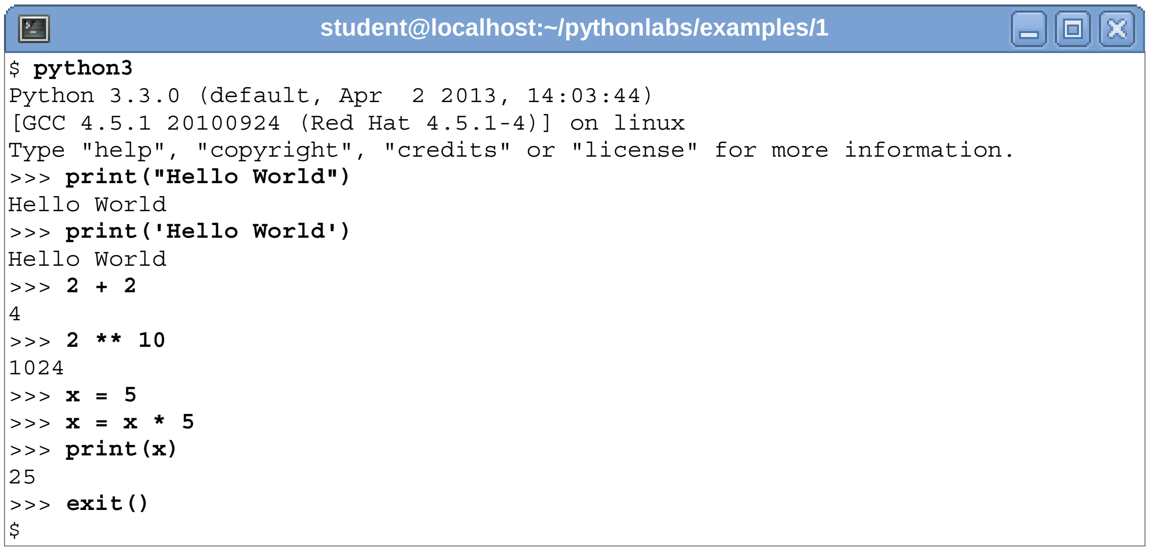
  - A variation on the above is shown below.
    - It uses `a+x` instead of `755`
    - It also uses `./hello.py` instead of `hello.py` if the current directory (`.`) is not on the PATH environment variable.



```
$ chmod a+x hello.py
$ ./hello.py
Hello World
$
```

Executing Python from the Command Line

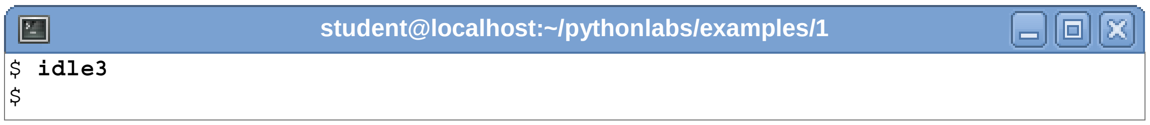
- This first program is offered merely to demonstrate the execution of a Python program from the command line.
  - The `print` function sends data to the standard output.
    - In Python 3, parentheses are required for function arguments.
    - This is not the case in previous Python versions.
- You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



```
$ 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()
$
```

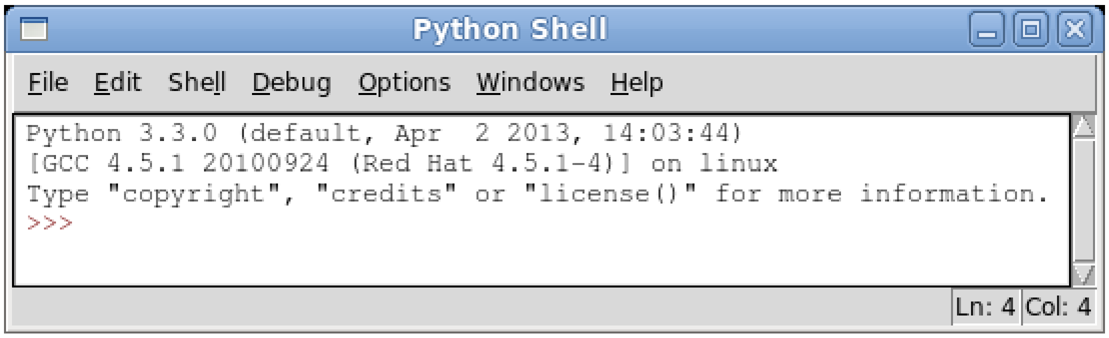
IDLE

- 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.



```
$ idle3
$
```

  - You will be presented with the following window and interactive environment.

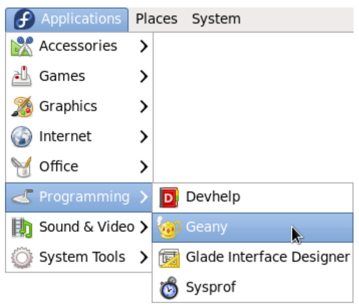
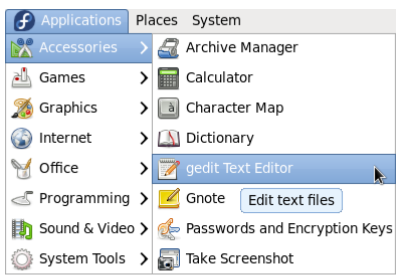


```
Python 3.3.0 (default, Apr  2 2013, 14:03:44)
[GCC 4.5.1 20100924 (Red Hat 4.5.1-4)] on linux
Type "copyright", "credits" or "license()" for more information.
>>>
```

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.

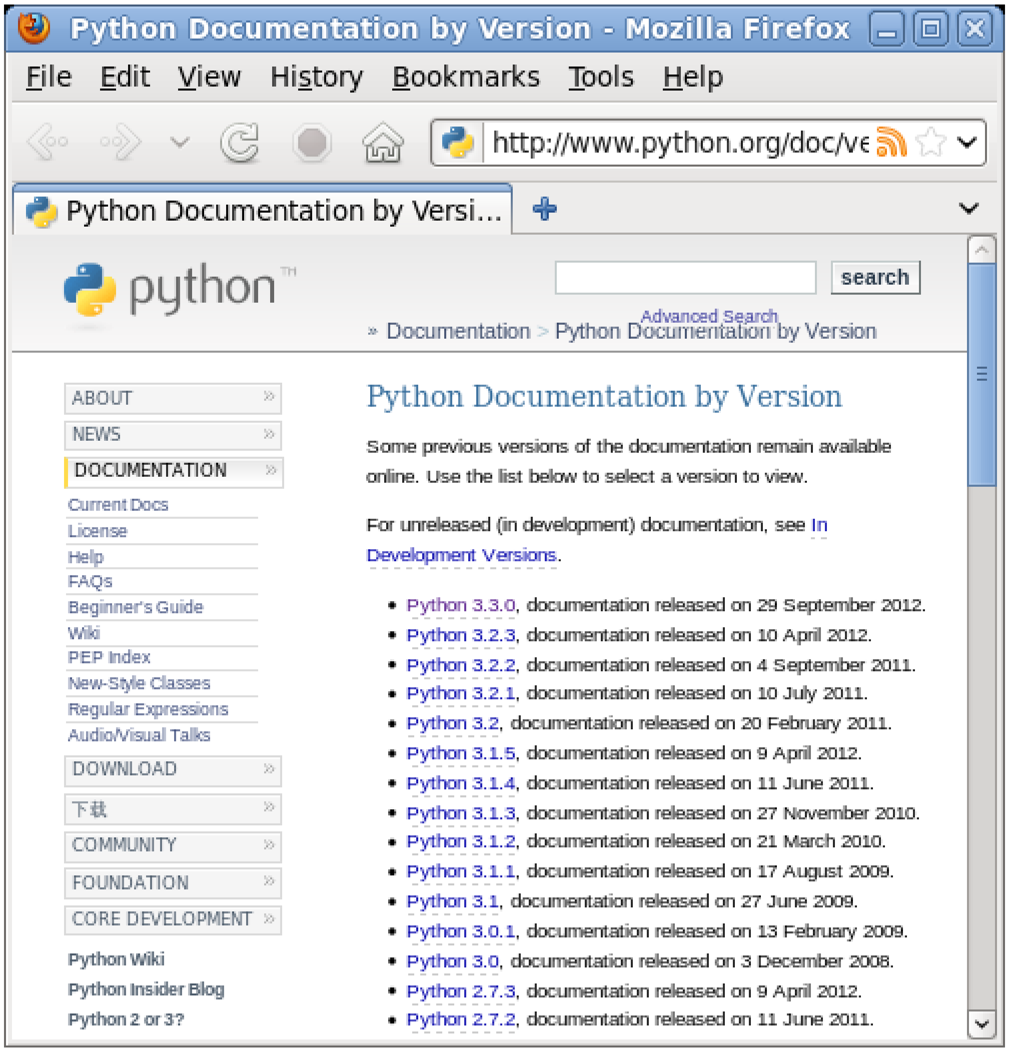
    `http://wiki.python.org/moin/PythonEditors`
- As we will see later, Python relies heavily on indention of the source code to define blocks of code.
  - This is in sharp contrast to the more traditional approach of using keywords or curly braces.
    - Since there are many different Integrated Development 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.

Geany	gedit
	

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/`



Python Documentation by Version

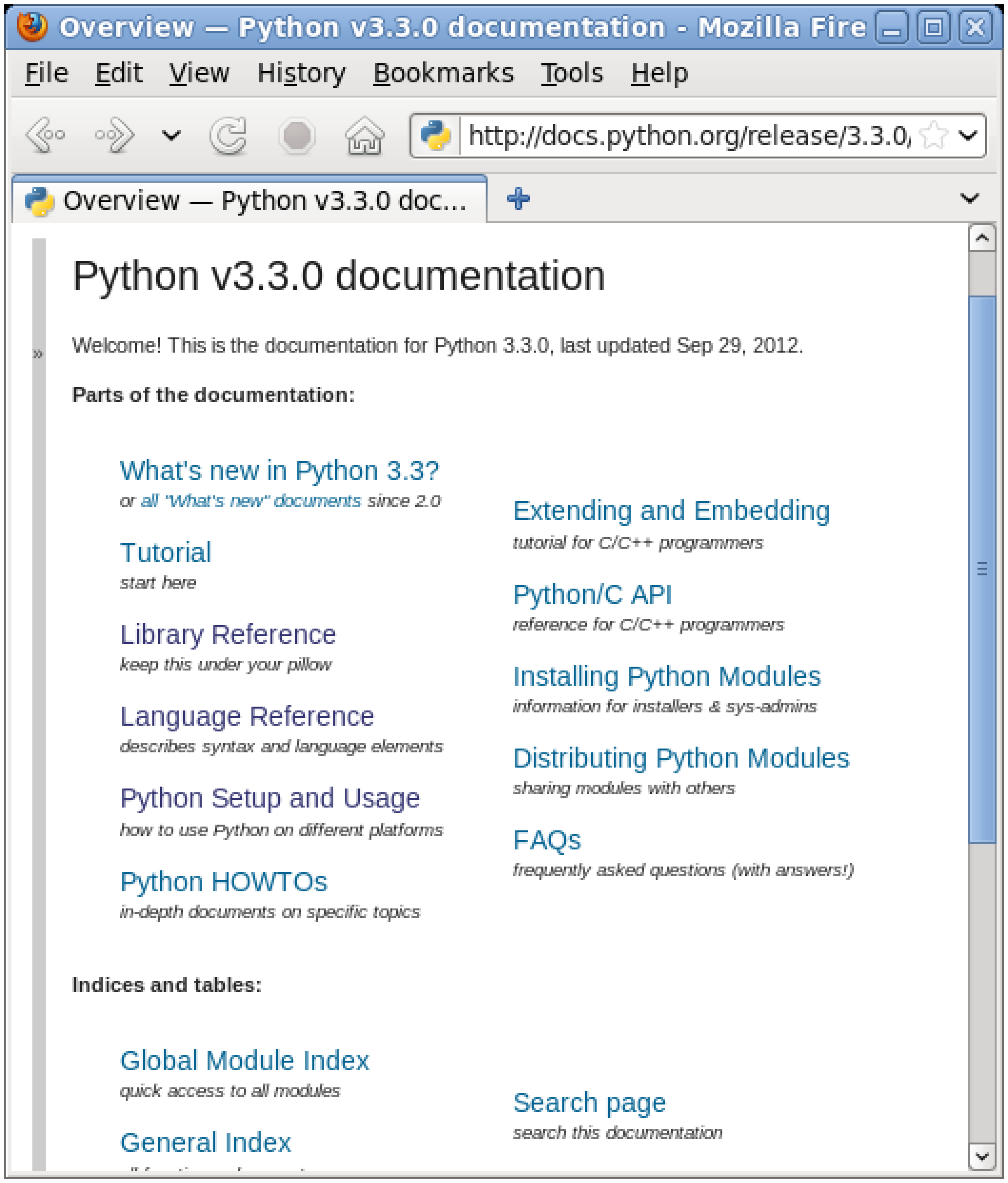
Some previous versions of the documentation remain available online. Use the list below to select a version to view.

For unreleased (in development) documentation, see In Development Versions.

- Python 3.3.0, documentation released on 29 September 2012.
- Python 3.2.3, documentation released on 10 April 2012.
- Python 3.2.2, documentation released on 4 September 2011.
- Python 3.2.1, documentation released on 10 July 2011.
- Python 3.2, documentation released on 20 February 2011.
- Python 3.1.5, documentation released on 9 April 2012.
- Python 3.1.4, documentation released on 11 June 2011.
- Python 3.1.3, documentation released on 27 November 2010.
- Python 3.1.2, documentation released on 21 March 2010.
- Python 3.1.1, documentation released on 17 August 2009.
- Python 3.1, documentation released on 27 June 2009.
- Python 3.0.1, documentation released on 13 February 2009.
- Python 3.0, documentation released on 3 December 2008.
- Python 2.7.3, documentation released on 9 April 2012.
- Python 2.7.2, documentation released on 11 June 2011.

- 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.

Python Documentation


Overview — Python v3.3.0 documentation - Mozilla Fire
File Edit View History Bookmarks Tools Help
http://docs.python.org/release/3.3.0/
Overview — Python v3.3.0 doc...

Python v3.3.0 documentation

Welcome! This is the documentation for Python 3.3.0, last updated Sep 29, 2012.

Parts of the documentation:

What's new in Python 3.3?
or all "What's new" documents since 2.0

Tutorial
start here

Library Reference
keep this under your pillow

Language Reference
describes syntax and language elements

Python Setup and Usage
how to use Python on different platforms

Python HOWTOs
in-depth documents on specific topics

Extending and Embedding
tutorial for C/C++ programmers

Python/C API
reference for C/C++ programmers

Installing Python Modules
information for installers & sys-admins

Distributing Python Modules
sharing modules with others

FAQs
frequently asked questions (with answers!)

Indices and tables:

Global Module Index
quick access to all modules

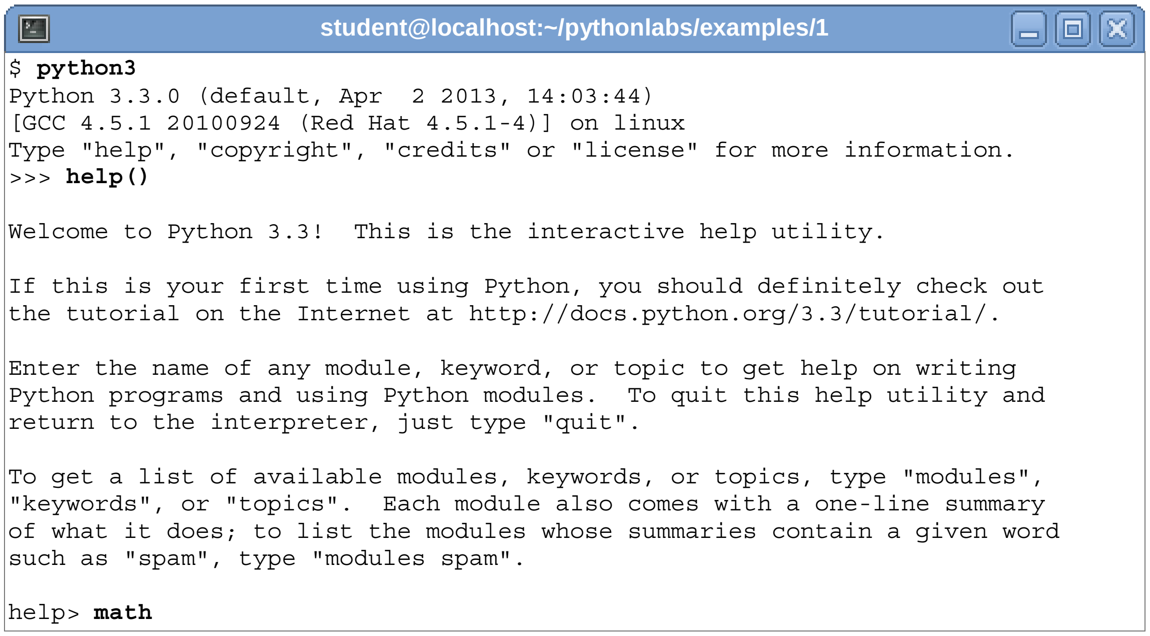
General Index

Search page
search this documentation

- 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.

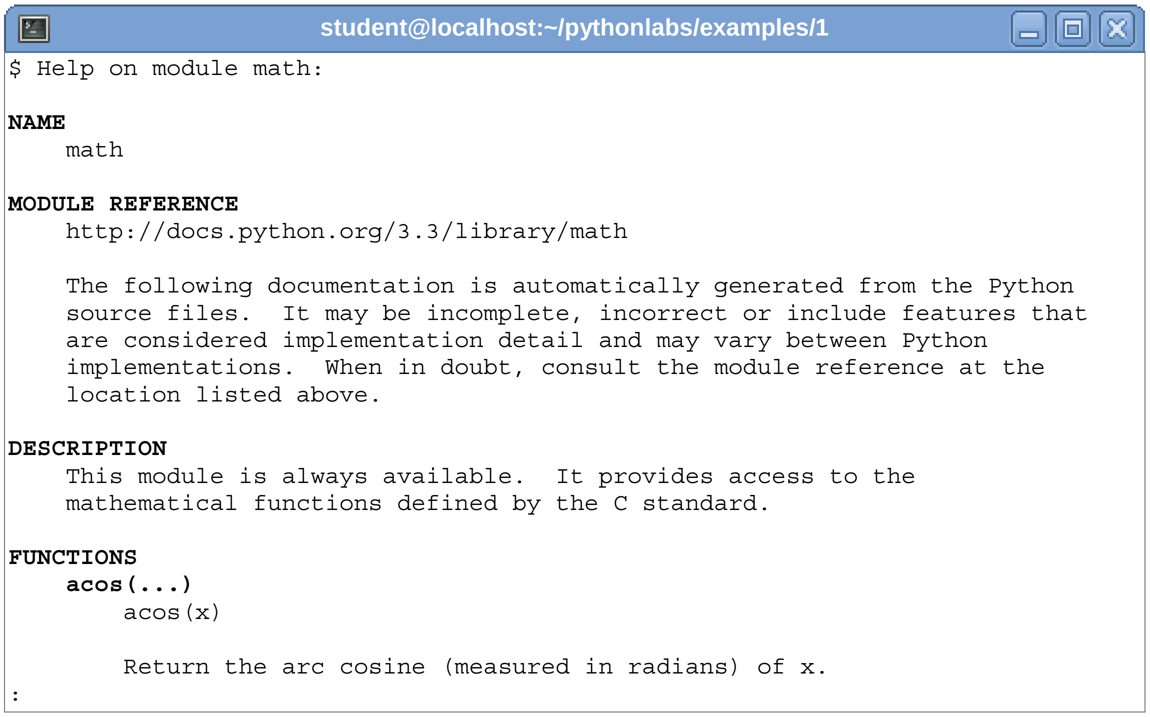


student@localhost:~/pythonlabs/examples/1

```
$ 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.
>>> 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.

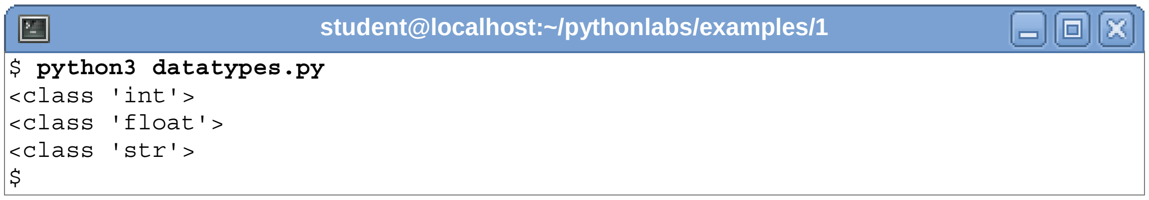
- Alternatively, at the interactive Python prompt `>>>`, help can be obtained by passing information to the help method as shown below.
  - `help('math')`, `help('str')`, etc.

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.

`datatypes.py`

```
#!/usr/bin/env python3
x = 10
print (type(x))
x = 25.7
print (type(x))
x = "Hello"
print (type(x))
```



```
$ python3 datatypes.py
<class 'int'>
<class 'float'>
<class 'str'>
$
```

- 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.

Python Keywords			
`False`	`def`	`if`	`raise`
`None`	`del`	`import`	`return`
`True`	`elif`	`in`	`try`
`and`	`else`	`is`	`while`
`as`	`except`	`lambda`	`with`
`assert`	`finally`	`nonlocal`	`yield`
`break`	`for`	`not`	
`class`	`from`	`or`	
`continue`	`global`	`pass`	

- 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 aready 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.

This Page Intentionally Left Blank

Chapter 2: Basic Python Syntax

1) Basic Syntax 2-2

2) Comments 2-4

3) String Values 2-5

4) String Methods 2-6

5) The `format` Method 2-10

6) String Operators 2-11

7) Numeric Data Types 2-13

8) Conversion Functions 2-14

9) Simple Output 2-15

10) Simple Input 2-16

11) The `%` Method 2-18

12) The `print` Function 2-19

Basic Syntax

- This section deals with many of the issues relating to the fundamental syntax within the Python language.

- Python is case sensitive.
  - Therefore, the following two variables are different.

    ```
    ThePerson = "Him"
    theperson = "Her"
    ```

- Python statements do not need to end with a semi-colon, but one can be used to separate two statements if they are on the same line.

  ```
  length = 10; width = 5
  area = length * width
  print(area)
  ```

- Python is not a forgiving language where source code layout is concerned.
  - All lines in a Python application must begin in the first column, unless the statement is in the body of a loop, conditional, function, or class definition.
    - We will discuss each of these topics later in the course.
  - Python relies heavily on indention to determine the control flow structure of an application.
    - Indention of lines is typically a multiple of four spaces.
  - As you are introduced to control structures, we will revisit the indentation rules.
    - For now, remember to start your Python statements in the first column.

Basic Syntax

- A long statement may be continued by placing the `\` character at the end of the line.

```
x = "This is a long string just to illustrate \
continuation"
```

- However, statements are automatically continued onto the following line(s) if the end of line is reached before the ending character for any of the following pairs.

```
()   []   {}
```

- Below is an example.

```
print(
    "Python \
     vs. Perl"
)
```

- Note that automatically continued lines can be indented any number of spaces.

Comments

- Programs are generally more understandable and maintainable if the code contains some programmer written comments.

- Single line comments begin with the `#` character and terminate with the physical line on which the `#` character has been typed.

  - A few examples of this style of are shown below.

comments.py

```
#!/usr/bin/env python3
#
#       some comments
#
x = 10                          # an integer
y = "hello there!"              # a string
z = "This # is not a comment "
```

- Python provides a simple way of writing multi-line comments.

  - You simply start the comments with three `"` characters and end with three `"` characters.

  - These comments can also be used for documentation purposes.

    - Below is an example of a few multi-line comments.

multiline.py

```
#!/usr/bin/env python3
"""
    This is an example of a
    multi-line comment
"""
""" another one """
```

String Values

- Literal string values may be enclosed in single or double quotes.
  - These quotes are equivalent to one another.

```
s1 = "This is a string"
s2 = 'and so is this'
s3 = 'My name is "Michael"'
s4 = "This is a \" single quote"
```

- In the last string above, the `\` character escapes the `"` character and causes the (middle) `"` character to be printed rather than interpreted as the end of string character.

- The following is a list of some of the more common escape sequences.
  - Each of them must be inside either single or double quotes.

Sequence	Character
`\n`	New line
`\r`	Carriage return
`\t`	Tab
`\b`	Backspace
`\\`	Backslash
`\"`	Double quote
`\'`	Single quote

- Strings can be defined using a multi-line string.

```
text = """This
is a multiline
string"""
```

- It is not treated as a multiline comment since the `"""` characters do not start in column one of the line of code.

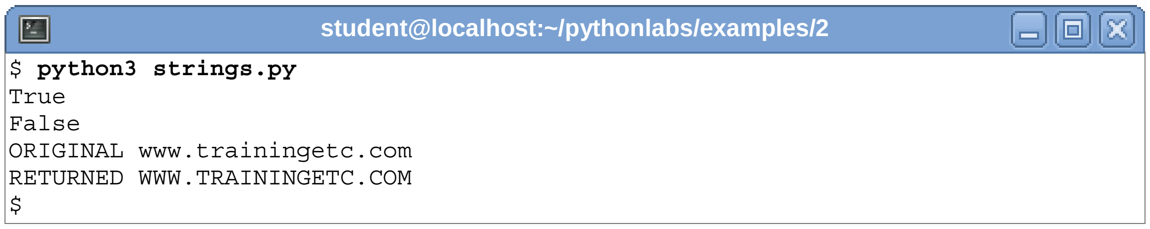
String Methods

- Later in this course we will discuss Object Oriented Programming in more depth.
  - For those not familiar with object-oriented programming, you can think of a class as a new data type and an object as an instance of that data type.
- The `str` class is Python's version of a string class.
  - This class has an abundance of methods defined within it.
  - While some of the methods return values such as `True` or `False`, other methods return a modified version of the string on which they operate.
    - This can be seen in the example below.

strings.py

```
#!/usr/bin/env python3
x = "www.trainingetc.com"
result = x.startswith("www")
print(result)
print(x.endswith(".org"))
y = x.upper()
print("ORIGINAL", x)
print("RETURNED", y)
```

- The output of the above program is shown below.



```
$ python3 strings.py
True
False
ORIGINAL www.trainingetc.com
RETURNED WWW.TRAININGETC.COM
$
```

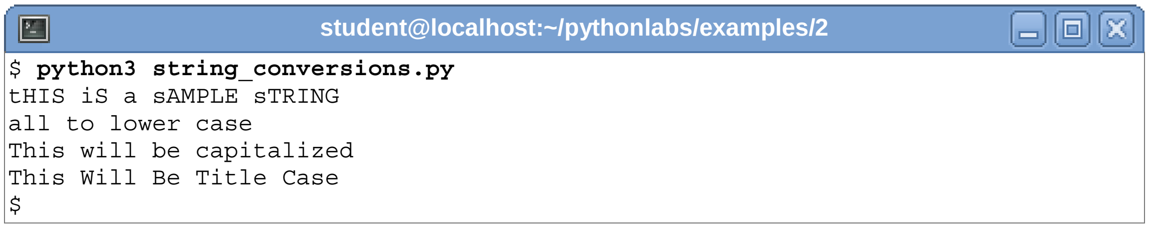
String Methods

- The program below demonstrates additional methods that convert the case of a string.

`string_conversions.py`

```
#!/usr/bin/env python3
print("This Is A Sample String".swapcase())
print("ALL To Lower Case".lower())
print("this will be capitalized".capitalize())
print("this will be title case".title())
```

  - The output of the above program is shown below.



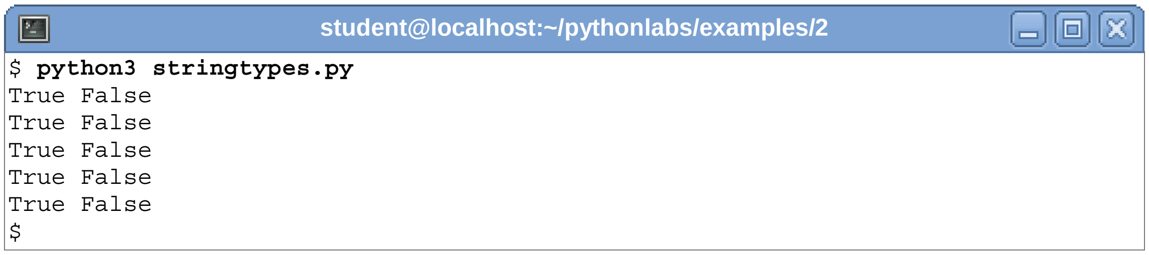
```
$ python3 string_conversions.py
tHIS iS a sAMPLE sTRING
all to lower case
This will be capitalized
This Will Be Title Case
$
```

- Below are some of the methods that, when called on a string, return `True` if the characters in the string are of a specific kind.

`stringtypes.py`

```
#!/usr/bin/env python3
print("AbCDefg".isalpha(), "AbCDe123".isalpha())
print("12345".isnumeric(), "12345BCD".isnumeric())
print("   \t\n".isspace(), "a  b\t\n".isspace())
print("ABCDEFG".isupper(), "abcdefg".isupper())
print("abcdefg".islower(), "ABCDEFG".islower())
```

  - The output of the above program is shown below.



```
$ python3 stringtypes.py
True False
True False
True False
True False
True False
$
```

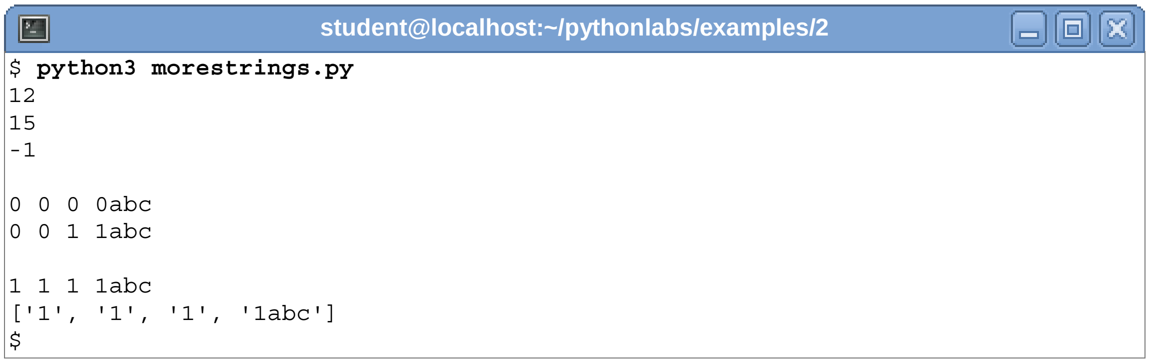
String Methods

- Here are a few additional string methods.

morestrings.py

```
#!/usr/bin/env python3
x = "Capital of Mississippi is Jackson"
pos = x.find("is")
print(pos)
print(x.find("is", pos + 1))
print(x.find("is", 8, 12))
print()

x = "1 1 1 1abc"
y = x.replace("1","0")
print(y)
y = x.replace("1", "0", 2)
print(y)
print()

list = x.split(' ')
print(x)
print(list)
```

- The output of the above program is shown below.



```
$ python3 morestrings.py
12
15
-1

0 0 0 0abc
0 0 1 1abc

1 1 1 1abc
['1', '1', '1', '1abc']
$
```

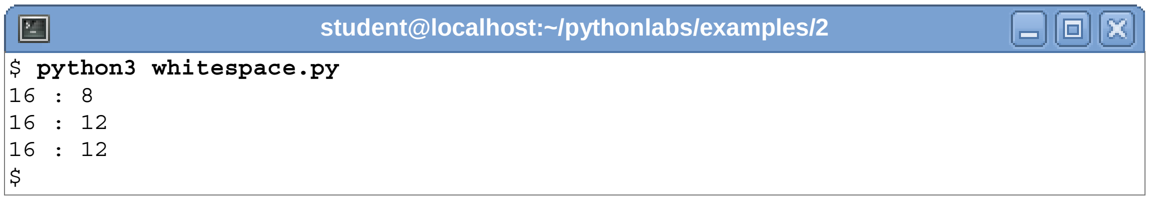
String Operations

- The following methods strip whitespace characters from a string.

`whitespace.py`

```python
#!/usr/bin/env python3
data = "\t  \nabc  def\t  \n"

# The strip method removes leading and trailing
# whitespace
result = data.strip()
print(len(data), ":", len(result))

# The rstrip method removes trailing whitespace
result = data.rstrip()
print(len(data), ":", len(result))

# The lstrip method removes leading whitespace
result = data.lstrip()
print(len(data), ":", len(result))
```

- The output of the above program is shown below.



```
$ python3 whitespace.py
16 : 8
16 : 12
16 : 12
$
```

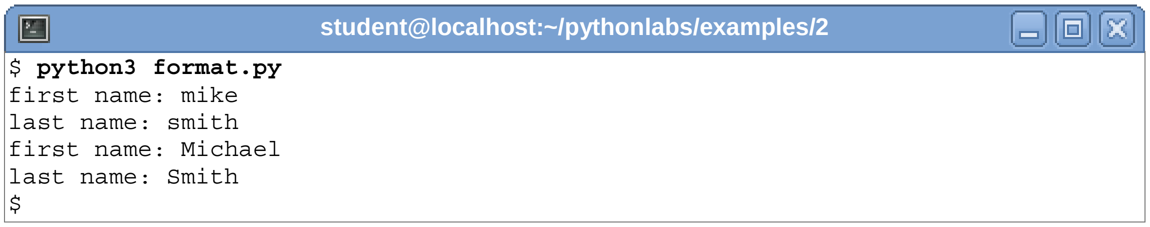
The `format` Method

- The `format()` method of Python's `str` class provides a simple way to construct string values using placeholder parameters.
  - Placeholders are specified within the string using the `{}` (curly brace) characters.
    - If the `{}` surround an integer number, the substitution value for the placeholder is passed to the `format()` method by position.
    - If the `{}` surround a word, the substitution value is passed by name.
- Below are a few examples of using the `format` method.

format.py

```
#!/usr/bin/env python3
quot = "first name: {0}\nlast name: {1}"
line = quot.format("mike", "smith")
print(line)

quot="first name: {first}\nlast name: {last}"
line=quot.format(first="Michael", last="Smith")
print(line)
```

  - The output of the above program is shown below.



```
$ python3 format.py
first name: mike
last name: smith
first name: Michael
last name: Smith
$
```

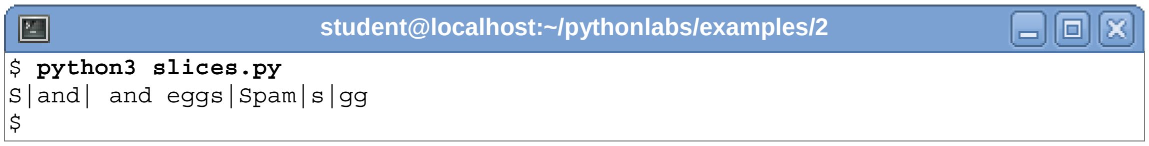
String Operators

- You can use the `[ ]` operator to obtain a substring from a Python string.
  - Each substring is called a **slice**.

slices.py

```
#!/usr/bin/env python3
text = "Spam and eggs"
s1 = text[0]                # "S"
s2 = text[5:8]              # "and"
s3 = text[4:]               # " and eggs"
s4 = text[:4]               # "Spam"
s5 = text[-1]               # "s"
s6 = text[-3:-1]            # "gg"

fmt = "{0}|{1}|{2}|{3}|{4}|{5}"
print(fmt.format(s1, s2, s3, s4, s5, s6))
```

  - The output of the above program is shown below.



```
$ python3 slices.py
S|and| and eggs|Spam|s|gg
$
```

- As you can see, a string may be subscripted to access individual characters or sequences of characters.
  - The leftmost character of a string has the index of 0 (zero).
  - Negative subscripts may be used to reference characters relative to the end of the string.
    - Note the following common slices.

```
s = "whatever"
s[1:]       # all but the first character
s[0:-1]     # all but the last character
```

String Operators

- Note that the slice `[5:8]` yields the characters at positions, 5, 6 and 7, but not 8, and that the length of such a slice is 3.

- Expressions can also be used within the square brackets.

```
s = "This is a string"
a = 1
b = 4
s[a:b -1]  # hi
```

- Additional string operators are demonstrated in the example below.

stringops.py

```
#!/usr/bin/env python3
# The plus (+) operator returns concatenated strings
first = "Mike"
last = "Smith"
full = first + " " + last
print(full)  # Mike Smith

# The asterisk (*) operator returns a repeated string
stars = "***" * 3
print(len(stars), ":", stars) # 9 : *********

# The in operator is convenient for membership tests
x = "Hello there"
print('t' in x)     # True
print('ell' in x)   # True
print('hell' in x)  # False
```

- Custom classes can also provide operator methods.
  - We will discuss custom classes and their operators later in the course.

Numeric Data Types

- Python supports three different numeric data types.
  - Integer
  - Floating Point
  - Complex
- While complex numbers are typically used for specific kinds of mathematical and engineering applications, almost all applications will involve some degree of integer and floating point operations.
  - Here is a list of Python's numerical operators and their meanings.

Operation	Result
`x + y`	Sum of `x` and `y`
`x - y`	Difference of `x` and `y`
`x * y`	Product of `x` and `y`
`x / y`	Quotient of `x` and `y`
`x // y`	Floored quotient of `x` and `y`
`x % y`	Remainder of `x` / `y`
`-x`	`x` negated
`+x`	`x` unchanged
`abs(x)`	Absolute value (or magnitude) of `x`
`int(x)`	`x` converted to integer
`float(x)`	`x` converted to floating point
`divmod(x,y)`	The pair `(x // y, x % y)`
`pow(x, y)`	`x` to the power `y`
`x ** y`	`x` to the power `y`

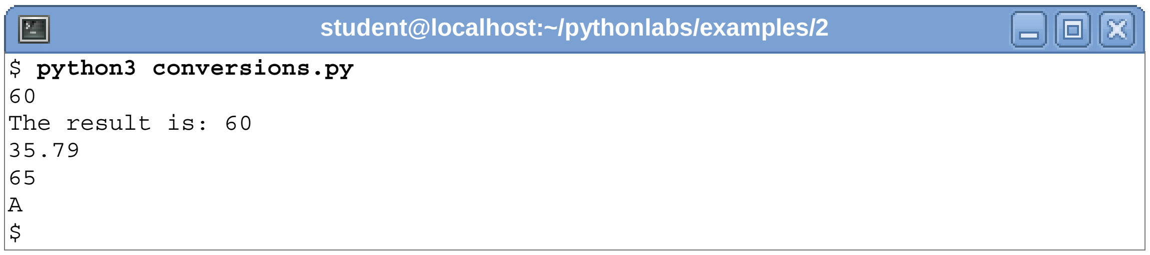
Conversion Functions

- Oftentimes you may need to treat a string as a numeric type or a number as a string within your code.
  - Python provides several functions that allow these types of conversions to be performed.
  - Some of the more common built-in conversion functions are `int()`, `float()`, `str()`, `ord()`, and `chr()`.
    - These functions are demonstrated in the program below.

`conversions.py`

```
#!/usr/bin/env python3
x = "10"
y = 20
result = int(x) + y + int("30")
print(result)
print("The result is: " + str(result))
a = 23.45
b = "12.34"
print(a + float(b))
print(ord("A"))
print(chr(65))
```

- The output of the above program is shown below.



```
$ python3 conversions.py
60
The result is: 60
35.79
65
A
$
```

Simple Output

- We have seen the `print` function in previous examples.
  - This function takes a series of arguments and outputs the data to the display.

    ```
    x = 2 + 3j        # a complex number
    print(10, "Mike", 11, "Jane", x)
    ```

- By default, the following occurs.
  - Each of the arguments will be sent to the display separated by a single space.
  - A newline is printed after the last argument.

- The programmer can alter the default behavior of the `print` method by providing the following optional named arguments.
  - `sep`
    - This is the argument separator.
    - The default is a single space.
  - `end`
    - This is the value appended at the end.
    - The default is a newline.
  - `file`
    - This is the data stream that is written to.
    - The default is the system console.

- An example is shown below.

```
print("spam", "eggs", sep=":", end="#")
```

  - Directing the output to a disk file will be shown later.

Simple Input

- The `input()` function provides a mechanism to obtain information from the user at runtime.
  - The function accepts an optional prompt string.
    - If provided, the prompt string is printed on the system console exactly as given.
    - No spaces or other formatting characters are added.
  - The `input()` function automatically removes the trailing newline from the console input prior to returning the entered data.
  - The return value of the `input()` function is always a string data type.
    - Therefore, if the entered data is to be used in a mathematical expression, a numeric conversion function must be employed.
- As an example, if we enter the following into the interactive Python shell, we get the following unexpected output.

```
>>> s = input("Enter a number: ")
Enter a number: 42
>>> t = s * 3
>>> t
'424242'
```

  - If you wish to treat the input value in a numerical way, you will need to convert it using either `int()` or `float()`.

Simple Input

- If you wanted to ask the user for a number and then display the square of the number, you could proceed as follows.

```
text = input("enter a number: " )
value = int(text)
value = value * value
print (text + " squared is ", value)
```

  - In the above code, the conversion function will raise a `ValueError` exception if the input does not represent an integer.
    - We will discuss runtime errors later in the course.

- If you wanted to input a number with a decimal point, then the relevant code would be as follows.

```
text = input("enter a number: " )
value = float(text)
```

- You could also use the `float()` and `int()` methods directly on the `input()` results.

```
value = float(input("enter a number: " ))
result = value * value
print (value, "squared is", result)
```

- Note also that you can use the `str` method to convert items to the string type.

```
print(str(value) + " squared is ", result)
```

The `%` Method

- The `str` class has a special % method that, when combined with the `print` function, can be used similarly to the `printf` functionality of the C programming language.
  - The `%` method can be used to convert various multiple arguments into a string.
    - Although the % is deprecated (not recommended for use anymore) in Python 3, it is discussed here because of its familiarity and similarity to the `printf` of the C language.
    - The recommended approach is to use the format method of the `str` class discussed earlier.
- Several of the most common formats are listed below.

Format	Resulting type
`%s`	String
`%d`	Integer
`%f`	Floating point

  - The general form of using the % method in a print statement is shown below.

```
print("formats" % (item1, item2, item3 ...))
```

    - The *`formats`* section of the print statement includes formats and text.
    - The `%` symbol separates the formats from the items to print.
    - The `item1, item2, item3 ...` section refers to the expressions and variables to be formatted.

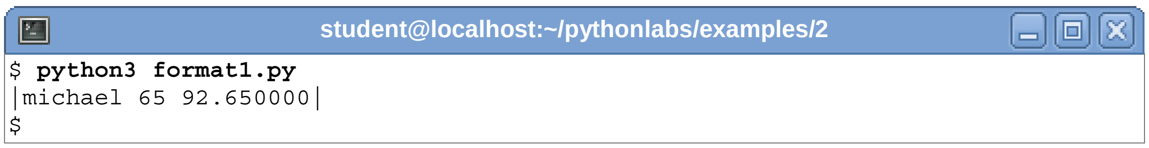
The `print` Function

- A basic example using the formatting capabilities of the `%` method within a print statement is shown below.

format1.py

```
#!/usr/bin/env python3
name = "michael"
age = 65
average = 92.65
print("|%s %d %f|" % (name, age, average))
```

- The output of the above program is shown below.



```
$ python3 format1.py
|michael 65 92.650000|
$
```

- Note the changes in the following version of the program.

format2.py

```
#!/usr/bin/env python3
name = "michael"
age = 65
average = 92.65
print("|%-10s %5d %5.2f|" % (name, age, average + 1))
```

- In the above example, the following occurs.

`-10s`	left adjustment using a field width of 10
`5d`	right adjustment with a field width of 5
`5.2f`	right adjustment with 2 decimal digits and width of 5

Exercises

1. Write an application that prompts the user to enter the radius of a circle.
    - Accept the user input into a variable.
    - Compute the area of the circle whose radius was input.
        - The formula for the area of a circle is pi times the square of the radius.
        - Use 3.14159 for pi.
2. Write a program that reads two integers, each on a separate line.
    - Print the product of the two numbers.
    - Once this works properly, try entering numbers with a decimal point.
        - What happens? Why?
    - Now try entering data that is non-numeric.
        - What happens? Why?
3. Write a program that prompts the user for a string and a number on separate lines.
    - The program should print the string replicated by the number.
        - For example, if the string is `hello` and the number is `3`, then `hellohellohello` should be printed.
4. Write a program that prompts for two numbers.
    - The first number will be the base, and the second number will be the exponent.
    - Print the result of raising the base to the exponent.

Exercises

5. Write a program that accepts a string from the user.

   - Determine the following information about the string.
     - Does it end in a period?
     - Does it contain all alphabetic characters?
     - Is there an 'x' in the string?
   - Use the `len` function to determine the length of the string.
     - Is the newline character part of the string?
   - Change all occurrences of 'e' to 'E'.

This Page Intentionally Left Blank

Chapter 3: Language Components

1) Indenting Requirements 3-2

2) The `if` Statement 3-3

3) Relational and Logical Operators 3-4

4) Bit Wise Operators 3-6

5) The `while` Loop 3-7

6) `break` and `continue` 3-10

7) The `for` Loop 3-11

Indenting Requirements

- Python provides a robust set of keywords and related items that control the flow of execution within an application.
  - In this section, we will explore the various conditional execution and looping options that Python provides.
  - In addition, we will look at the various operators used by Python in control flow constructs.
- Python mandates the use of indenting to define the statements contained in the body of a control structure.
  - Python refers to the body of a control structure as a **suite**.
    - All statements that are indented the same number of columns are part of the same suite.
    - The suite ends with the first statement that is "outdented" to the column of the header.
    - For example:

```
if someCondition:
    suiteStatement1
    suiteStatement2
    suiteStatement3
print("some output")  # suite ends here
```

  - Suites must be indented the same number of spaces from the starting column of the header.
    - If tabs are used in the source code, a single tab is not equal to the number of spaces used in a tab.
  - Typically, tools such as IDLE or Geany will automatically indent for you.

The `if` Statement

- The fundamental decision making construction in most high level programming languages is the `if` statement.
  - The following examples will demonstrate proper indenting when using the `if` statement and its variants.
  - Also, notice the required use of the colon to end the header portion of the `if` or the `else`.

if_else.py

```
#!/usr/bin/env python3
x = 0
if x == 10:
    print("x is 10")
else:
    print("x is not 10")

print("done with the if else")
```

- Note that the keyword `elif` is the Python way of saying “else if” and is useful for avoiding excessive indentation.
  - Since Python does not have a “switch” statement or a “case” statement like those found in other languages, the `elif` is a suitable substitute.

elif.py

```
#!/usr/bin/env python3
x = 5
if x <= 5:
    print("x le 5")
elif x <= 10:
    print("x between 5 and 10")
elif x <= 15:
    print("x between 10 and 15")
else:
    print("x gt 15")

print("done with the elif tests")
```

Relational and Logical Operators

- Most decisions in programming language code depend upon how one value compares to another.
  - Below is a table of the Python relational operators.

Operator	Meaning
`<`	Strictly less than
`<=`	Less than or equal
`>`	Strictly greater than
`>=`	Greater than or equal
`==`	Equal
`!=`	Not equal
`is`	Object identity
`is not`	Negated object identity

    - The `is` and `is not` operators will be discussed later.

- Python supports the following logical operators.
  - `not`
  - `and`
  - `or`
    - This means that you can write expressions consisting of compound conditions such as these.

`logicals.py`

```
#!/usr/bin/env python3
x = y = 0
if x == 0 and y == 0:
    print("x and y are zero")

if x == 0 or y == 0:
    print("x or y or both are zero")

if not ( x >= 10 and x <= 20 ):
    print("x not between 10 and 20")
```

Relational and Logical Operators

- Both the `and` and the `or` are short circuited operators.
  - This means that when the first part of an `or` evaluates as `True`, the second part is not evaluated.
  - Likewise, when the first part of an `and` evaluates `False`, the second part of the `and` is not evaluated.
- The relational and logical operators yield results of either `True` or `False`.
  - You can write Python statements that check for these values.

```
x = 10
y = 20
z = x < y
if z:
    print("True")
```

- In addition to the literal values `True` and `False`, there are other expressions that evaluate as follows.
  - `True`
    - Any non-zero value
  - `False`
    - `None`
    - `0`
    - Empty sets, dictionaries, tuples, or lists
    - Empty strings

Bit Wise Operators

- Python also has a rich set of bit wise operators.
  - Admittedly, you can do a lot of programming without ever needing these operators, but it is nice to know they are there when you need them.
  - The list of the bitwise operators and their meanings is shown below.

Operator	Meaning
`x << y`	Returns `x` with the bits shifted to the left by `y` places (and new bits on the right-hand-side are zeros). This is the same as multiplying `x` by `2**y`.
`x >> y`	Returns `x` with the bits shifted to the right by `y` places. This is the same as dividing `x` by `2**y`.
`x & y`	Does a "bitwise and". Each bit of the output is `1` if the corresponding bit of `x` AND of `y` is `1`, otherwise it is `0`.
`x | y`	Does a "bitwise or". Each bit of the output is `0` if the corresponding bit of `x` AND of `y` is `0`, otherwise it is `1`.
`~x`	Returns the complement of `x` - the number you get by switching each `1` for a `0` and each `0` for a `1`. This is the same as `-x - 1`.
`x ^ y`	Does a "bitwise exclusive or". Each bit of the output is the same as the corresponding bit in `x` if that bit in `y` is `0`, and it is the complement of the bit in `x` if that bit in `y` is `1`.

- When working at the bit level within a program, it is often convenient to encode constants in bases other than decimal.

```
x = 0xff        # 255
y = 0o77        # 63
z = 0b111       # 7
print(x, y, z) # 255  63  7
```

  - The `hex()`, `oct()`, and `bin()` functions can also be used to convert numbers to bases other than decimal.
    - The return value of each function is of type string.

The `while` Loop

- The `while` statement causes Python to loop through a suite of statements if the test condition evaluates to `True`.
    - The `while` statement uses the same evaluations for `True` and `False` as the `if` statement.
    - Likewise, the same indentation rules are used for a `while` as they are for an `if`.
- Here is an example program that utilizes a `while` loop to add the integers from 1 to 10.

sum.py

```
#!/usr/bin/env python3
""" Add the integers from 1 to 10 """
count = 1
total = 0

while count <= 10:
    total += count
    count += 1

print(total)
```

- If the condition in the `while` is true, then the suite of statements in the `while` loop is executed.
    - This is followed by a re-test of the condition.
    - When the condition is false, then the first statement after the while loop is executed.

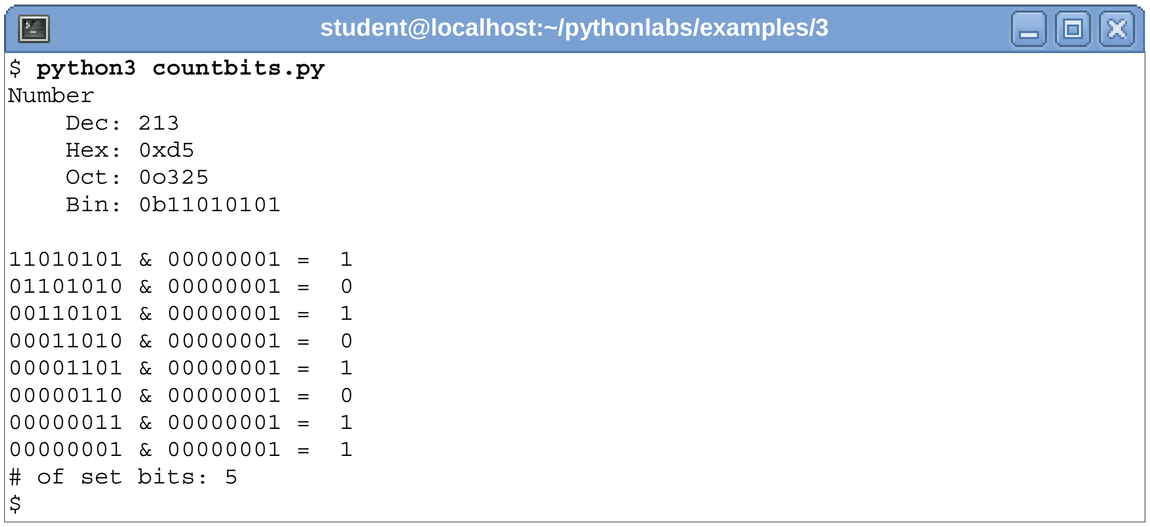
The `while` Loop

- Here is another `while` loop.

countbits.py

```
#!/usr/bin/env python3
""" Count the bits that are set in a variable """

count  = 0
value = 213
print("Number")
print("    Dec:", value)
print("    Hex:", hex(value))
print("    Oct:", oct(value))
print("    Bin:", bin(value))
print()
while value > 0:
    result = value & 1
    fmt = "{0:08b} & {1:08b} =  {2}"
    txt = fmt.format(value, 1, result)
    print(txt)

    if value & 1:   # bitwise AND
        count += 1
    value = value >> 1 # bit shift right

print("# of set bits:", count)
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/3

```
$ python3 countbits.py
Number
    Dec: 213
    Hex: 0xd5
    Oct: 0o325
    Bin: 0b11010101

11010101 & 00000001 =  1
01101010 & 00000001 =  0
00110101 & 00000001 =  1
00011010 & 00000001 =  0
00001101 & 00000001 =  1
00000110 & 00000001 =  0
00000011 & 00000001 =  1
00000001 & 00000001 =  1
# of set bits: 5
$
```

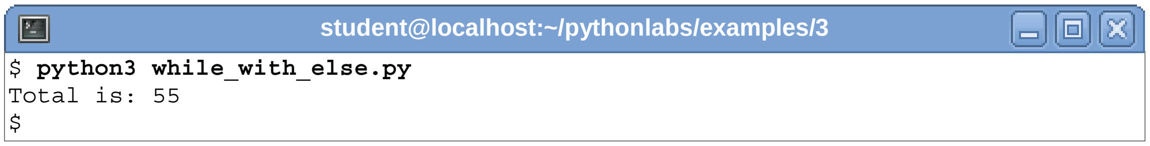
The `while` Loop

- Python also permits a `while` loop to have an `else` clause.
  - If present, the `else` clause is always executed whenever the loop test fails.

`while_with_else.py`

```
#!/usr/bin/env python3
counter = 1
total = 0
while counter <= 10:
    total += counter
    counter += 1
else:
    print("Total is:", total)
```

  - The output of the above program is shown below.



```
$ python3 while_with_else.py
Total is: 55
$
```

- It is important to note that the `else` clause is executed when the `while` test fails.
  - While this might seem inevitable, we will see on the next page that there are additional ways of exiting a loop.
    - Under these conditions the exit would not be because of the test failing and, as such, the `else` clause would not be executed.

`break` and `continue`

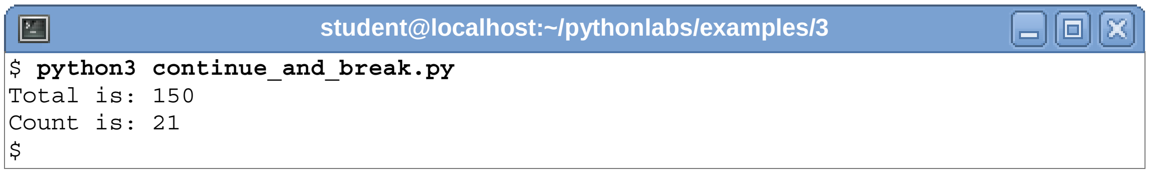
- Any looping construct can also have its control changed through a `break` or `continue` statement within it.
  - When a `break` is executed, control of the program jumps to the first statement beyond the loop.
    - A break is often used when searching through a collection for the occurrence of a particular item.
  - When a `continue` statement is executed in a loop, the rest of the suite is skipped and control goes to the next iteration of the looping construct.

- The example below incorporates both `break` and `continue` statements.

continue_and_break.py

```
#!/usr/bin/env python3
cnt = 0
total = 0
while cnt <= 100:
    cnt += 1
    if cnt % 4 == 0:
        continue          # skip even multiples of 4
    if cnt * cnt >  400:
        break             # will happen at cnt = 21
    total += cnt
else:
    print("Might not get here")
print("Total is:", total)
print("Count is:", cnt)
```

  - The output of the above program is shown below.



```
$ python3 continue_and_break.py
Total is: 150
Count is: 21
$
```

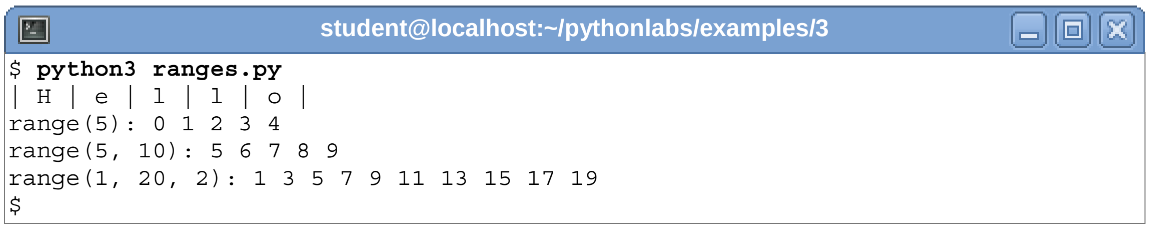
The `for` Loop

- The `for` loop in Python is used to iterate over the items of any sequence, such as a list or a string.
  - You can also use the `range` function to generate a sequence of numbers and then iterate over them with a `for` loop.
    - The example below demonstrates using `for` loops to loop through a string and several sequences created via the `range` function.

ranges.py

```
#!/usr/bin/env python3
txt = "| "
for character in "Hello":
    txt += character + " | "
print(txt)

txt = "range(5): "
for i in range(5):
   txt += str(i) + " "
print(txt)

txt = "range(5, 10): "
for i in range(5, 10):
   txt += str(i) + " "
print(txt)

txt = "range(1, 20, 2): "
for i in range(1, 20, 2):
   txt += str(i) + " "
print(txt)
```

  - The output of the above program is shown below.



```
$ python3 ranges.py
| H | e | l | l | o |
range(5): 0 1 2 3 4
range(5, 10): 5 6 7 8 9
range(1, 20, 2): 1 3 5 7 9 11 13 15 17 19
$
```

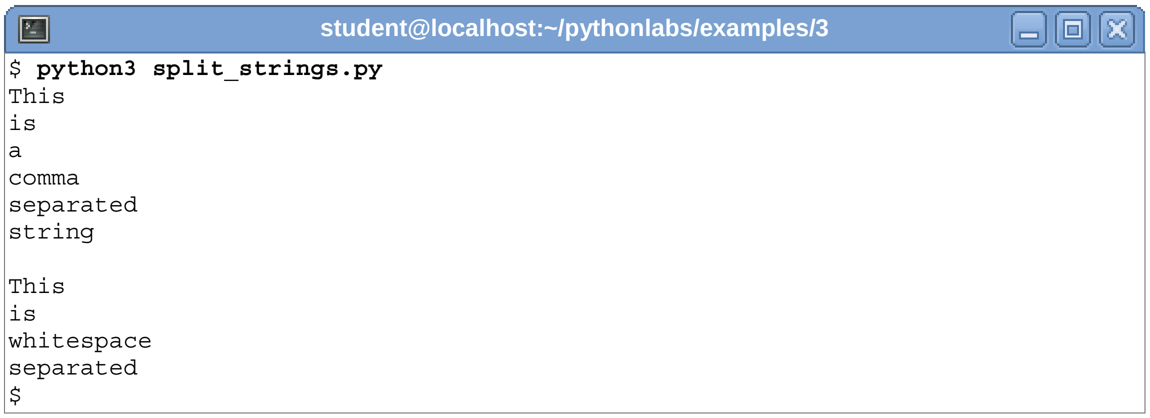
The `for` Loop

- Lists are another type of sequence that a `for` loop is often used to loop through.
  - While a formal discussion of lists does not come until the next chapter, recall that the `split` method of a string returns a list of strings.
    - The split is passed an optional argument indicating the delimiter to use when splitting the string.
    - If no argument is passed, whitespace is used as the delimiter.

`split_strings.py`

```
#!/usr/bin/env python3
txt1 = "This,is,a,comma,separated,string"
txt2 = "This    is whitespace\t\tseparated"
pieces = txt1.split(",")
for piece in pieces:
    print(piece)
print()

pieces = txt2.split()
for piece in pieces:
    print(piece)
```

- The output of the above program is shown below.



```
$ python3 split_strings.py
This
is
a
comma
separated
string

This
is
whitespace
separated
$
```

Exercises

1. Write a program that prompts the user to enter two integers, each on a separate line.
   - The program should print the larger of the two numbers.
   - If the numbers are equal, the program should confirm it.
2. Write a program that asks the user to input two numbers.
   - The program should output the sum of the integers between those two numbers.
     - For example, if the user inputs the numbers 10 and 15, then the sum would be 75.

       ```
       10 + 11 + 12 + 13 + 14 + 15 = 75
       ```
3. Ask the user to input a set of numbers on one input line.
   - Split the numbers into a list.
   - Write a loop that examines each element of the list and displays the ones that are greater than zero.
4. Ask the user to input three numbers representing a lower limit, a higher limit, and a step value.
   - The program should loop through and print the numbers from low to high, taking into consideration the step.
     - A `for` loop and `range` function should be used for this exercise.
5. Produce a table of the integers from 0 to 49.
   - Try to get 10 items per row as shown below.

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
 30  31  32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47  48  49
```

This Page Intentionally Left Blank

Chapter 4: Collections

1) Introduction **4-2**

2) Lists **4-3**

3) Tuples **4-6**

4) Sets **4-7**

5) Dictionaries **4-10**

6) Sorting Dictionaries **4-14**

7) Copying Collections **4-17**

8) Summary **4-19**

Introduction

- Python's provides the following general purpose built-in collection classes.
  - `list`
    - An ordered collection of elements
    - Similar to an array in other languages
    - Dynamic in that it can grow or shrink depending upon the needs of the application
  - `tuple`
    - An immutable `list`
  - `set`
    - An unordered collection of elements
    - Does not permit duplicates
  - `dictionary`
    - An unordered collection of key value pairs
    - Keys have to be unique
- The `list` and `tuple` classes are also referred to as sequences (similar to a Python string) because you can refer to one item or more by using subscripts (slices).
- The following built in functions can be used to simplify the conversion of one type of collection to another.
  - `list()`, `tuple()`, `set()`, and `dict()`

Lists

- The `list` type is a container that holds a number of other objects in a given order.
  - A `list` allows the addition and removal objects.

- A `list` can be created in several ways.
  - One way is to create an empty list using the `list()` function.

```
list1 = list()      # Length is 0
print(len(list1))
```

  - Another way to do this is by using a set of square brackets and placing one or more comma-separated items within them.
    - The elements can be of any type.

```
list2 = [ ]     # Length is 0
print(len(list2))

list3 = [20, 10, 40, 60, 50, 3] # Length is 6
print(len(list3))
```

- The `[]` operator can be used to obtain a slice of a `list`.

```
a = list3[0]      # 20
b = list3[-1]     # 3
c = list3[0:2]    # [20, 10]
d = list3[:3]     # [20, 10, 40]
e = list3[-3:-1]  # [60, 50]
f = list3[-3:]    # [60, 50, 3]
```

- Unlike arrays in more traditional languages, a `list` is dynamic (it can grow and/or shrink as needed).

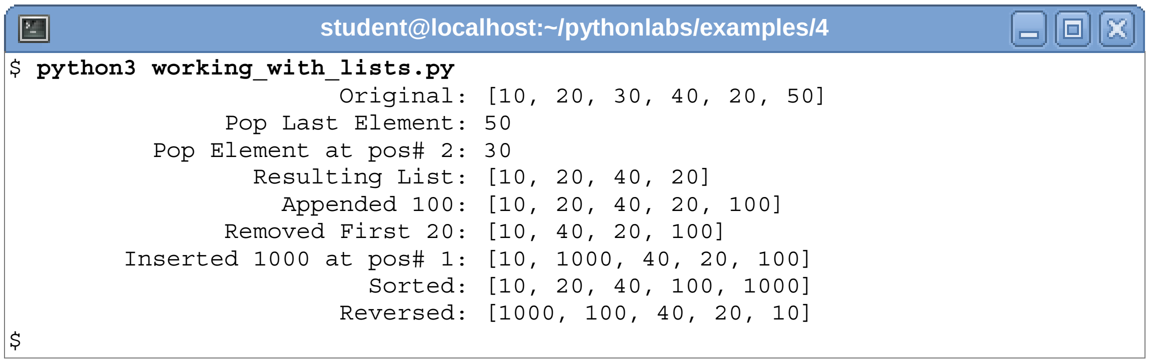
Lists

- Adding and deleting elements from a `list` are common operations.
  - Here are a few examples of some methods that modify the contents of a `list`.

working_with_lists.py

```
#!/usr/bin/env python3
lis = [10, 20, 30, 40, 20, 50]
fmt = "%32s %s"
print(fmt % ("Original:", lis))

print(fmt % ("Pop Last Element:", lis.pop()))
print(fmt % ("Pop Element at pos# 2:", lis.pop(2)))
print(fmt % ("Resulting List:", lis))

lis.append(100)
print(fmt % ("Appended 100:", lis))
lis.remove(20)
print(fmt % ("Removed First 20:", lis))
lis.insert(1, 1000)
print(fmt % ("Inserted 1000 at pos# 1:", lis))

lis.sort()
print(fmt % ("Sorted:", lis))
lis.reverse()
print(fmt % ("Reversed:", lis))
```

  - The output of the above program is shown below.



```
$ python3 working_with_lists.py
                       Original: [10, 20, 30, 40, 20, 50]
               Pop Last Element: 50
          Pop Element at pos# 2: 30
                 Resulting List: [10, 20, 40, 20]
                   Appended 100: [10, 20, 40, 20, 100]
               Removed First 20: [10, 40, 20, 100]
        Inserted 1000 at pos# 1: [10, 1000, 40, 20, 100]
                         Sorted: [10, 20, 40, 100, 1000]
                       Reversed: [1000, 100, 40, 20, 10]
$
```

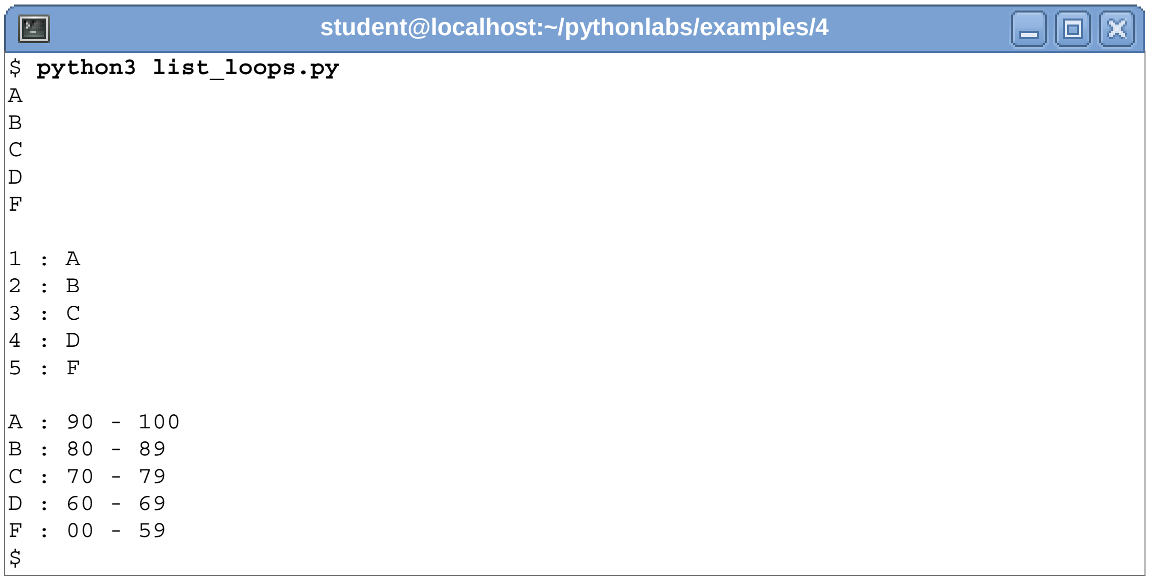
Lists

- Looping through a list is typically accomplished through the use of a simple `for` loop.
  - The built-in `enumerate()` function can be helpful when both the index and the item at that index are desired.

list_loops.py

```python
#!/usr/bin/env python3
grades = ["A", "B", "C", "D", "F"]
ranges = ["90 - 100", "80 - 89", "70 - 79",
          "60 - 69", "00 - 59"]
for grade in grades:
    print(grade)
print()

for i, grade in enumerate(grades):
    print(i + 1, ":", grade)
print()

for index, grade in enumerate(grades):
    print(grade, ":", ranges[index])
```

  - The output of the above program is shown below.



```
$ python3 list_loops.py
A
B
C
D
F

1 : A
2 : B
3 : C
4 : D
5 : F

A : 90 - 100
B : 80 - 89
C : 70 - 79
D : 60 - 69
F : 00 - 59
$
```

Tuples

- A `tuple` is an ordered immutable collection.
  - A `tuple` is somewhat like a `list`, but it is typically more efficient since it cannot shrink, grow, or be changed in any other way.
  - All of the non-mutable operations that can be performed on a `list` can also be performed on a `tuple`.
    - However, there are some functions that require a `list` as a parameter and not a `tuple`.
    - The conversion functions, `list()` and `tuple()`, provide an easy way of converting one collection to the other.
- The example below demonstrates working with a `tuple` and converting it to a `list` in order to sort its contents.

working_with_tuples.py

```
#!/usr/bin/env python3
x = 10;
y = 20;
data = (x, y, 5, 30, 7) # creating a tuple
print("tuple contents:", data)
print("a slice:", data[2:4])

# converting tuple to list
as_list = list(data)
as_list.sort()
print("List:", as_list)

# converting list to tuple
data = tuple(as_list)
print("tuple contents", data)
```

Sets

- A set is an unordered collection with no duplicates.
  - A set can be created by using the `set()` function.

    ```
    s = set()
    ```

  - A `set` can also be initialized by passing a comma-separated collection of values inside the { } symbols.
    - Any duplicate entries will be removed when the `set` is created.

      ```
      mySet = {'apple', 'pear', 'banana', 'apple'}
      print(len(mySet))    # 3
      ```

- If the `set()` function is passed a string, it will create a `set`, with each character being an element of the `set`.

  ```
  mySet =  set('mississippi')
  print(len(mySet))    # 4
  ```

- The `set()` function can also be used to convert a list or a tuple into a set.

  ```
  aList = ['fe', 'fi', 'fo', 'fum', 'fi']
  mySet = set(aList)
  print(len(mySet))        # 4
  ```

Sets

- Sets are often used for membership testing using the `in` operator.

```
fruit = {'apple', 'orange', 'pear', 'kiwi'}
print('orange' in fruit)    # True
print('crabgrass' in fruit) # False
```

  - A commonly used test might be as follows.

```
odds = { 1, 3, 5, 7, 9 }
item = int(input("enter a digit: "))
if ( item in odds ):
    print("Yes")
```

- The following methods can also be used for sets.
  - The `add(value)` method adds an element to a `set`.
  - The `remove(value)` method removes an element from a `set`.
    - This raises a `KeyError` exception if element is not present.
  - The `discard(value)` method also removes an element from a `set`.
    - This method does not raise any exception.
  - The `clear` method removes all the elements from a `set`.

- You can determine if a set is a subset or superset of another with the following methods.

```
x = { 1, 2, 3, 4, 5 }
y = { 2, 3, 4 }
print(x.issuperset(y))  # True
print(y.issubset(x))    # True
```

  - Each of the methods returns either `True` or `False`.

Sets

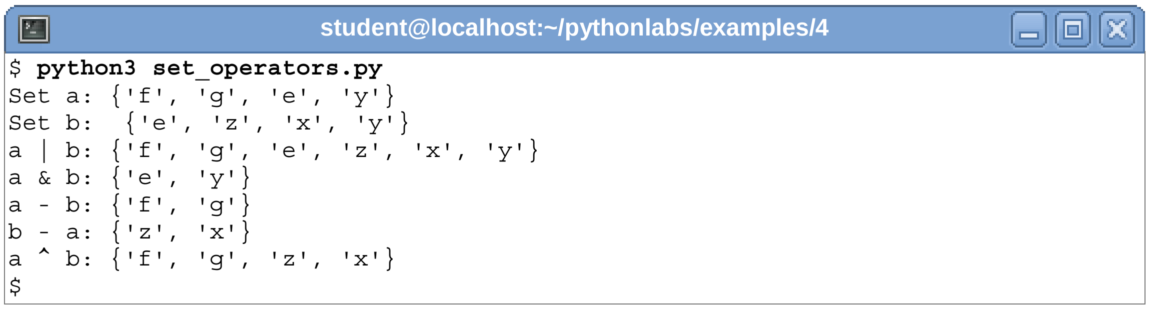
- The following mathematical set operators are available, as well, when working with a `set`.

Operator	Name	Meaning
|	Union	An element is in one set or the other set.
&	Intersection	An element is in one set and the other set.
-	Difference	An element is in the first set but not in the second set.
^	Symmetric Difference	An element is in only one set and not both sets.

set_operators.py

```
#!/usr/bin/env python3
a = set('efgy')
b = set('exyz')
print("Set a:", a)
print("Set b:  ", b)

print("a | b:", a | b )   # union
print("a & b:", a & b )   # intersection
print("a - b:", a - b )   # difference
print("b - a:", b - a )   # difference
print("a ^ b:", a ^ b )   # symmetric difference
```

- The output of the above program is shown below.



```
$ python3 set_operators.py
Set a: {'f', 'g', 'e', 'y'}
Set b:  {'e', 'z', 'x', 'y'}
a | b: {'f', 'g', 'e', 'z', 'x', 'y'}
a & b: {'e', 'y'}
a - b: {'f', 'g'}
b - a: {'z', 'x'}
a ^ b: {'f', 'g', 'z', 'x'}
$
```

Dictionaries

- A `dict` in Python (typically referred to as a dictionary) is an unordered collection of entries.
  - Each entry contains a key-value pair.
  - Other languages refer to this data type as a hash, map, or associative array.
- A `dict` can be created in several ways.

```
map1 = {'key1':'val1', 'key2':'val2'}

map2 = dict([['key1','val1'], ['key2','val2']])

map3 = dict(key1='val1', key2='val2')

map4 = {}
```

- The keys in a dictionary are unique.
  - Note the following.

```
map4['mike'] = 10
map4['mike'] = 20
print(map4['mike'])         # 20
```

- Dictionaries are intended as fast lookup data structures.
  - Given a key, its corresponding value can be fetched directly, as shown here.

```
map5 = {'mike': 15, 'susan': 19, 'chris': 21}
value1 = map5['mike']       # 15
value2 = map5['susan']      # 19
```

  - A `KeyError` will be generated if a key is used that is not present in the dictionary.

```
value3 = map5['dave']    #KeyError
```

Dictionaries

- As an alternative to using the `[]` operator to retrieve a value for a given key, Python also provides the `get()` method.
  - The `get()` method returns the value for a given key.

    ```
    map6 = {'mike': 15, 'susan': 19, 'chris': 21}
    value1 = map6.get('mike')      # 15
    value2 = map6.get('susan')     # 19
    ```

  - What makes it different from the `[]` operator is that the `get()` method does not generate a `KeyError` if the key used does not exist.
    - Instead, it returns the special value `None` by default if a key is not present.

      ```
      value3 = map5.get('dave')    # None
      ```

    - If a second argument is passed to the `get()` method, it will use that argument as the return value if the key is not present.

      ```
      value4 = map5.get('sally', 99)    # 99
      ```

  - The example below demonstrates the use of the `get()` method and returns a value of `0` for any key that does not exist.

working_with_dictionaries.py

```
#!/usr/bin/env python3
data = {'mike': 15, 'susan': 19, 'chris': 21}
while True:
    key = input("Please enter a key: ")
    value = data.get(key, 0)
    if key.lower() == "quit":
        break
    if value:
        print("Value for ", key, "=", value)
    else:
        print(key, "is not present")
```

Dictionaries

- Removing elements from a dictionary is done through the `pop()` and `popitem()` methods.
  - The general syntax for the `pop()` method is as follows.

    ```
    value = obj.pop(key[,default])
    ```

    - *`obj`* represents the dictionary from which to remove something.
    - *`key`* represents the key of the pair to be removed.
    - *`default`* represents an optional value to return if *`key`* does not exist. A `KeyError` is raised if no *`default`* is given and the *`key`* does not exist.
  - The general syntax for the `popitem()` method is as follows.

    ```
    a_tuple = obj.popitem()
    ```

    - *`obj`* represents the dictionary from which to remove something.
    - The `popitem()` method remove and returns some (key, value) pair as a 2-`tuple`, but raises a `KeyError` if `obj` is empty.

`removing_from_dictionary.py`

```
#!/usr/bin/env python3
cars = {'Mustang':'Ford', 'Falcon':'Ford',
        'Camaro':'Chevy', 'Corvette':'Chevy',
        'Eclipse':'Mitsubishi', 'Integra':'Acura'}
unknown = "Make is Unknown"
make = cars.pop("Corvette")
print(make)
make = cars.pop("Accord",unknown)
print(make)
a_tuple = cars.popitem()
print(a_tuple[0], a_tuple[1])
model, make = cars.popitem()
print(model, make)
print(cars)
```

Dictionaries

- When there is a need to loop through a dictionary, the following methods are useful.
  - The `keys()` method returns a set-like object providing a view of the dictionary keys.

```
grades = {'mary':99, 'john':75, 'anne':89}
for student in (grades.keys()):
    print(student)
```

  - The `values()` method returns an object providing a view of the dictionary values.

```
grades = {'mary':99, 'john':75, 'anne':89}
for grade in (grades.values()):
    print(grade)
```

  - There is also an `items()` method that returns a set-like object providing a view of the dictionary items.

```
grades = {'mary':99, 'john':75, 'anne':89}
for student, grade in (grades.items()):
    print(student, grade)
```

    - If the list of keys is available, retrieving the corresponding values could also be done as follows.

```
grades = {'mary':99, 'john':75, 'anne':89}
for student in (grades.keys()):
    print(student, grades[student])
```

- If you need to determine how many pairs exist in a dictionary, the `len()` function can be used as follows.

```
howmany = len(theMap)
```

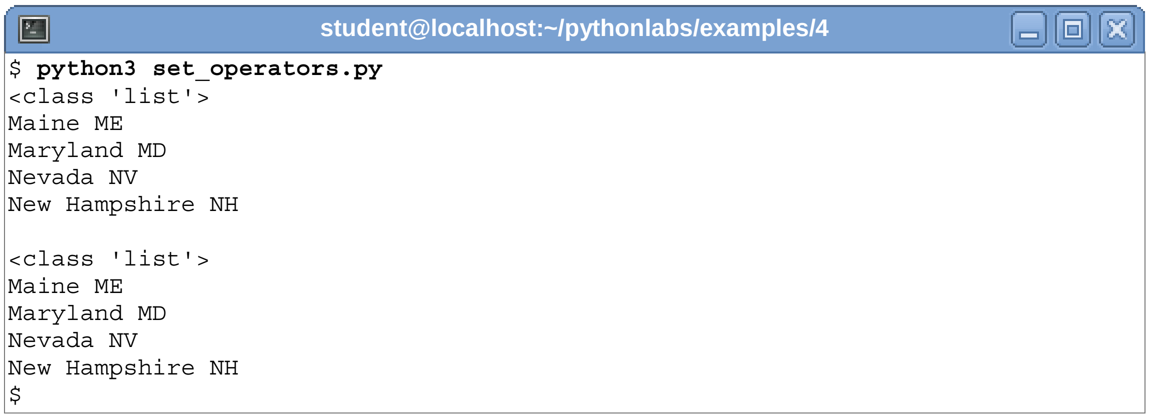
Sorting Dictionaries

- Sorting a dictionary can be done in several ways.
  - Sorting alphabetically by keys can be accomplished as follows.
    - Convert the return value of the dictionary's `keys()` method into a list, and then sort the list of keys.
  - Python also provides a built-in `sorted()` function that returns an alphabetically sorted list of the keys within the dictionary passed as an argument.

`sort_by_keys.py`

```
#!/usr/bin/env python3
states = {'New Hampshire':'NH', 'Maryland':'MD',
          'Nevada':'NV', 'Maine':'ME'}

long_names = list(states.keys())
print(type(long_names))
long_names.sort()
for name in long_names:
    print(name, states[name])
print()

result = sorted(states)
print(type(result))
for name in result:
    print(name, states[name])
```

- The output of the above program is shown below.



```
$ python3 set_operators.py
<class 'list'>
Maine ME
Maryland MD
Nevada NV
New Hampshire NH

<class 'list'>
Maine ME
Maryland MD
Nevada NV
New Hampshire NH
$
```

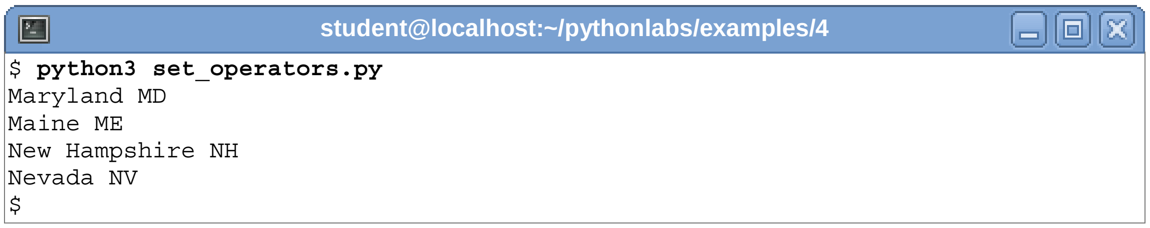
Sorting Dictionaries

- Sorting by values is a little more difficult.
  - It brings to mind the idea of various sorting criteria.
    - Sorting algorithms proceed in two phases, 1) the comparison phase, and 2) the exchange phase.
    - As comparisons are made, data in the data structure is either exchanged or not exchanged depending upon the nature of the comparison and whether the data items being compared are out of order or not.
- In order to sort by values, you have to instruct the `sort()` method to call a special function that indicates how to compare the items being sorted.
  - The code below is an example of such a custom sort.

sort_by_values.py

```
#!/usr/bin/env python3
def usethis(akey):
    return states[akey]

states = {'New Hampshire':'NH', 'Maryland':'MD',
          'Nevada':'NV', 'Maine':'ME'}

long_names = list(states.keys())
long_names.sort(key=usethis)
for name in long_names:
    print(name, states[name])
```

- The output of the above program is shown below.



```
$ python3 set_operators.py
Maryland MD
Maine ME
New Hampshire NH
Nevada NV
$
```

Sorting Dictionaries

- In the code from the previous page, the sort routine takes an argument that carries the name of a function.

  ```
  long_names.sort(key=usethis)
  ```

  - This function is called by the sort routine to compare the keys and exchange them if necessary so they reflect the sorted order of the values.

    ```
    def usethis(akey):
        return states[akey]
    ```

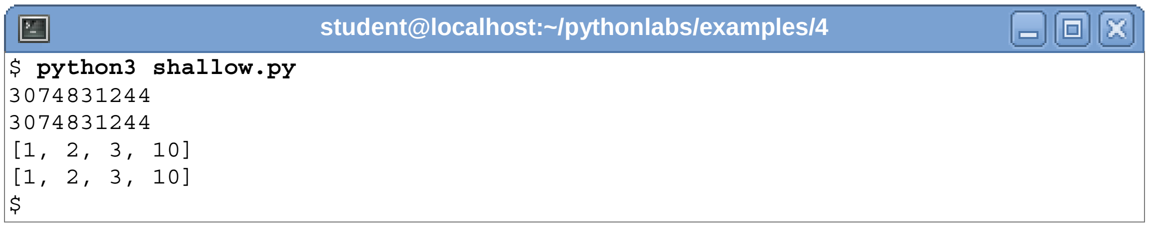
Copying Collections

- The Python `id` function returns the unique id of an object.
  - You can use this function to prove that the following code does not create a second list but rather another reference to the original list.

shallow.py

```
#!/usr/bin/env python3
a = [ 1, 2, 3 ]     # create a list
b = a               # shallow copy
print(id(a))
print(id(b))

a.append(10)
print(a)
print(b)
```

  - The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/4

```
$ python3 shallow.py
3074831244
3074831244
[1, 2, 3, 10]
[1, 2, 3, 10]
$
```

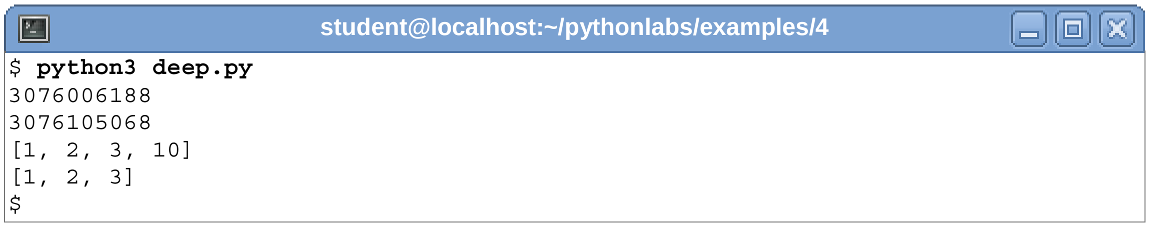
- An actual copy can be made by using the `list()` function.

deep.py

```
#!/usr/bin/env python3
a = [ 1, 2, 3 ]     # create a list
b = list(a)         # deep copy
print(id(a))
print(id(b))

a.append(10)
print(a)
print(b)
```

Copying Collections

- The output of the previous program is shown below.



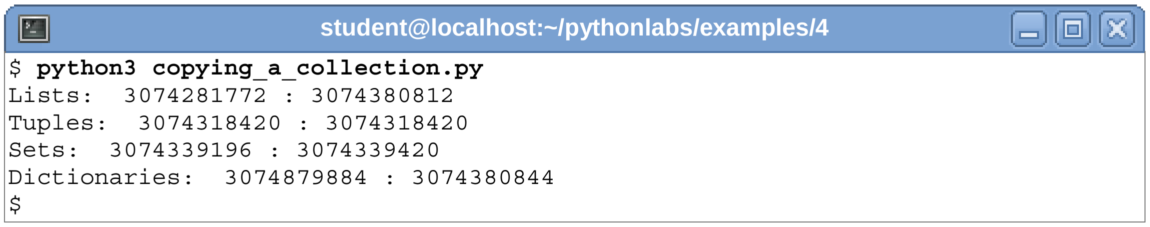
```
$ python3 deep.py
3076006188
3076105068
[1, 2, 3, 10]
[1, 2, 3]
$
```

- Things are similar with the other collection data types.

`copying_a_collection.py`

```
#!/usr/bin/env python3
list1 = [ 1, 2, 3 ]
list2 = list(list1)
print("Lists: ", id(list1), ":", id(list2))

tuple1 = (4, 5, 6)
tuple2 = tuple(tuple1)
print("Tuples: ", id(tuple1), ":", id(tuple2))

set1 = {7, 8, 9}
set2 = set(set1)
print("Sets: ", id(set1), ":", id(set2))

dict1 = {"A":10, "B":11, "C":12}
dict2 = dict(dict1)
print("Dictionaries: ", id(dict1), ":", id(dict2))
```

  - The output of the above program is shown below.



```
$ python3 copying_a_collection.py
Lists:  3074281772 : 3074380812
Tuples:  3074318420 : 3074318420
Sets:  3074339196 : 3074339420
Dictionaries:  3074879884 : 3074380844
$
```

Summary

- A `list` is a dynamic mutable ordered collection of elements.
  - A `list` is like an array in other languages.

```
lis = []           # 0 elements
lis = [1, 10, 3]   # 3 elements
```

- A `tuple` is an immutable ordered `list`.
  - Tuples are more efficient than lists.

```
t1 = ()      # 0 elements
t2 = (1,2,3) # 3 elements
```

- A `set` is a mutable unordered collection of non-duplicate elements.
  - They are mostly used to:
    - determine if a particular item is among them; or
    - streamline a set of values by casting away duplicates.

```
s1 = set()          # 0 elements
s2 = {'a', 'b', 'a'} # 2 elements
```

- A `dictionary` is a mutable collection of key-value pairs.
  - The keys are unique.
  - Keys can be used to retrieve associated values.

```
d1 = {}                # 0 elements
d2 = {'a':1, 'b':2 }   # 2 pairs
```

- All collections be passed to the `len` function.

- You can use a `for` loop with any collection.

Exercises

1. Write a program that asks the user to input a number.
   - Repeat this process until the user enters the value "end."
   - Enter each number into a set.
     - However, before you enter the number, check to see if the number is in the set.
   - Just before the program ends, print the set and the number of elements that were NOT added to the set.
2. Use a set to determine the number of unique words in the user's input.
   - Use the following to loop through the user's input lines.

```
while True:
    data = input("enter a line (q to quit) ")
    if data == 'q':
        break
    #
    # process your lines here
    # linewords = data.split()
```

   - Output the unique elements sorted alphabetically!
   - Also, output the number of unique words.
3. Use a dictionary to create a mapping from the digits 0-9 to the words 'zero', 'one,' 'two,' etc.
   - Then, ask the user to input a number.
   - If the user enters `1437`, then the program should print "`one four three seven`."

Exercises

4. Rewrite Exercise 2 but this time count the frequency of each word in the user's input.
   - A dictionary provides the perfect data structure for this problem.
     - Let the words be the keys, and let the counts be the values.
   - Print the results sorted by the words.
   - Then, print the results sorted by the counts.

This Page Intentionally Left Blank

Chapter 5: Functions

1) Introduction 5-2

2) Defining Your Own Functions 5-3

3) Parameters 5-4

4) Function Documentation 5-5

5) Keyword and Optional Parameters 5-6

6) Passing Collections to a Function 5-7

7) Variable Number of Arguments 5-8

8) Scope 5-11

9) Functions - "First Class Citizens" 5-13

10) Passing Functions to a Function 5-14

11) `map` 5-15

12) `filter` 5-16

13) Mapping Functions in a Dictionary 5-17

14) Lambda 5-18

15) Inner Functions 5-19

16) Closures 5-20

Introduction

- Large programs are more easily understood, maintained, and debugged if they are divided into manageable reusable pieces.
  - One way to do this is by using functions.
    - A function represents a code body that can be written once and then executed whenever the need arises.
    - The function might be encoded in the program that needs it, or it may be located in a separate file, imported, and used in any program that needs it.
- Functions can also be grouped together into a library and reused over a series of applications by the entire enterprise.
  - Libraries in Python are called **modules**.
    - Modules will be discussed in the next chapter.
- We have already seen many functions that are part of the standard Python distribution.
  - Below are just some of the functions that have been used.
    - `input()`
    - `print()`
    - `len()`
    - `hex()`
    - `oct()`
    - `bin()`
    - `int()`
    - `float()`
    - `str()`
    - `sorted()`
    - `type()`
    - `id()`
- However, you can always write your own functions, and this chapter explores the details of doing that.

Defining Your Own Functions

- A function is created using the keyword `def`.
  - Following that keyword is the name of the function and a set of parenthesis.
    - Below is a simple example.

```
def printmsg():
    print("This function simply")
    print("prints this message")
```

- A function definition, like the one above, should precede the actual invocation of the function.
  - Here is a complete program that uses the `printmsg` function.

first_function.py

```
#!/usr/bin/env python3
def printmsg():
    print("This function simply ", end="")
    print("prints this message")

printmsg()
```

  - The definition of a function must precede its invocation.

- In general, functions are more useful when you can send data to them and have the function operate on the data.
  - Values sent to functions are called arguments.
  - Each argument that is sent to the function must have a corresponding parameter defined in the header part of the function.
    - It is important to recognize that an argument that is of type string, tuple, or number cannot be altered by a change to its corresponding parameter.
    - This is due to them being immutable.

Parameters

- Suppose we needed a function that computed a base raised to a power.
  - In order to invoke such a function (should it exist), we might proceed as follows.

    ```
    b = int(input("enter base: "))
    e = int(input("enter exponent: "))
    result = power(b, e)
    print(result)
    ```

- Data items sent to a function are known as arguments.
  - In the above example, the variables `base` and `exp` are arguments.
- Functions require as many parameters as arguments.
  - Therefore, in this case there will be two parameters.
    - Parameter names are unimportant.
    - Here they are called `base` and `expo`.

      ```
      def power(base, expo):
          answer = 1
          while(expo > 0 ):
              answer *= base
              expo -= 1
          return answer
      ```

  - When the above function is invoked, the values of the arguments are passed to the parameters.
  - Notice also that the value computed by the function is returned to the call site (i.e., the place where the function was called).
    - If no `return` is specified, a function returns the value `None`, by default.

Function Documentation

- If the first statement within the suite of a function begins with `"""` characters, then everything between them and the ending `"""` characters is interpreted as a multi-line documentation string.
  - Furthermore, the documentation string can be displayed with either of the following.

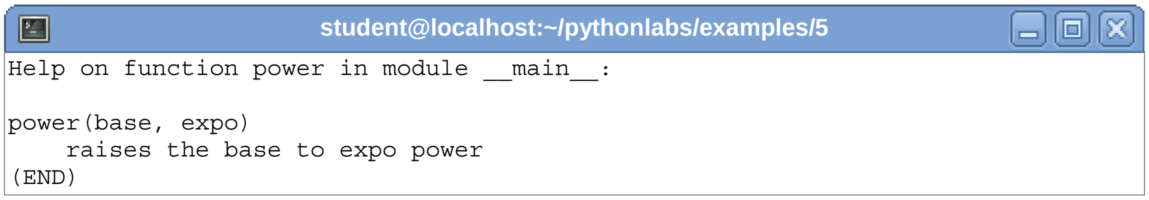
    ```
    help(function_name)
    print(function_name.__doc__)
    ```

- Notice the output from the following program.

power.py

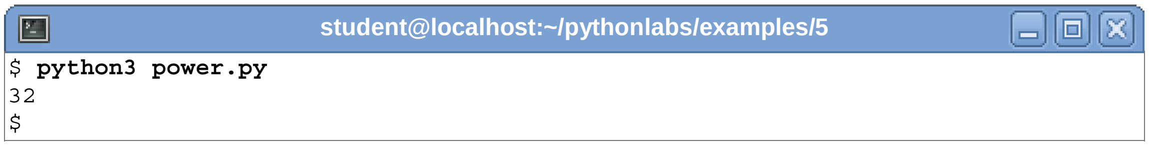
```
#!/usr/bin/env python3
def power(base, expo):
    """ raises the base to expo power """
    answer = 1
    while expo > 0:
        answer *= base
        expo -= 1
    return answer

help(power)
print(power(2,5))
```

- The output of the above program is shown below.



```
Help on function power in module __main__:

power(base, expo)
    raises the base to expo power
(END)
```

- Typing "q" above then displays the following.



```
$ python3 power.py
32
$
```

Keyword and Optional Parameters

- Python makes it easy to pass arguments to functions.
  - You can always pass arguments in the traditional way.

    ```
    answer = power(b, e)
    ```

- Python also lets you pass named arguments to a function.
  - Therefore, you can invoke it with either of the following.
    - `answer = power(expo=e, base=b)`
    - `answer = power(base=b, expo=e)`
  - The advantage in the named approach is that the arguments can be passed in either order.
    - An additional benefit is that since the parameters are named, their purpose is well known to whoever is maintaining the code.

- A function can also have default values for parameters.

  ```
  def volume(length = 10, width = 5, height = 2):
      return length * width * height
  ```

  - With the above default values, the following are legitimate calls to `volume`.

    ```
    • print(volume())            # 100
    • print(volume(20))          # 200
    • print(volume(20, 20))      # 800
    • print(volume(20, 20, 20)) # 8000
    • print(volume(30, width=25, height=20))
    ```

  - Whenever keyword or optional parameters are used, they must be the rightmost in the list.

Passing Collections to a Function

- Here is a function that expects a list (or any collection) as an argument and returns the sum of its elements.

pass_list.py

```
#!/usr/bin/env python3
def thesum(param):
    total = 0
    for elem in param:
        total += elem
    return total

data = [ 10, 20, 30, 40 ]
x = thesum(data)
print(x)

vals = [ ]
for i in range(10):
    vals.append(i)
x = thesum(vals)
print(x)
```

- In the following example, a dictionary is passed.

pass_dict.py

```
#!/usr/bin/env python3
def thesum(param):
    total = 0
    for elem in param.values():
        total += elem
    return total

themap = { 'mike':10, 'jill':20, 'sam':30 }
x = thesum(themap)
print(x)
```

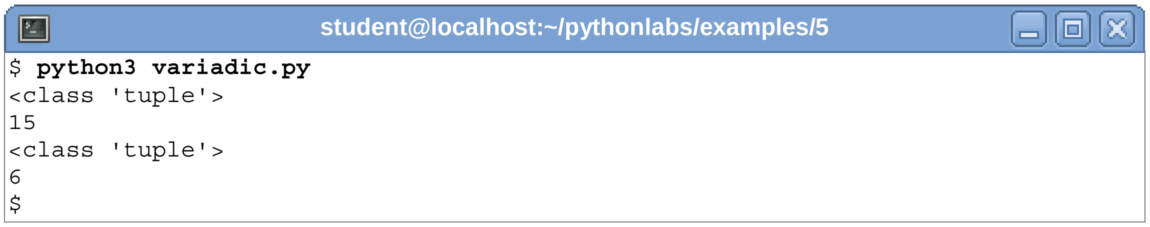
Variable Number of Arguments

- Python functions are capable of accepting a variable number of arguments.
  - Note the way in which the parameter is defined.
  - Note also the data type of `args` in the output.

variadic.py

```
#!/usr/bin/env python3
def thesum(*args):
    total = 0
    print(type(args))
    for elem in args:
        total += elem
    return total
x = thesum(1,2,3,4,5)
print(x)
x = thesum(1,2,3)
print(x)
```

- ▸ The output of the above program is shown below.



```
$ python3 variadic.py
<class 'tuple'>
15
<class 'tuple'>
6
$
```

- There may be occasions when you need to pass several arguments to a function and have them treated as a collection and also pass one or two other items.
  - ▸ In this case, the collected items will be treated as a `tuple`.
  - ▸ There are two ways of doing this, each requiring special syntax.
    - Pass the arguments first, followed by the `tuple`.
    - Pass the `tuple` first, and then the named the arguments.

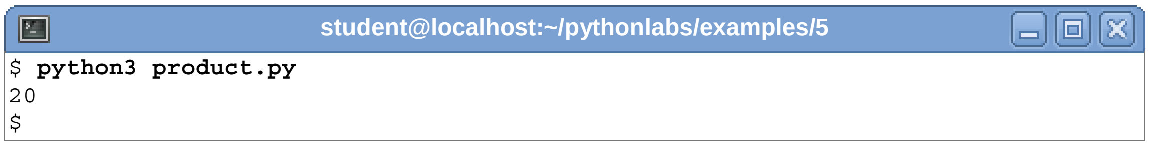
Variable Number of Arguments

- The example below passes the argument(s) first, followed by the `tuple`.

product.py

```
#!/usr/bin/env python3
def product(begin, *items):
    res = begin
    for i in items:
        res *= i
    return res
result = product(5, 3, 4, 2)
print(result)
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/5

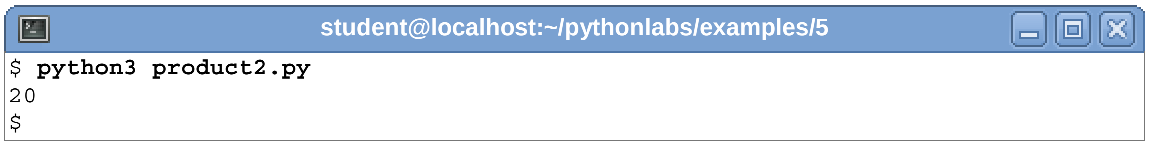
```
$ python3 product.py
20
$
```

- The next example passes the `tuple` first, and then the named arguments.

product2.py

```
#!/usr/bin/env python3
def product(*items, begin):
    res = begin
    for i in items:
        res *= i
    return res
result = product(3, 4, 2, begin = 5)
print(result)
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/5

```
$ python3 product2.py
20
$
```

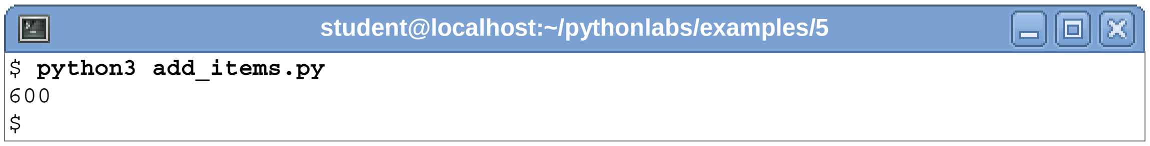
Variable Number of Arguments

- If you wish the collected elements to be interpreted as a dictionary, the following rules must be followed.
  - The calling code must use the keyword syntax for passing arguments.
  - The parameter that receives the dictionary needs to use the `**` notation.

add_items.py

```
#!/usr/bin/env python3
def addvals(val, **items):
    total = 0
    for amt in items.values():
        if amt > val:
            total += amt
    return total

total = addvals(0, mike=100, jane=200, peter=300)
print(total)
```

  - The output of the above program is shown below.



```
$ python3 add_items.py
600
$
```

Scope

- In programming languages, the term **scope** refers to that part of the program where a symbol is known.

- Python defines the following three scopes.

  - **local**

    - Local scope refers to the same suite of statements.
    - Variables used within the same suite are local to the suite.

  - **global**

    - Names with the global scope are available to all statements in the module.
    - If the developer does not make a variable global, then it is assumed local to the suite.

  - **built-in**

    - Names in the built-in scope are defined by Python and are available to all statements in the application.

- When a symbol (variable name, function name, class name, etc.) is referenced in the application, Python will always search for the symbol in the local scope first.

  - If the symbol is not found in the local scope, Python then searches the global scope for the symbol.

  - The built-in scope is only searched if the symbol is not found in the local or global scopes.

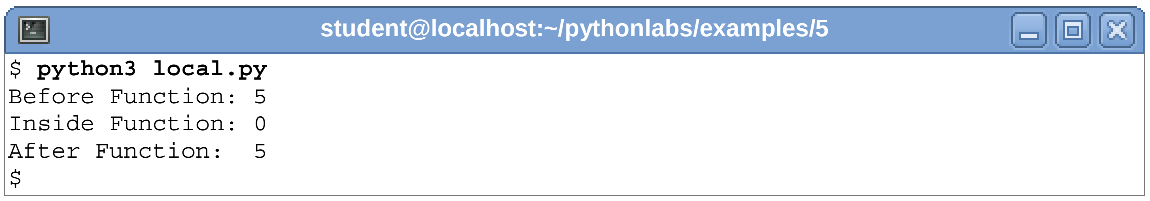
- The examples on the next page demonstrate the difference between a global and a local variable.

Scope

local.py

```
#!/usr/bin/env python3
def demo():
    count = 0
    print("Inside Function:", count)

count = 5
print("Before Function:", count)
demo()
print("After Function: ", count)
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/5

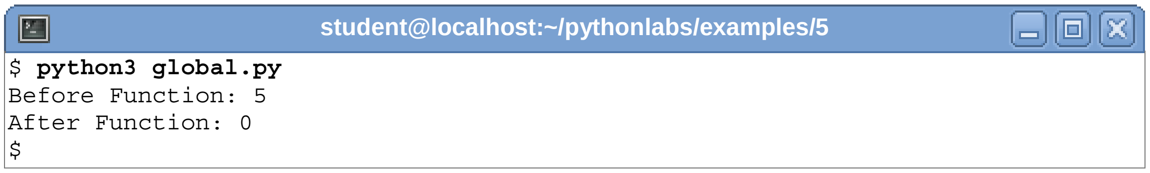
```
$ python3 local.py
Before Function: 5
Inside Function: 0
After Function:  5
$
```

- The following example makes the `count` a global variable inside of the function using the `global` keyword.

global.py

```
#!/usr/bin/env python3
def demo():
    global count
    count = 0

count = 5
print("Before Function:", count)
demo()
print("After Function:", count)
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/5

```
$ python3 global.py
Before Function: 5
After Function: 0
$
```

Functions - "First Class Citizens"

- Most programmers are comfortable with the notion of sending data to a function.
  - In Python, it is just as easy to send a function to a function.
- To understand this, it is important to recognize the difference between the following two notions.
  - `fun()` = invoking the function named `fun`
  - `fun` = referencing the function named `fun`
- The latter of the two forms has some special uses.
  - For example, suppose we have the following two functions available.

```
def square(p):
    return p * p

def increment(p):
    return p + 1
```

  - We could write a generalized `compute` function that, in addition to taking a list as an argument, also takes a function as an argument.

```
def compute(func, lis):
    for index, item in enumerate(lis):
        lis[index] = func(item)
```

  - Here are a few calls to the `compute` function.

```
lis = [ 10, 20, 30, 40 ]
compute(square, lis)
compute(increment, lis)
```

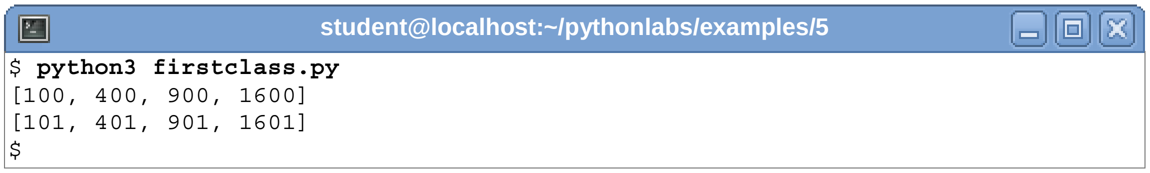
Passing Functions to a Function

- Here is the entire script.

firstclass.py

```
#!/usr/bin/env python3
def square(p):
    return p * p

def increment(p):
    return p + 1

def compute(func, lis):
    for index, item in enumerate(lis):
        lis[index] = func(item)

lis = [10, 20, 30, 40]
compute(square, lis)
print(lis)
compute(increment, lis)
print(lis)
```

- The output of the above program is shown below.



```
$ python3 firstclass.py
[100, 400, 900, 1600]
[101, 401, 901, 1601]
$
```

  - In each of the calls to the `compute` method, the first argument is a reference to a function, and the second argument is a list.
  - Inside the function, the first parameter is applied to each element of the list.

- Python provides various data types that apply a function to each of the elements of a list.
  - The `map` and `filter` data types are two such types that are available.

`map`

- The syntax for using the `map` data type may look like the other functions we have been working with.
  - It will become more apparent in the chapter on Python classes.

- The syntax for using a `map` is as follows.

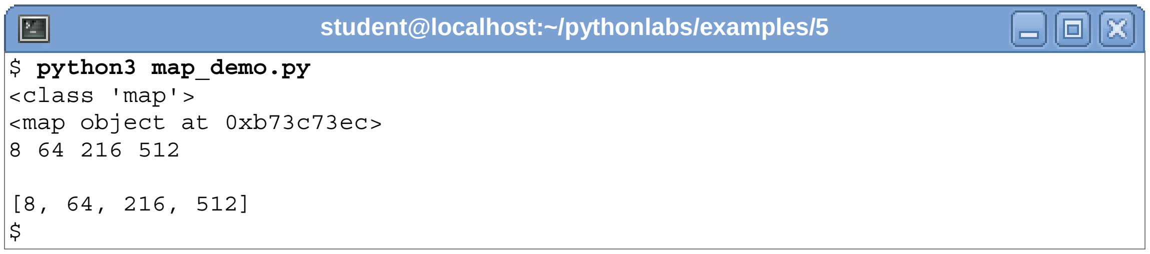
  `result = map(`*`function_name, a_sequence`*`)`

  - The above call to `map` calls the function named *`function_name`* for each element in `a_sequence`.
  - The `result` can then be iterated through using a `for` loop, or it could be converted to a `list` with the `list()` function.

`map_demo.py`

```
#!/usr/bin/env python3
def cube(x):
    return x * x * x

data = [2, 4, 6, 8]
cubed_data = map(cube, data)
print(type(cubed_data))
print(cubed_data)
for value in cubed_data:
    print(value, end=" ")
print("\n")

cubed_data = list(map(cube, data))
print(cubed_data)
```

- The output of the above program is shown below.



```
$ python3 map_demo.py
<class 'map'>
<map object at 0xb73c73ec>
8 64 216 512

[8, 64, 216, 512]
$
```

`filter`

- Another data type provided by Python that applies a function to each of the elements of a list is the `filter` data type.

- The syntax for the using a `filter` is as follows.

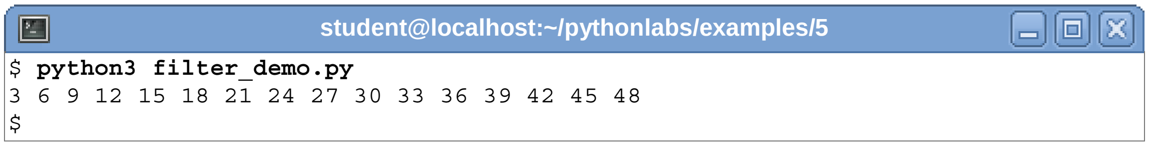
  `result = filter(`*`a_function`*`, `*`a_sequence`*`)`

  - The `result` is a `filter` object consisting of all of the elements of *`a_sequence`* for which *`a_function`* returns `True`.

`filter_demo.py`

```
#!/usr/bin/env python3
def multiple_of_three(x):
    return x % 3 == 0

results = filter(multiple_of_three, range(2, 51))
for value in results:
    print(value, end=' ')
print()
```

  - The output of the above program is shown below.



```
$ python3 filter_demo.py
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
$
```

Mapping Functions in a Dictionary

- The following example further demonstrates the advantage of the "first class citizen" status of Python functions by creating a "switch table."
  - A switch table is a dictionary whose keys map to functions.

switch_table.py

```
#!/usr/bin/env python3
def add():
    val = input("enter value to add: ")
    lis.append(val)

def delete():
    x = lis.pop(0)
    print("removing", x)

def display():
    print("displaying:", lis)

def terminate():
    print("terminating")
    exit()

lis = []
themap = {1:add, 2:delete, 3:display, 4:terminate}
while True:
    for index, fun in themap.items():
        print(index, fun.__name__)
    key = int(input("Make selection: "))
    if key in themap.keys():
        themap[key]()
    else:
        print("illegal selection\n")
```

- When a user selects a menu item, that value is used as the key into the dictionary.
  - `themap[key]` is the function name.
  - `themap[key]()` invokes the function.

Lambda

- Python supports the runtime creation of "anonymous" functions (functions that are not bound to a name) using the `lambda` statement.
    - This is a powerful concept and one that is often used in conjunction with functions and data types, such as `filter` and `map`.

- Here is the difference between a regular function and a `lambda` function.
    - Regular

```
def sample(x):
    return x ** 2
print(sample(8))        # 64
```

    - Lambda

```
sample = lambda x: x ** 2
print(sample(8))          # 64
```

- `lambda` functions must be one-liners.
    - The parameters are defined before the colon.
    - The work of the function is defined after the colon.
    - A return statement is unnecessary.

- `lambda` functions are not limited to one parameter.

```
f = lambda x, y: x * y
print(f(3,4))          # 12
```

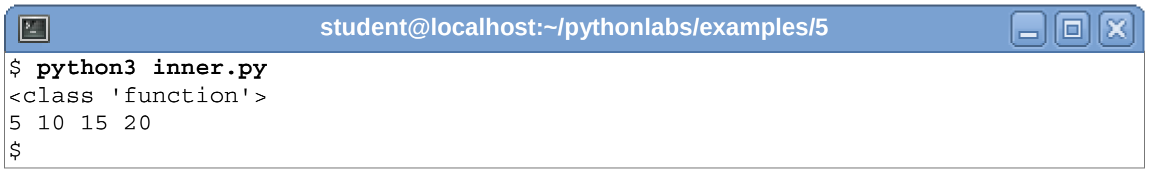
Inner Functions

- In Python, a function can contain another function.
  - In this case, the inner function will have access to any variables in the scope of the outer function.

inner.py

```
#!/usr/bin/env python3
def outer(a, b):
    x = 15
    y = 20

    def inner():
        print(a, b, x, y)

    return inner # reference to inner function

y = outer(5, 10)
print(type(y))
y() # call the returned function
```

  - The output of the above program is shown below.



```
$ python3 inner.py
<class 'function'>
5 10 15 20
$
```

- The example above demonstrates a few principles.
  - A function can return another function.
  - When the inner function is executed, the variables from the outer function are accessible.

- We could have used a lambda function in the code above.

```
def func(a, b):
    x = 15
    y = 20
    return lambda : print(a, b, x, y)
```

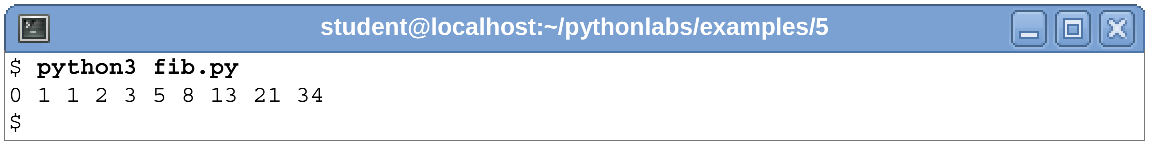
Closures

- A slight modification to the above code can be used to create a closure.
  - A **closure** is an inner function that remembers variables from an outer function from invocation to invocation.
    - In the example below, we use a closure to compute numbers in the Fibonacci sequence.

fib.py

```
#!/usr/bin/env python3
def fun():
    f = 0
    s = 1
    def fibonacci():
        nonlocal f
        nonlocal s
        next = f
        f,s = s, f + s
        return next
    return fibonacci

result = fun()

for i in range(10):
    print(result(), end = " ")
print()
```

  - The output of the above program is shown below.



```
$ python3 fib.py
0 1 1 2 3 5 8 13 21 34
$
```

    - In the code above, the `nonlocal` keyword makes those variables reference the outer functions variables, rather than define a new set of local variables to the inner function.
    - The values of the `nonlocal` variables can also be modified, if needed.

Exercises

1. Write a function that receives a list as its parameter and returns a new list of the positive elements only.
2. Write a function that is passed a variable number of arguments and a number, say `num`, as its two parameters.
   - The function should return the count of the values in the tuple (the variable number of parameters) that are greater than `num`.
     - For example, one such call to the function is shown below.

       ```
       res = pos(5, -10, 10, -20, 30, num=0)
       ```
     - In this case, the function would return `3`.
3. Write a function that returns a function that, when executed, returns the sum of two integers.
4. Re-write your solution to Exercise 3 using a `lambda` function.
5. In Python, if you wish to reverse sort a list, you will need to do the following.

```
values = [10, 40, 30, 20, 5]
values.sort()
values.reverse()
print(values)
```

   - Write your own versions of `sort` and `reverse` so that each of the following is possible.

```
values = [10,40,30,20,5]
print(sort(values))
print(reverse(values))
print(reverse(sort(values)))
```

     - Make sure the lists themselves are not altered.

Exercises

6. Write and test a function that takes two lists as arguments and returns a list of the elements that are common to both of the argument lists.

7. Write and test a function that takes a number and a dictionary and adds the number to all values in the dictionary.

    - You can assume that all the values (but not necessarily the keys) in the dictionary are numbers.

Chapter 6: Modules

1) Modules **6-2**

2) Standard Modules - `sys` **6-5**

3) Standard Modules - `math` **6-6**

4) Standard Modules - `time` **6-7**

5) The `dir` Function **6-9**

Modules

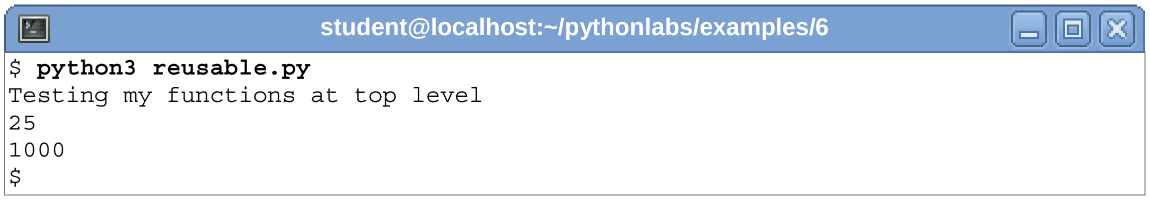
- As we have seen, programs are better supported by using functions.
    - As your scripts get longer, you may want to split them into several files for easier maintenance.
    - You may also want to use a handy function that you have written in several programs without copying its definition into each program.
- Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter.
    - Such a file is called a **module**.
- A **module** is a file containing Python definitions and statements.
    - The file name is the module name with the suffix `.py` appended.
    - Definitions from a module can be imported into other modules or into the main module.
        - The main module (named `__main__`) is the collection of variables that you have access to in a script executed at the top level and in an interactive Python shell.
- Within a module, the module's name is accessible via the value of the global variable `__name__`.
    - For instance, suppose we need to use the functions in the example on the next page in various Python applications.
        - We will define them in a file called `reusable.py`, also shown on the next page.

Modules

reusable.py

```
#!/usr/bin/env python3
def square(p):
    return p * p
def cube(p):
    return p * p * p

if __name__ == "__main__":
    print("Testing my functions at top level")
    print(square(5))
    print(cube(10))
```

- The output of the above program is shown below.



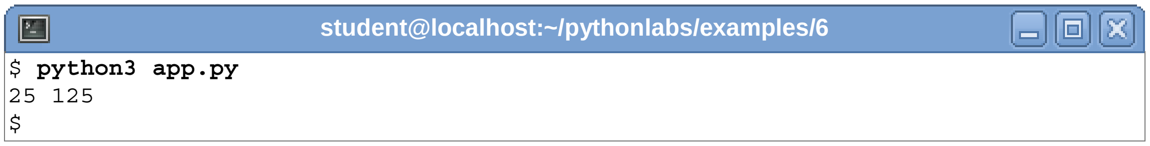
```
$ python3 reusable.py
Testing my functions at top level
25
1000
$
```

- Any application desiring to use the above functions must import `reusable.py`.
  - Suppose the following application wishes to use one or more of the functions in the `reusable` module.

app.py

```
#!/usr/bin/env python3
import reuseable

a = reuseable.square(5)
b = reuseable.cube(5)
print(a, b)
```

- The output of the above program is shown below.



```
$ python3 app.py
25 125
$
```

Modules

- Each module has its own symbol table, which contains the identifiers (names) used in that module.
  - Programming languages often refer to these tables as **namespaces**.
    - In the absence of any imported modules, Python always looks for names in the namespace of the main module.
    - To direct Python to look into another module in order to resolve a name, use the module name ahead of the function name, as in the code below.

      ```
      xsq = reuseable.square(5)
      ```
  - Alternately, you can import the entire `reuseable` namespace.
    - In this situation, the symbols from the imported file become a part of the main namespace.

alternate.py

```
#!/usr/bin/env python3
from reuseable import *

print("Module name is: " + __name__)

a = square(5)
b = cube(5)
print(a, b)
```

- When a module is imported, Python first searches for the `.py` file in the current directory.
  - If it is not found there, then it looks in the list of directories specified by the variable `sys.path`.
    - `sys.path` is initialized, in part, from the environment variable `PYTHONPATH)`.

Standard Modules - `sys`

- Python comes with a library of standard modules, some of which are built into the interpreter.
  - These provide access to operations that are not part of the core language but are built in, either for efficiency or to provide access to operating system primitives, such as system calls.
- The `sys` module contains some system specific properties and functions.
  - `sys.argv`
    - A list of command line arguments
  - `sys.maxsize`
    - The maximum size of a C-style integer
    - Usually `2**31 - 1` on a 32-bit platform and `2**63 - 1` on a 64-bit platform
  - `sys.version`
    - The version of the Python interpreter
- The following program illustrates these properties from the `sys` module.

sys_testing.py

```
#!/usr/bin/env python3
import sys
print("Command Name:", sys.argv[0])
print("Command Line Arguments Are:", sys.argv[1:])
print("sys.maxsize =", sys.maxsize)
print()
print("sys.path =", sys.path)
print()
print("sys.version =", sys.version)
print()
print("Version = ", sys.version.split()[0])
```

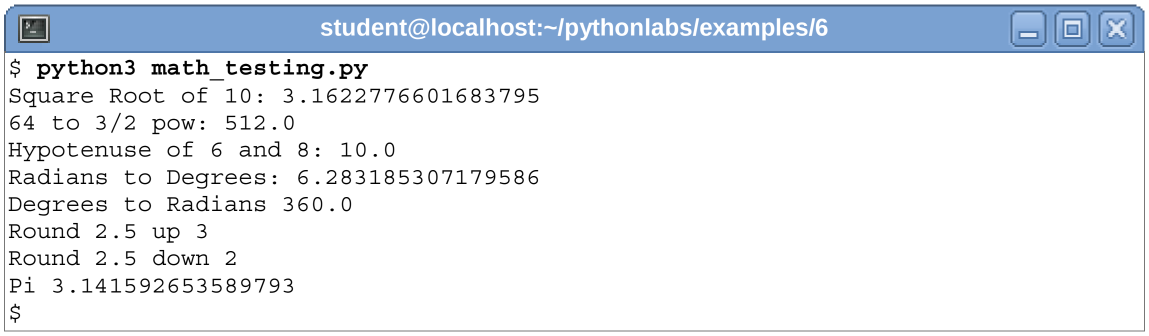
Standard Modules - `math`

- The `math` module contains some standard mathematical functionality.

`math_testing.py`

```
#!/usr/bin/env python3
import math

print("Square Root of 10:", math.sqrt(10))
print("64 to 3/2 pow:", math.pow(64, 1.5))
print("Hypotenuse of 6 and 8:", math.hypot(6,8))
radians = math.radians(360)
print("Radians to Degrees:", radians)
print("Degrees to Radians", math.degrees(radians))
print("Round 2.5 up", math.ceil(2.5))
print("Round 2.5 down", math.floor(2.5))
print("Pi", math.pi)
```

- The output of the above program is shown below.



```
$ python3 math_testing.py
Square Root of 10: 3.1622776601683795
64 to 3/2 pow: 512.0
Hypotenuse of 6 and 8: 10.0
Radians to Degrees: 6.283185307179586
Degrees to Radians 360.0
Round 2.5 up 3
Round 2.5 down 2
Pi 3.141592653589793
$
```

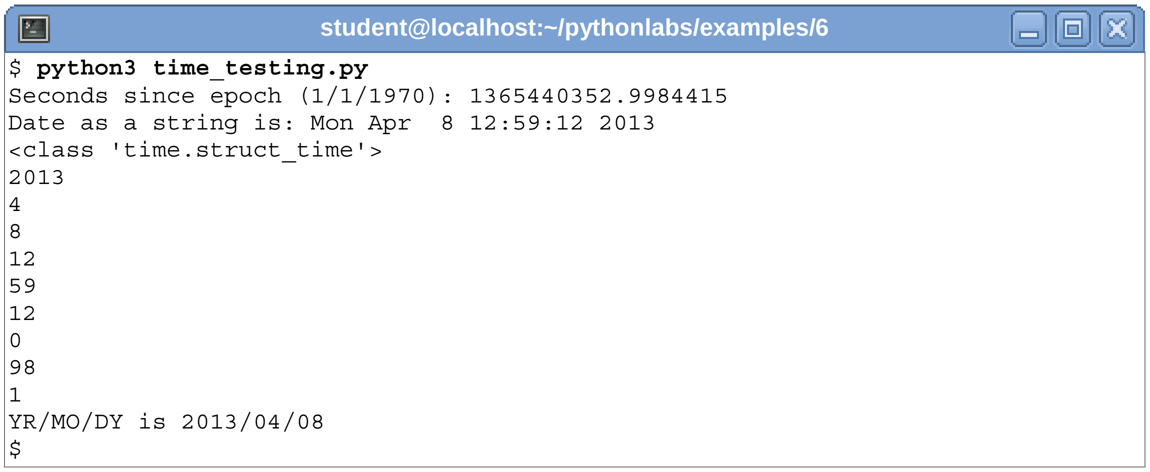
Standard Modules - `time`

- Below is a program that illustrates some handy functions from the `time` module.

mytime.py

```
#!/usr/bin/env python3
import time

now = time.time()     # time as seconds since epoch
print("Seconds since epoch (1/1/1970):", now)

date = time.ctime(now)     # time as the date
print("Date as a string is:", date)

pieces = time.localtime()     # grab pieces of time
print(type(pieces))
for piece in pieces:     # print each piece
    print(piece)

print(time.strftime("YR/MO/DY is %Y/%m/%d"))
```

  - The output of the above program is shown below.



```
$ python3 time_testing.py
Seconds since epoch (1/1/1970): 1365440352.9984415
Date as a string is: Mon Apr  8 12:59:12 2013
<class 'time.struct_time'>
2013
4
8
12
59
12
0
98
1
YR/MO/DY is 2013/04/08
$
```

- The type of the time value sequence returned by `localtime()` is an object with a named `tuple` interface.
  - Its values can be accessed by index and by attribute name.

Standard Modules - `time`

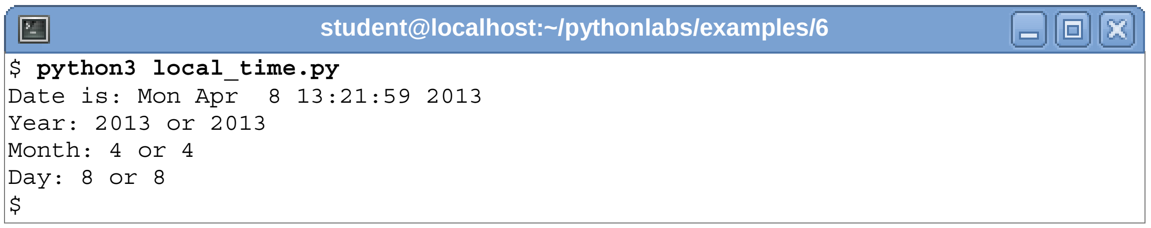
- The following table lists the values that are available from the `localtime()` function.

Index	Attribute	Values
0	`tm_year`	(for example, 1993)
1	`tm_mon`	Range from 1 to 12
2	`tm_mday`	Range from 1 to 31
3	`tm_hour`	Range from 0 to 23
4	`tm_min`	Range from 0 to 59
5	`tm_sec`	Range from 0 to 61
6	`tm_wday`	Range from 0 to 6, Monday is 0
7	`tm_yday`	Range from 1 to 366
8	`tm_isdst`	0, 1 or -1
N/A	`tm_zone`	abbreviation of time zone name
N/A	`tm_gmtoff`	offset east of UTC in seconds

`local_time.py`

```
#!/usr/bin/env python3
import time
date = time.ctime()
print("Date is:", date)

pieces = time.localtime()
print("Year:", pieces[0], "or", pieces.tm_year)
print("Month:", pieces[1], "or", pieces.tm_mon)
print("Day:", pieces[2], "or", pieces.tm_mday)
```

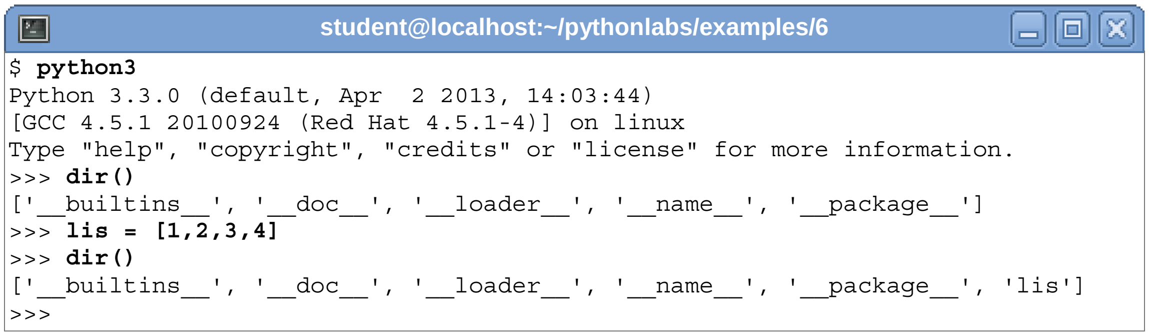
- The output of the above program is shown below.


student@localhost:~/pythonlabs/examples/6

```
$ python3 local_time.py
Date is: Mon Apr  8 13:21:59 2013
Year: 2013 or 2013
Month: 4 or 4
Day: 8 or 8
$
```

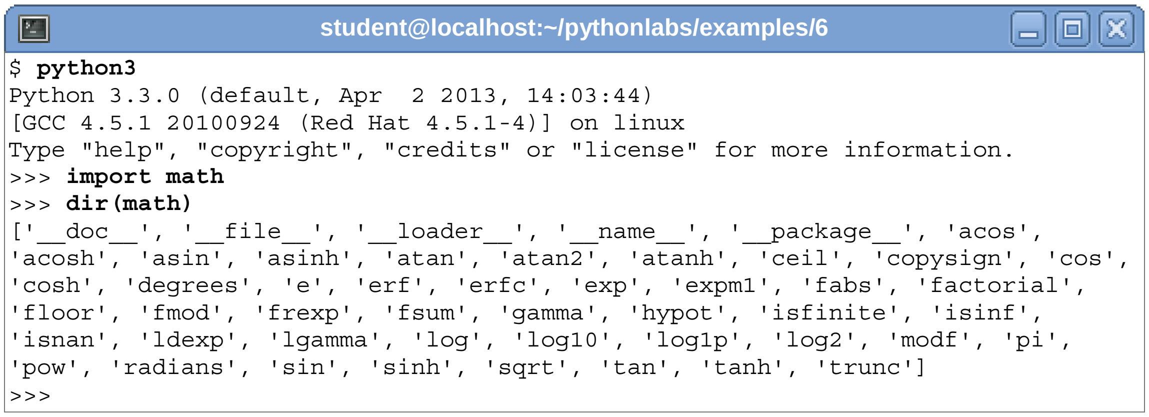
The `dir` Function

- The built-in function `dir` is used to determine which symbols a module defines.
  - It returns a sorted list of strings.
    - In its simplest form, you will see the names defined in the main module.


student@localhost:~/pythonlabs/examples/6

```
$ 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.
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
>>> lis = [1,2,3,4]
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', 'lis']
>>>
```

  - When the name of a module is passed as the argument to the `dir` function, a list of all of the symbols in the module is returned.


student@localhost:~/pythonlabs/examples/6

```
$ 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.
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos',
'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>>
```

- If you wish to see a list of Python built-in symbols, you can use `dir` in the following way.

```
dir(__builtins__)
```

Exercises

1. Create a few functions and place them in a file.
    - Now, write a Python program that imports the module.
        - Use the functions from the module in this program.
2. Create a new file and define a function in it with the same name as one of the functions from the previous exercise.
    - Within the same file, create an application that imports module from the previous exercise.
        - The application should call both the function from the imported module, and the function within the current module.
3. Write a program that sorts its command line arguments.
4. Write a program that sums the command line arguments.

Chapter 7: Exceptions

1) Errors 7-2

2) Runtime Errors 7-3

3) The Exception Model 7-4

4) Exception Hierarchy 7-7

5) Handling Multiple Exceptions 7-9

6) `raise` 7-11

7) `assert` 7-12

Errors

- Writing a program is often a difficult task.

- Any program typically undergoes many revisions before it is ready for production.
  - Errors found by the compiler are typically easier to fix than those not found by the compiler.

- Once the program is free of compiler errors, things can still go wrong.

- Subsequent executions of the program will show you errors in logic.
  - Some logical errors may be "hiding" and not surface until the proper data has been sent through the program.
  - Testers can generate data that causes the execution of all of the control paths in a program.
    - In this way, further logical errors can be flushed out.

- Once the logic errors have been fixed, there is the possibility of run time errors occurring during the execution of the program.
  - Some of these errors are listed below.
    - `ValueError`
    - `IndexError`
    - `KeyError`
    - `IOError`
  - This chapter concentrates on catching and handling these and other types of Python runtime errors.

Runtime Errors

- When a Python exception occurs in a program, the application terminates with an error message.
  - The error message typically lists the exception that was raised and the line number in the source file that caused the exception.
  - Therefore, if you wish to "trap" this error and handle it yourself, you will need to write an exception handler.
    - To illustrate the need for these handlers, observe the following program that expects a series of integers and then outputs the sum of those integers.

totals.py

```
#!/usr/bin/env python3
total = 0
while True:
    value = input("Please enter a number: ")
    if  value == "end":
        break
    total += int(value)

print("Total is", total)
```

- If the user enters non-integer data (other than "`end`"), the program responds with the following.

```
ValueError: invalid literal for int()
```

  - In other words, a `ValueError` exception has been raised and the program terminates.
    - This is the default Python runtime response to this kind of exception.
    - If developers wish to respond in their own way, they must enable their own exception handling code.

The Exception Model

- In Python, the exception model consists of the following.
  - A `try` statement
    - The `try` statement enables the programmer-written handlers defined below it.
  - One or more exception handlers
    - Each exception handler is specified in an `except` clause.
- The code below is a rewrite of the code on the previous page.
  - This version incorporates Python's exception handling model.

handler.py

```
#!/usr/bin/env python3
total = 0
while True:
    value = input("Please enter a number: ")
    if  value == "end":
        break
    try:
        total += int(value)
    except ValueError:
        print("Invalid number - try again")

print("Total is", total)
```

- Now, when an illegal value is entered, the program responds with a message and re-prompts the user for the next number.
- Of course, one could elect to terminate the program upon the occurrence of this exception.

```
except ValueError:
    print("Invalid number - program exiting")
    exit(1)
```

The Exception Model

- The combination of `try` and `except` clauses can occur in various parts of your program.

various.py

```
#!/usr/bin/env python3
total = 0

# Custom Handler
try:
    value = input("Please enter a number: ")
    total += int(value)
except ValueError:
    print("Invalid number")
    exit(1)
print("Total is", total)

# Default System's Handler
value = input("Please enter a number: ")
total += int(value)
print("Total is", total)


#Another Custom Handler
try:
    value = input("Please enter a number: ")
    total += int(value)
except ValueError:
    print("Invalid number")
    exit(2)
print("Total is", total)
```

- A `ValueError` is raised when the `int()` function is passed an inappropriate argument.
  - The `except` clause in the above code will execute only when a `ValueError` occurs within a `try` suite.
  - If a `ValueError` occurs outside of a `try` suite, the standard Python runtime termination will occur.

The Exception Model

- Since an `Exception` is an object, it has methods that can be called on it.
  - You can receive a reference to the object in a handler and then execute certain methods on the exception object.

    ```
    except ValueError as err:
        print(err)
        exit(1)
    ```

    - The exception's `__str__` method will be called whenever a reference to the Exception is passed either to the `print()` or `str()` functions.
  - You could also provide an `else` clause after the `except` clause.
    - The `else` suite would execute if the exception did not occur.

      ```
      except ValueError as err:
          print(err)
          exit(1)
      else:
          print("Exception did not occur");
      ```
  - You can also make your handler "mute" by doing nothing when the exception is raised.
    - In this case, you will need the `pass` statement to satisfy the Python syntax requirement.

      ```
      except ValueError:
          pass
      ```

Exception Hierarchy

- The complete list of Python exceptions is shown below.

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError, OverflowError, ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError, KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError, ConnectionAbortedError
      |    |    +-- ConnectionRefusedError, ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError, UnicodeEncodeError
      |         +-- UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning, PendingDeprecationWarning
           +-- RuntimeWarning, SyntaxWarning
           +-- UserWarning, FutureWarning
           +-- ImportWarning, UnicodeWarning
           +-- BytesWarning, ResourceWarning
```

Exception Hierarchy

- It is important to note that Python's exception hierarchy underwent some major changes with version 3.3.0 and newer.

- PEP 3151 deals with the reworking of the OS and IO exception hierarchy.
  - In Python version 3.3.0 and newer, the hierarchy of exceptions raised by operating system errors is now both simplified and finer-grained.
  - Developers need not worry anymore about choosing the appropriate exception type between exception classes like `OSError`, `IOError`, `EnvironmentError`, and `WindowsError`.
    - All these exception types are now only one: `OSError`.
    - The other names are kept as aliases for compatibility.
  - Also, it is now easier to catch a specific error condition.
    - Instead of inspecting the `errno` attribute for a particular constant from the `errno` module, you can catch the adequate `OSError` subclass.

- It is important to note that this is not a change from Python 2 to Python 3.
  - It is a refinement of the exception hierarchy from version `3.2.x` to version `3.3.x`.
  - The exception hierarchy shown on the previous page is that of Python version `3.3.0`.

Handling Multiple Exceptions

- As seen from the list of exceptions on the previous page, there are many different exception types in Python.
  - A programmer might be interested in handling more than one exception at various points in the program.
    - In any area of your program, you can trap any number of exceptions.
  - The program below handles several different exceptions.

multi.py

```
#!/usr/bin/env python3
names = ['Mike', 'John',  'Jane',  'Alice']
themap = {'Mike':15, 'Chris':10, 'Dave':25}

while True:
    try:
        value = input("\n\nEnter an integer: ")
        if value == "end":
            break

        value = int(value)
        print("Name is: " + names[value])
        name = input("Enter a name: ")
        print(name, " => ", themap[name])

    except ValueError:
        print("Value Error: non numeric data")

    except IndexError as ie:
        print("Illegal subscript:", ie)

    except KeyError as ke:
        print("Illegal key")

    except:
        print("Unknown Error: ")
```

Handling Multiple Exceptions

- In the event that multiple exceptions are handled, you may wish to handle a few of them in the same way.
  - For example, the following code could have been used to handle both a `KeyError` and an `IndexError`.

```
except (KeyError, IndexError) as err:
    print("Error", err)
```

    - In the above case, the two exceptions that were handled together descend from a `LookupError`.
    - Therefore, the code could also have been handled in the following way.

```
except LookupError as err:
    print("Error", err)
```

- When an exception occurs, the list of `except` clauses is searched from beginning to end until a matching exception handler is found.
  - In the case where you are trapping exceptions from the same hierarchy, make sure to place the "deepest-derived' exceptions ahead of the others.

- For example, the following is the correct order between `KeyError` and `Exception`.

```
while True:
    try:
        val = input("Enter an integer: ")
        val = val * val;
    except KeyError:
        print("Key Error")
    except Exception:
        print("Exception")
```

raise

- The Python exception model allows the developer to `raise` an exception at any strategic time.
  - Sometimes, this is the case when you are in the middle of a handler and you wish to defer the rest of the handling up the call stack.

```
except Exception as err:
    print(err)
    raise
```

  - As the handler above is being executed, the presence of `raise` raises the same exception that is being handled.
    - Python's action is to see if there is a handler in the next (and then the next …) function in the call stack.
    - Ultimately, if there are none, then the Python runtime handler will execute and handle the exception.
- You can also explicitly raise any exception that you wish.
  - The example below raises a `ValueError` for any odd number.

raising.py

```
#!/usr/bin/env python3
result = input("enter an even number: ");
try:
    pos = int(result)
    if int(pos) % 2 != 0:
        raise ValueError("# Not Even:" + str(pos))
    print("Number is", pos)
except ValueError as err:
    print(err)
```

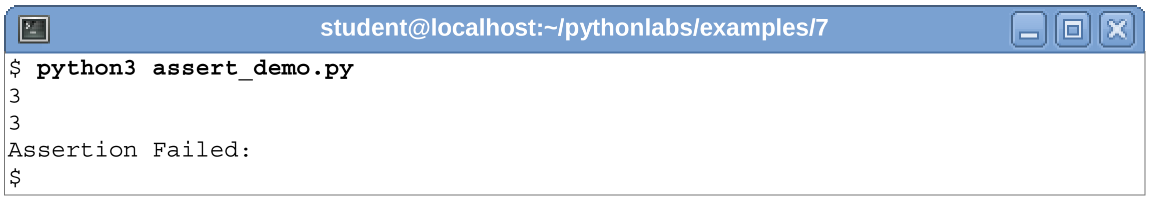
`assert`

- The `assert` function allows the programmer to make assertions at strategic points in the development of a program.
  - Assertions are expressions that evaluate to either `True` or `False`.
    - If an assertion evaluates to `False`, an `AssertionError` exception is raised.
    - If the assertion evaluates to `True`, control flow passes to the first statement following the assertion.

`assert_demo.py`

```
#!/usr/bin/env python3
def findmax(a,b):
    max= 0
    if a < b:
        max= b
    if b < a:
        max= a
    assert (max == a or max == b)
    return max;

try:
    print(findmax(2,3))
    print(findmax(3,2))
    print(findmax(3,3))
except AssertionError:
    print("Assertion Failed:")
```

- The output of the above program is shown below.



```
$ python3 assert_demo.py
3
3
Assertion Failed:
$
```

Exercises

1. Create a list in your program that has 10 numbers.
   - Then, in a loop, ask the user for a number.
     - Use this number as an index into your list and print the value located at that index.
     - End the program when the user enters "end."
     - Handle the case of an illegal number.
     - Handle the case of an illegal subscript.

2. Test Exercise 1 again by using a few negative numbers as the index.
   - Eliminate negative numbers as legitimate subscripts by raising the `IndexError` exception when a negative number is given.

3. Write a program that uses a loop to prompt the user and get an integer value.
   - The program should print the sum of all the integers entered.
     - If the user enters a blank line or any other line that cannot be converted to an integer, the program should handle this `ValueError`.
     - If the user uses Ctrl-C to terminate the program, it should be trapped with a `KeyboadInterrupt`, and a suitable message should be printed.
     - When the user enters the end of file character (Ctrl-D on Linux or Ctrl-Z on Windows), the program should trap this with the `EOFError` and break out of the loop and print the sum of all the integers.

This Page Intentionally Left Blank

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

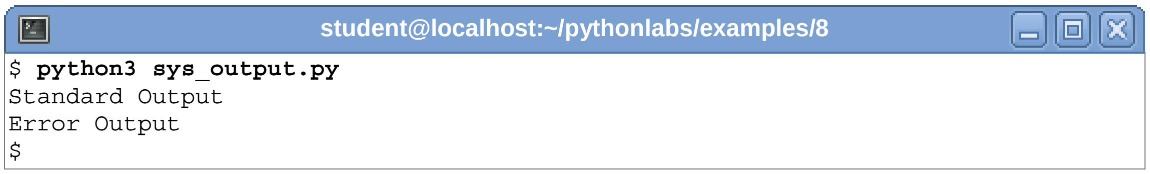
Data Streams

- 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

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

sys.stdout.write("Standard Output\n")
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.

```
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.

```
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.

Mode	File Opened for:
`r`	• Reading • Assumes file already exists
`w`	• Writing • Assumes file does not exist • If it does exist, file is truncated
`a`	• Appending • If file does not exist, it is created
`b`	• Binary
`r+`	• Read and Write • Assumes file already exists
`w+`	• Read and Write • Assumes the file does not exist
`a+`	• Read and Write at the end of the file • If file does not exist, it is created

  - 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`.

write1.py

```
#!/usr/bin/env python3
f = open('output', 'w')
f.write('This is a test.\n')
f.write('This is another test.\n')
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.

write2.py

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

f = open(sys.argv[1], 'w')
f.write('This is a test.\n')
f.write('This is another test.\n')
f.close()
```

Writing Data to a File

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

write3.py

```
#!/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

```
#!/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

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

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`.

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

```
#!/usr/bin/env python3
import sys
ct = lines = 0
file = open(sys.argv[1], "r")

while True:
    line = file.readline()
    if not line:
        break
    ct += len(line)
    lines += 1

file.close()
print("Characters:", ct, " Lines:", lines)
```

- The next example uses a slightly different technique.

read2.py

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

lines = 0
file = open(sys.argv[1], "r")

line = file.readline()
while line:
    lines += 1
    line = file.readline()
file.close()
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

```
#!/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

```
#!/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.

read5.py

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

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

data = file.read(10)
while data:
    print(data, end="")
    data = file.read(10)

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.

Constant	Value	Meaning
`os.SEEK_SET`	0	Reference point is the beginning of the file
`os.SEEK_CUR`	1	Reference point is the current position
`os.SEEK_END`	2	Reference point is the end of the file

    - 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

```
#!/usr/bin/env python3
import os
# Reading from file in binary mode
file = open("seekdata.txt", "rb")
print(file.read())

file.seek(3, os.SEEK_SET)
print(file.read(5))
print("Position:", file.tell())

```

Continued on the following page.

Additional File Methods

seek1.py ***(continued)***

```
file.seek(12, os.SEEK_CUR)
print(file.read(7))
print("Position:", file.tell())

file.seek(-20, os.SEEK_CUR)
print(file.read(5))
print("Position:", file.tell())
file.close()

print("#" * 30)

# Reading from file in text mode
file = open("seekdata.txt", "r")
print(file.read())

file.seek(12, os.SEEK_SET)
print(file.read(5))
print("Position:", file.tell())

file.seek(0, os.SEEK_CUR)
print(file.read(7))
print("Position:", file.tell())

file.seek(0, os.SEEK_CUR)
print(file.read(5))
print("Position:", file.tell())
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:

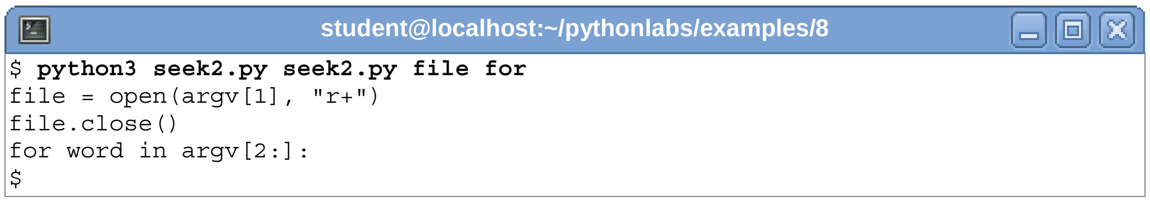
```
program filename strings...
```

- Here is such a program.

seek2.py

```
#!/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.

```
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.

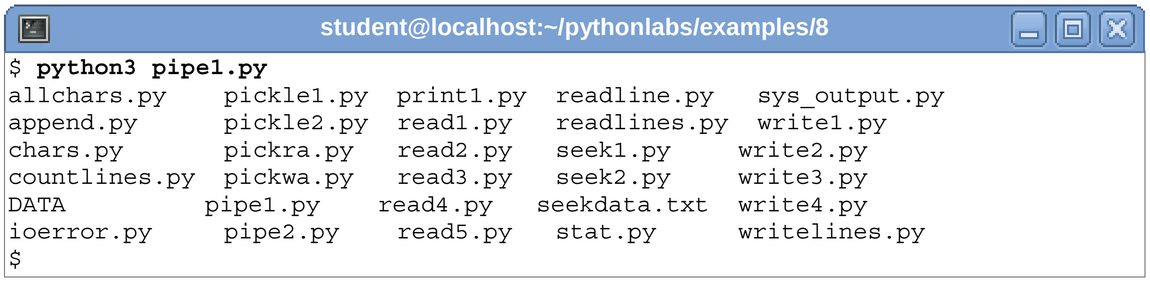
- Below is a program that reads data from a process.

pipe1.py

```
#!/usr/bin/env python3
import platform
p = platform.popen("dir", "r")
data = p.readlines()
p.close()
for i in data:
    print(i, end="")
```

Using Pipes as Data Streams

- The output of the previous program is shown below.



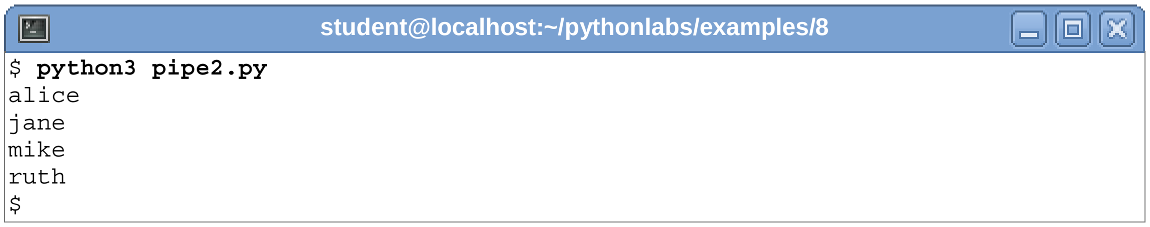
```
$ python3 pipe1.py
allchars.py    pickle1.py  print1.py  readline.py   sys_output.py
append.py      pickle2.py  read1.py   readlines.py  write1.py
chars.py       pickra.py   read2.py   seek1.py     write2.py
countlines.py  pickwa.py   read3.py   seek2.py     write3.py
DATA          pipe1.py    read4.py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

- The following program writes data to a process.

pipe2.py

```
#!/usr/bin/env python3
import platform

data = [ "mike", "jane", "alice", "ruth" ]

p = platform.popen("sort", "w")
for i in data:
    print(i, file=p)

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.

ioerror.py

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

try:
    file = open(sys.argv[1], "r")
except OSError as err:
    print(type(err))
    print(err)
    exit(1)
finally:
    print("IN FINALLY")

num = 1
line = file.readline()
while line:
    print(str(num) + "\t" + line, end="")
    num += 1
    line = file.readline()

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.

countlines.py

```
#!/usr/bin/env python3
import os

files = os.listdir(".")
ct = 0

for file in files:
    f = open(file, "r")
    lines = f.readlines()
    ct += len(lines)
    f.close()

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.

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

- 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

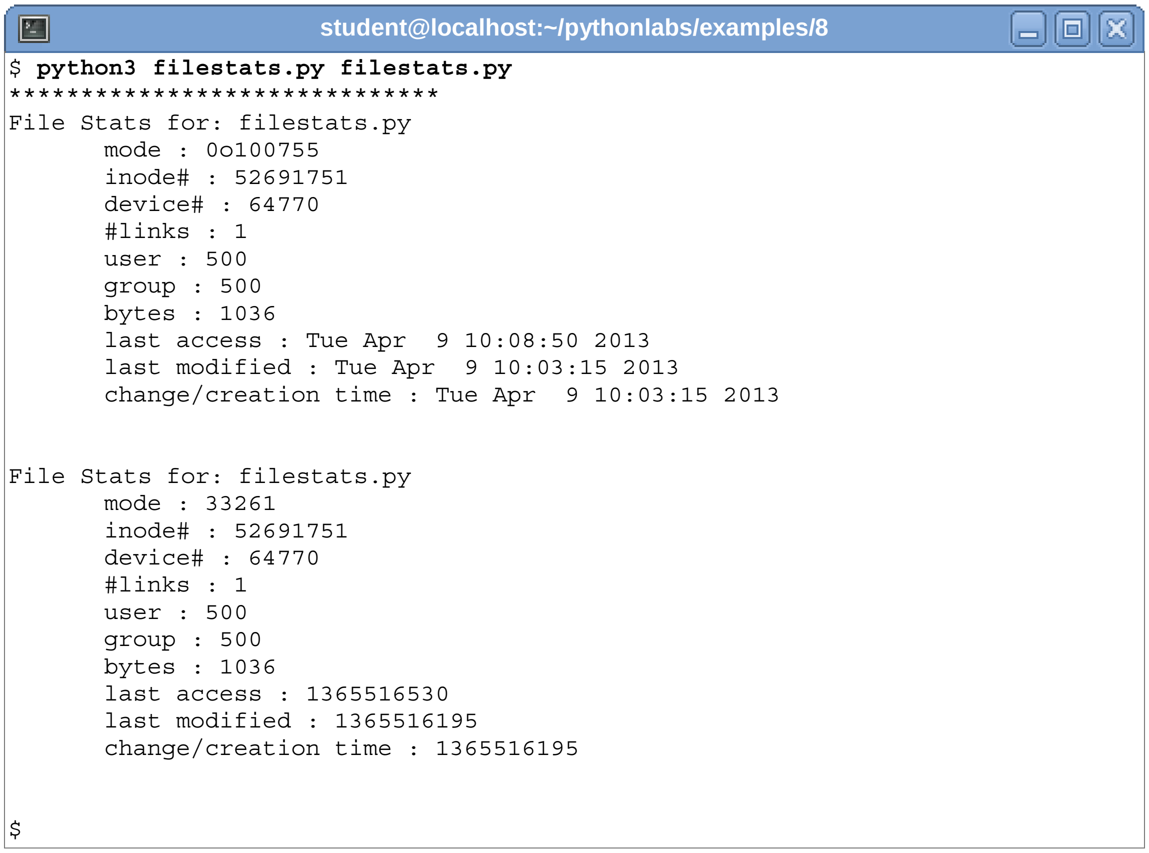
Metadata

filestats.py

```
#!/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.

Metadata


student@localhost:~/pythonlabs/examples/8

```
$ python3 filestats.py filestats.py
******************************
File Stats for: filestats.py
        mode : 0o100755
        inode# : 52691751
        device# : 64770
        #links : 1
        user : 500
        group : 500
        bytes : 1036
        last access : Tue Apr  9 10:08:50 2013
        last modified : Tue Apr  9 10:03:15 2013
        change/creation time : Tue Apr  9 10:03:15 2013


File Stats for: filestats.py
        mode : 33261
        inode# : 52691751
        device# : 64770
        #links : 1
        user : 500
        group : 500
        bytes : 1036
        last access : 1365516530
        last modified : 1365516195
        change/creation time : 1365516195


$
```

- 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

```
#!/usr/bin/env python3
import pickle
values = [50, 32.29, "Michael"]
f = open("output", "wb")
for value in values:
    pickle.dump(value, f)
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.

pickle2.py

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

f = open("output", "rb")
value = pickle.load(f)
cost = pickle.load(f)
name = pickle.load(f)
f.close()

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

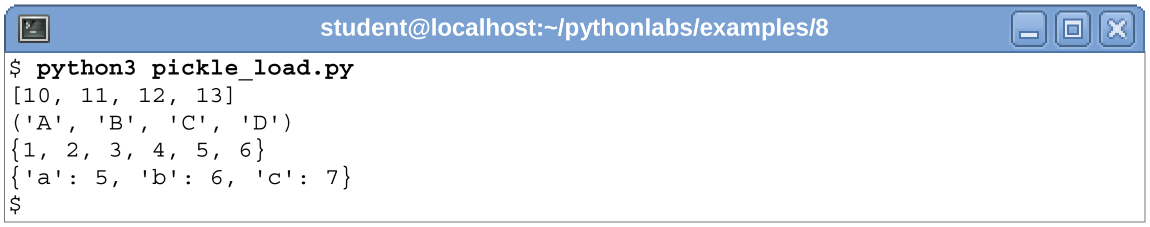
`pickle_dump.py`

```
#!/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`

```
#!/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.



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

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:

file1	**file2**	**file3**	**file4**
jane	susan	jane	jane
john	mary	john	chris
peter	dave	peter	peter
bill	mike	bill	bill
mike	alice	mike	mike
alice	chris	alice	alice
frank	beverly	frank	frank
bart	bill	bart	susan

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.

Chapter 9: Classes in Python

1) Classes in Python.. 9-2
2) Principles of Object Orientation.. 9-3
3) Creating Classes.. 9-4
4) Instance Methods... 9-7
5) File Organization... 9-8
6) Special Methods... 9-9
7) Class Variables... 9-13
8) Inheritance.. 9-14
9) Polymorphism... 9-17
10) Type Identification.. 9-22
11) Custom Exception Classes.. 9-23

Classes in Python

- Object-oriented programming is a style of programming that lends itself to the principle that a software solution should closely model the problem domain.
  - This is a course in Python programming and not object-oriented programming (OOP).
    - However, because Python is an object-oriented language, we need to develop a minimum vocabulary so that we can discuss OOP in Python.
- If your problem domain was banking, then some of the data types in your program might be Customer, Account, and Loan.
  - If your problem domain was system software, then there would be need for types, such as Process or File.
- The aforementioned types would give rise to objects (entities) characterized by the following.
  - Behavior: the actions the objects can perform
  - Attributes: the characteristics of each object
- In Python, the `class` keyword is used to create these new data types.

Principles of Object Orientation

- As you will see, the behavior of the objects is implemented as a set of functions.
  - In object-oriented parlance, functions are referred to as **methods**.

- Object Orientation is characterized by the following.
  - Encapsulation
    - The coupling of data and methods
  - Data abstraction
    - The mandate that access to data be achieved via a public interface, that is, those methods that are exposed to the user of this class
    - This is sometimes referred to as information hiding.
  - Inheritance
    - The derivation of specific data types from more general types
    - A typical object-oriented system contains inheritance hierarchies.
  - Polymorphism
    - In an inheritance hierarchy, there may be a set of classes, each with the same named methods and interfaces, but whose implementations are different.
    - Polymorphism is the ability of a language to differentiate these same named functions at run time.

Creating Classes

- For domain specific problems, good software will have domain specific entities (objects).
  - We illustrate this concept by creating a `Student` data type.
- The creation of a new data type is accomplished by using the `class` keyword.
  - A minimal `Student` data type would be defined with a `class` definition such as the one below.

```
class Student:
    """ This is the Student class file.  It
        represents a student abstraction.
    """
```

    - Everything inside the class will obey the usual indentation rules.
    - The first line inside a class affords the developer the capability of providing a documentation string.
    - The "doc string" is available via `Student.__doc__`.

  - Ultimately, the class definition will contain many methods, the collection of which defines the behavior for the class.
    - For now, we are just explaining the syntax.
    - Once we have a class definition, we can create objects of the class as in the last two lines below.

```
class Student:
    """ Documentation for the class """

    mike = Student()
    harley = Student()
```

Creating Classes

- Of course, the `Student` objects created on the previous page are skeletal in that they do not yet have any data or methods within them.

- In a software system in which classes are defined, there would typically be an analysis and design phase before any coding was undertaken.
  - In that phase, the behavior and the attributes of class objects would be determined by the designers.

- If we assume, for simplicity, that each `Student` object will have a name and a major, then it would be convenient to make the creation of an object include those two items.
  - This is accomplished by passing the name and the major to the "constructor" method.

    ```
    mike = Student("Michael", "Mathematics")
    harley = Student("Harley", "American History")
    ```

  - When an object is created In Python, a special method named `__init__` is called.
    - Therefore, the `Student` class should include this special method.
    - In this case, two explicit arguments are fed to this special method.
    - However, Python also passes a reference to the object being constructed.

Creating Classes

- The declaration of the `__init__` method might begin like this.

```
def __init__(self, name, major):
    pass
```

  - For now, we use the `pass` statement to satisfy the syntax requirement for a method.
  - We still need to add some code to fill the object with its data.

- The first parameter above is named `self`.
  - This is a reference to the object being created.
  - You can name this variable anything, but traditionally it is named `self`.

- The `__init__` method can be seen as a constructor of the object.
  - Its role is to at least copy the parameters to the object's data.
    - This is typically done as follows.

```
class Student:
    def __init__(self, name, major):
        self.thename = name
        self.themajor = major
```

  - `thename` and `themajor` are object data.
    - Each object will have these two variables.
  - You could also define other object data if you wish.
    - For example, the above `__init__` method might include the following.

```
self.desc = name + " " + major
```

Instance Methods

- Typically, a class would have methods to retrieve the instance data and to modify the instance data.

- Here is an extended version of the `Student` class and some code to test it out.

`student_example.py`

```
#!/usr/bin/env python3
class Student:
    """ This is the Student class file """

    def __init__(self, name, major):
        self.thename = name
        self.themajor = major

    def major(self):
        return self.themajor
    def name(self):
        return self.thename

    def setname(self, newname):
        self.thename  = newname
    def setmajor(self, newmajor):
        self.themajor = newmajor

s1 = Student("Elizabeth", "Electrical Engineering")
print(s1.major())
print(s1.name())
s1.setmajor("Computer Science")
s1.setname("Beth")
print(s1.major())
print(s1.name())
```

- Each method of Student has a parameter named `self`.
  - The parameter is a reference to the object on which the method is called.

```
s1.setmajor("Computer Science")
print(s1.major())
```

File Organization

- Classes are typically created so that the data type they represent can be reused over many programs.
  - Ideally, such a class would exist in its own file and be imported when needed.
  - An appropriate separation of the class definition, from the application that uses it, is shown here.

student01.py

```
#!/usr/bin/env python3
class Student:
    """ This is the Student class file """

    def __init__(self, name, major):
        self.thename = name
        self.themajor = major

    def major(self):
        return self.themajor
    def name(self):
        return self.thename

    def setname(self, newname):
        self.thename  = newname
    def setmajor(self, newmajor):
        self.themajor = newmajor
```

test_student01.py

```
#!/usr/bin/env python3
from student01 import Student
s1 = Student("Elizabeth", "Electrical Engineering")
print(s1.major())
print(s1.name())
s1.setmajor("Computer Science")
s1.setname("Beth")
print(s1.major())
print(s1.name())
```

Special Methods

- In addition to the `__init__` method, Python defines other special methods.
  - One such method is named `__del__`.
    - This method is called whenever an object goes out of scope.
    - This method can be used to undo what might have been done in the `__init__` method.
  - Another special method is the `__str__` method.
    - This method is called whenever the standard `str()` function is called on the object.
    - This method is also called when a reference to the object is passed to the `print()` function.

```
s1 = Student("Joe", "Accounting")
print("As a string: " + str(s1))
print(s1)
```

  - Another special method is the `__len__`.
    - This method is called when a reference to the object is passed to the `len()` function.

- Other special methods include the following.
  - `__lt__(self, obj)`
  - `__le__(self, obj)`
  - `__eq__(self, obj)`
  - `__ne__(self, obj)`
  - `__gt__(self, obj)`
  - `__ge__(self, obj)`
  - `__add_(self, obj)`
  - `__sub_(self, obj)`
  - `__mul_(self, obj)`
  - `__div_(self, obj)`
    - The above methods should only be used in classes where their inclusion makes sense.

Special Methods

- The Student class below is rewritten to include the `__str__`, `__len__`, and `__del__` methods.

`student02.py`

```
#!/usr/bin/env python3
class Student:
    """ This is the Student class file """

    def __init__(self, name, major):
        self.thename = name
        self.themajor = major

    def major(self):
        return self.themajor
    def name(self):
        return self.thename

    def setname(self, newname):
        self.thename  = newname
    def setmajor(self, newmajor):
        self.themajor = newmajor

    def __del__(self):
        print("Debug: Deleting " + self.thename)

    def __len__(self):
        length = 4 # default degree length
        if self.themajor == "Accounting":
            length = 2 # 2 year degree
        return length

    def __str__(self):
        return self.thename + " " + self.themajor
```

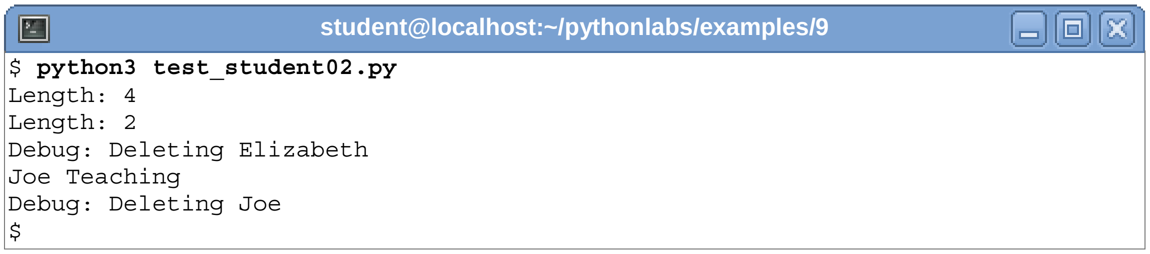
- An application that uses the above class is shown on the following page.

Special Methods

`test_student02.py`

```
#!/usr/bin/env python3
from student02 import Student
s1 = Student("Elizabeth", "Electrical Engineering")
print("Length:", len(s1))
s1.setmajor("Accounting")
print("Length:", len(s1))
s1 = Student("Joe", "Teaching")
print(s1)
```

- The output of the above program is shown below.



```
$ python3 test_student02.py
Length: 4
Length: 2
Debug: Deleting Elizabeth
Joe Teaching
Debug: Deleting Joe
$
```

- The Fraction class defined below contains several special methods.

`Fraction.py`

```
#!/usr/bin/env python3
class Fraction:
    def __init__(self, n, d):
        self.n =  n
        self.d = d
    def __str__(self):
        return str(self.n) + "/" + str(self.d)
    def __lt__(self, other):
        left = self.n / self.d
        right = other.n / other.d
        return left < right
    def __mul__(self, other):
        num = self.n * other.n
        den = self.d * other.d
        return Fraction(num, den)
```

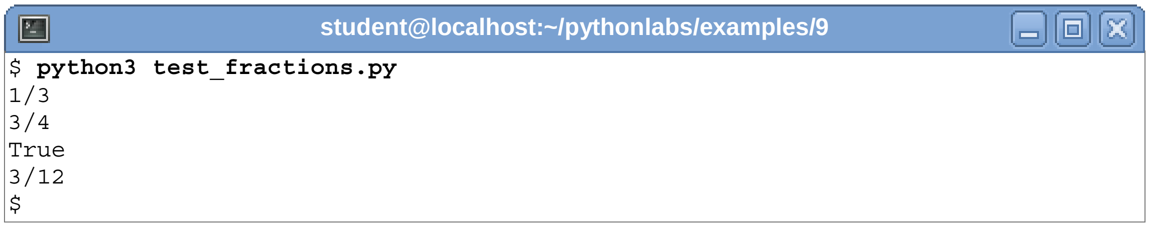
Special Methods

- The program below tests the `Fraction` class.

test_fractions.py

```
#!/usr/bin/env python3
from Fraction import Fraction

f1 = Fraction(1,3)
print(f1)

f2 = Fraction(3,4)
print(f2)
print(f1 < f2)

f3 = f1 * f2
print(f3)
```

- The output of the above program is shown below.


student@localhost:~/pythonlabs/examples/9

```
$ python3 test_fractions.py
1/3
3/4
True
3/12
$
```

Class Variables

- Class data is a part of the class but not a part of an object.
  - The example below defines a class variable named `howmany`.
    - Note that access to class variable is done by placing the class name in front of the variable name.

auto.py

```
#!/usr/bin/env python3
class Auto:
    """This is the Auto class file"""
    howmany = 0
    def __init__(self, make, model):
        self.make = make
        self.model = model
        Auto.howmany += 1
    def __del__(self):
        Auto.howmany -= 1
        print("deleting: " + self.make)
        print("count is now:", Auto.howmany)
    def count(self):
        return Auto.howmany
```

autoapp.py

```
#!/usr/bin/env python3
from auto import Auto

a = Auto("Chevy", "Malibu")
print (a.count())
b = Auto("Mazda", "Miata")
print(Auto.howmany)
```

- Note the difference in the syntax between accessing instance variables and accessing class variable.
  - Each object of type `Auto` has its own `make` and `model`.
  - Both `Auto` objects have access to the single `howmany` variable that exists in memory.

Inheritance

- Inheritance is one of the signature characteristics of object-oriented programming.
  - In designing a set of classes that is important to a set of applications, there will usually be some classes that are related to one another.
    - One kind of relationship, known as inheritance, is described by the words "is a."
  - Some examples of this inheritance relationship are shown below.
    - A Manager **is an** Employee.
    - A Directory **is a** File.
    - A Square **is a** Shape.
    - An Array **is a** Data Structure.
- The main idea behind inheritance is that the data and methods of the superclass (the base class) are directly reused by a subclass (the derived class).
  - For example, a Directory object could directly reuse File methods since a Directory is a File.
- The subclass may add newly created methods to implement additional behavior.
- Inheritance can be illustrated by deriving the `GradStudent` class from the `Student` class.
  - We will model the `GradStudent` as a `Student` with a monetary stipend.

Inheritance

- Inheritance is implemented with the following syntax.

```
class GradStudent(Student):
```

  - The code above specifies that `GradStudent` is derived from `Student`.
  - Keep in mind that every `GradStudent` object will inherit all of the methods of the `Student` class.
    - This includes, but is not limited to, the `name`, `major`, `setname`, and `setmajor` methods from the `Student` class.
  - There is one other issue.
    - The `__init__` method in the `GradStudent` class must call the `__init__` method in the `Student` class to initialize the `Student` portion of a `GradStudent` object.

gradstudent.py

```
#!/usr/bin/env python3
from student02 import Student
class GradStudent(Student):
    """ GradStudent Class """

    def  __init__(self, name, major, stipend):
        Student.__init__(self, name, major)
        self.stipend = stipend

    def getstipend(self):
        return self.stipend

    def setstipend(self, stipend):
        self.stipend = stipend

    def __str__(self):
        parent = Student.__str__(self)
        return parent + " " + str(self.stipend)
```

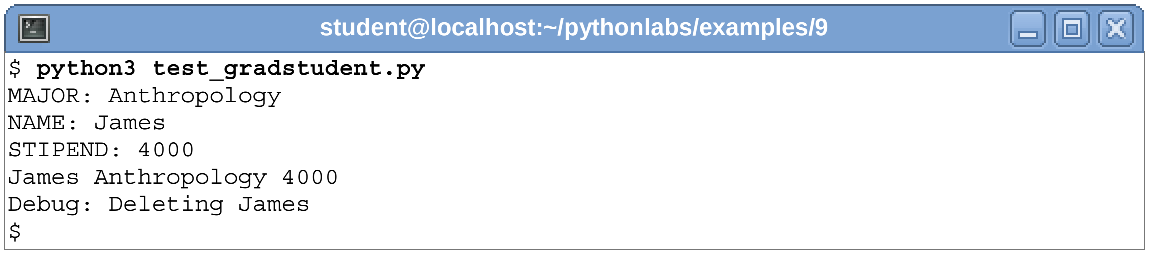
Inheritance

- Here is a program to test the `GradStudent` class.

`test_gradstudent.py`

```
#!/usr/bin/env python3
from gradstudent import GradStudent

gs1 = GradStudent("James", "Anthropology", 4000)

print("MAJOR:", gs1.major())
print("NAME:", gs1.name())
print("STIPEND:", gs1.getstipend())

print(gs1)
```

- The output of the above program is shown below.



```
$ python3 test_gradstudent.py
MAJOR: Anthropology
NAME: James
STIPEND: 4000
James Anthropology 4000
Debug: Deleting James
$
```

Polymorphism

- The word polymorphism literally means "many forms."
  - When the word is used in object-oriented programming languages, it means the ability of the language to execute a method based on the run time type of a variable.
- For example, suppose we have a `Shape` class with derived classes named `Square` and `Circle`.
  - We could certainly create objects of these derived classes.

```
s1 = Square("square1", 10)
c1 = Circle("circle1", 5)
s2 = Square("square2", 15)
c2 = Circle("circle2", 10)
```

  - Suppose further that each of the derived classes has its own `area()` method.
    - This is sensible since the `area` method is dependent upon the specific kind of `Shape`.
  - We could then create the following list.

```
shapes = [ s1, c1, s2, c2 ]
```

    - Now, we could iterate over the list as shown here.

```
for shape in shapes:
    print(shape.area())
```

  - In Python, the correct (type specific) `area` method would be called to display the area.
    - This is polymorphic behavior.
    - Python knows the run time type of each `Shape` object and calls the appropriate `area` method.
  - The entire code to implement the above is shown next.

Polymorphism

Shapes.py

```
#!/usr/bin/env python3
class Shape:
    idnumber = 100
    def __init__(self, name):
        self.name = name
        self.number = Shape.idnumber
        Shape.idnumber += 1

    def shapeId(self):
        return self.number

    def myname(self):
        return self.name

    def area(self):
        pass
```

- Three subclasses of `Shape` follow.

shape_circle.py

```
#!/usr/bin/env python3
import math
from shape import Shape

class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def __str__(self):
        return "Circle: " + self.name + \
               " Radius:" +  str(self.radius)

    def area(self):
        return math.pi * self.radius * self.radius
```

Polymorphism

shape_rectangle.py

```python
#!/usr/bin/env python3
from shape import Shape

class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def __str__(self):
        return "Rectangle: " + self.name + \
               " Length:" + str(self.length) + \
               " Width:" + str(self.width)

    def area(self):
        return self.length * self.width
```

shape_square.py

```python
#!/usr/bin/env python3
from shape_rectangle import Rectangle

class Square(Rectangle):
    def __init__(self, name, length):
        super().__init__(name, length, length)
```

- In inheritance hierarchies like the `Shape` hierarchy above, one can conceive of two kinds of methods.
  - Those whose behavior is invariant over specialization
    - The `shapeId` method is such a method and, therefore, it is not overridden in derived classes.
  - Those whose behavior varies over specialization
    - The `area` method is one such method and, therefore, it is overridden in derived classes.

Polymorphism

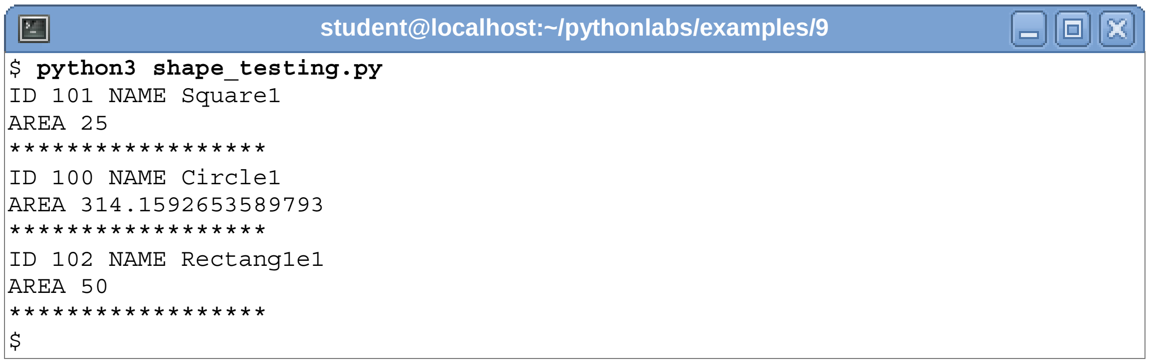
- Sometimes, a method in a superclass can be used as a default method for all the classes in the hierarchy.
  - This would be true of the `shapeId` method.
    - It has no need to be overridden in the subclasses.
  - It is also true of several of the methods in the `Rectangle` class that have no need to be overridden in the `Square` class.
    - Such as the `__str__` and `__area__` methods
- If the behavior is specialized (depending upon the class), then that function needs to be overidden in derived classes.
  - This would be true of the `area` method.
- These concepts are illustrated with the following program.

shape_testing.py

```
#!/usr/bin/env python3
from shape_circle import Circle
from shape_square import Square
from shape_rectangle import Rectangle

c1 = Circle("Circle1", 10)
s1 = Square("Square1", 5)
r1 = Rectangle("Rectangle1", 5, 10)

shapes = [ s1, c1, r1 ]
for shape in shapes:
    print("ID", shape.shapeId(), "NAME",
          shape.myname())
    print("AREA", shape.area())
    print("*****************")
```

- The output of the program is shown on the following page.

Polymorphism


student@localhost:~/pythonlabs/examples/9

```
$ python3 shape_testing.py
ID 101 NAME Square1
AREA 25
******************
ID 100 NAME Circle1
AREA 314.1592653589793
******************
ID 102 NAME Rectangle1
AREA 50
******************
$
```

Type Identification

- We have previously seen the `type` function, which identifies the type of a variable.
  - On some occasions, there is the need to determine whether an object is of one class or another.
- The `isinstance` function determines whether a particular object is of a particular class.
- The `issubclass` function determines whether one class is a subclass (direct or otherwise) of another class.

classes.py

```
#!/usr/bin/env python3
class Employee:
    pass
class Manager(Employee):
    pass
class Executive(Manager):
    pass

m = Manager()

print(isinstance(m, Employee))              #True
print(isinstance(m, Manager))               #True
print(isinstance(m, Executive))             #False

print(issubclass(Executive, Executive))     #True
print(issubclass(Executive, Manager))       #True
print(issubclass(Executive, Employee))      #True
print(issubclass(Executive, object))        #True
```

Custom Exception Classes

- Earlier in the course, you were introduced to the exception class hierarchy.
  - Each exception class was derived from the base class named `BaseException`.
- Your applications may be such that you will want to create your own exception classes.
  - These exceptions should typically be derived from the `Exception` class.
- For example, suppose you were reading strings into a program and you wished to raise an exception each time one of these strings exceeded a certain length.
  - You might begin by creating a `LongStringError` class as follows.

long_string_error.py

```
#!/usr/bin/env python3
class LongStringError(Exception):
    def __init__(self, msg, theLength):
        Exception.__init__(self, msg)
        self.theLength = theLength

    def length(self):
        return self.theLength
```

  - The class above has the usual `__init__` method as well as a method for retrieving the length of the offending string.
  - Since this exception is a custom exception, the developer would need to `raise` it at the appropriate time.

Custom Exception Classes

- Here is a program that uses the `LongStringError` class defined on the previous page.

`long_string_test.py`

```
#!/usr/bin/env python3
from long_string_error import LongStringError
max_length = 10

def checkstring(s):
    length = len(s)
    if length > max_length:
        raise LongStringError("Line too long",
                                          length)

while True:
    try:
        line = input("Enter a line: ")
        checkstring(line)
        print("line is OK")
    except LongStringError as e:
        print(e, e.length())
```

- If an input string is too long, then the exception is raised.
  - This causes a `LongStringError` object to be constructed and "sent" to the exception handler.

- A common practice for a large application is to create a base class for all of your exceptions and then create subclasses for specific exceptions.

```
class CustomBaseException(Exception):
    ...
class ExceptionOne(CustomBaseException):
    ...
class ExceptionTwo(CustomBaseException):
    ...
```

Exercises

1. Create a class called `Person`.

   - Each `Person` should have a `name`, an `age`, and a `gender`.
   - In addition to getters and setters for the above methods, the `Person` class should have a `__init__` method and a `__str__` method.
   - Therefore, the following application should test your work.

   ```
   p1 = Person("Michael", 45, "M")
   print(p1)
   ```

2. Create a class called `Family`.

   - The `Family` should be composed of two `Person` objects representing the parents and a `list` of `Person` objects representing the children.
     - Therefore, the `__init__` method should take two required parameters, followed by a variable number of arguments.
     - The first two Person objects will be the parents, and any remaining objects will be children.
     - Use the following to test your classes.

     ```
     mom = Person("Mommy", 45, "F")
     dad = Person("Daddie", 45, "M")
     kid1 = Person("Johnie", 2, "M")
     kid2 = Person("Janie", 3, "F")
     myFamily = Family(mom, dad, kid1, kid2)
     kid3 = Person("Paulie", 1, "M")
     myFamily.add(kid3)
     print(myFamily)
     ```
     - Note the `add` method in the `Family` class.

Exercises

3. Create an exception class named `FamilyError`.

   - This exception should get raised when there is an attempt to add a non-`Person` object to a `Family` object.
     - Remember that `Person` objects can be added in both the `__init__` method and the `add` method.

4. Implement the necessary special methods so that the `<`, `==`, and `>` operators can be used with Family objects.

   - The criteria for the methods should be the number of children in each `Family`.

```
myFamily = Family(mom, dad, kid1, kid2)
smiths = Family(mom, dad, kid1)
if (myFamily > smiths):
    print("we have more kids than smiths")
if (myFamily == smiths):
    print("families have same # of kids")
if (myFamily < smiths):
    print("we have fewer kids than smiths")
```

5. Implement the following class hierarchy.

   - Define a `Worker` class with a name, a salary, and number of years worked.
     - Provide a method named `pension` that returns an amount equal to the years worked times 10% of the salary.
   - Derive `Manager` from `Worker`.
     - A manager's pension is defined by the number of years worked times 20% of the salary.
   - Derive `Executive` from `Manager`.
     - An executive's pension is defined by the number of years worked times 30% of the salary.
   - Implement a `getname()` method in the Worker class and have this be a default method for all derived classes.

Chapter 10: Regular Expressions

1) Introduction.......... 10-2

2) Simple Character Matches.......... 10-4

3) Special Characters.......... 10-6

4) Character Classes.......... 10-7

5) Quantifiers.......... 10-9

6) The Dot Character.......... 10-10

7) Greedy Matches.......... 10-11

8) Grouping.......... 10-12

9) Matching at Beginning or End.......... 10-14

10) Match Objects.......... 10-15

11) Substituting.......... 10-17

12) Splitting a String.......... 10-18

13) Compiling Regular Expressions.......... 10-19

14) Flags.......... 10-20

Introduction

- When you have been writing programs for a while, certain paradigms reveal themselves, regardless of the problem domain.

- For example, it is quite common for a program to request various kinds of data from the user.
  - The program might request a:
    - name;
    - number;
    - phone number; or
    - email address.
  - Well-written programs will first validate the format of the user responses before the requested information is used in the program.

- Likewise, there are times when a program might search a set of strings with the intention of replacing a portion of the string with another one.
  - For example, you might wish to change the year part of a string (or a file name) from 2008 to 2009.

- These and similar requests can be handled via a body of knowledge known as regular expressions.
  - A regular expression is a compact notation for representing a collection of strings.
  - In Python, regular expressions are handled using the `re` module.

Introduction

- Regular expression capabilities are found in many modern programming languages.
  - One could make a strong case for Perl in terms of the ease in which regular expressions are handled.
  - In Python, the `re` module provides Perl-style regular expression patterns.
- Regular expressions are essentially a tiny, highly specialized language.
  - Using this language, you specify the rules for the set of possible strings that you want to match.
  - For example, you might specify the rules for a set of integers or a set of domain names.
    - You can then ask questions such as, "Does a string match this pattern?"
    - You can also use regular expressions to modify a string or to split it in various ways.

Simple Character Matches

- Before we use the methods in the `re` module, we need to know something about forming regular expressions using the regular expression language.

- The rules for specifying regular expression rules can quickly become complex, so we will start with some fairly simple rules.
    - Regular expressions are strings that are formed using alphanumeric characters and some non-alphanumeric characters.
        - The latter are typically treated as meta-characters in that they do not represent themselves.

- Listed here are some regular expressions that consist of alphanumeric characters.

  `the, b52, training, 15th`

    - In the case of the regular expression `the`, it would match a string that contains a single `t`, followed by a single `h`, followed by a single `e`.
        - It does not matter what comes before or after `the`.
        - It also does not matter whether there is a single or many occurrences of `the`.
        - Likewise, it does not matter whether `the` is a word or a part of a word such as `theatre` or `breathe`.

- The program on the following page demonstrates how the `re` module can be used to determine if `the` occurred in a string.

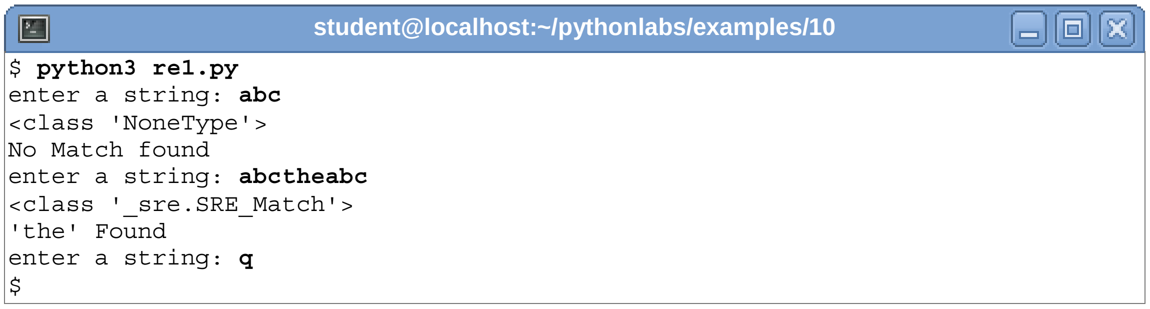
Simple Character Matches

- The `re.search` method requires a regular expression as its first argument and a string as its second argument.

re1.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("enter a string: ")
    if line == "q": break
    x = re.search('the', line)
    print(type(x))
    if x:
        print("'the' Found")
    else:
        print("No Match found")
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/10

```
$ python3 re1.py
enter a string: abc
<class 'NoneType'>
No Match found
enter a string: abctheabc
<class '_sre.SRE_Match'>
'the' Found
enter a string: q
$
```

- In the code above, the `search` method determines whether the first argument occurs within the second argument.
  - If it does not, the value `None` is returned.
  - Therefore, the code above displays all lines containing `the`.

Special Characters

- Some regular expression characters represent themselves, such as `t`, `h`, and `e`, as well as other alphanumeric characters.

- Other characters, as shown below, can have special meanings.

```
\   .   ^   $   ?   +   *   {   }   [   ]   (   )   |
```

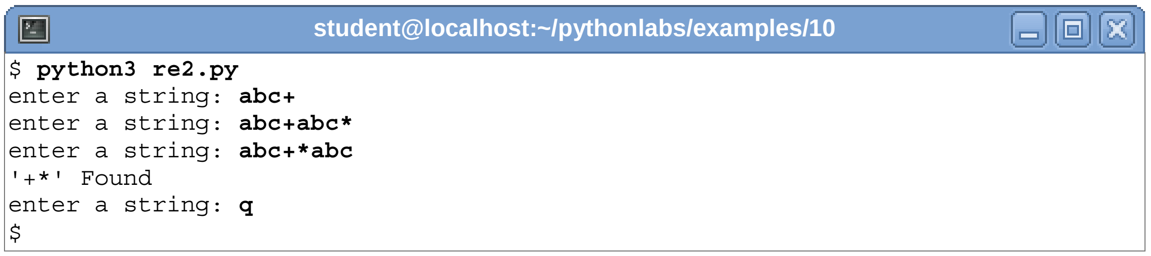
  - If you wish for any of the above characters to lose their special meaning, you need to use a `\` in front of them.

    - For example, if you wanted to display all lines containing the consecutive characters `+*`, you could use the following.

re2.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("enter a string: ")
    if line == "q": break
    x = re.search('\+\*', line)
    if x:
        print("'+*' Found")
```

  - The output of the above program is shown below.



```
$ python3 re2.py
enter a string: abc+
enter a string: abc+abc*
enter a string: abc+*abc
'+*' Found
enter a string: q
$
```

Character Classes

- A **character class** can be used to match any one of a set of characters.
  - Character classes are enclosed within the `[ ]` characters.
    - In the following regular expression, the brackets are used to match any single digit.

      ```
      x = re.search('[0123456789]', line)
      ```
  - Within a character class, you can use a dash to specify a range.
    - Therefore, the following two lines perform the identical task.

      ```
      x = re.search('[0123456789]', line)
      x = re.search('[0-9]', line)
      ```
- The following example searches for three consecutive digits.

re3.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("enter a string: ")
    if line == "q": break
    x = re.search('[0-9][0-9][0-9]', line)
    if x:
        print("3 Consecutive Digits Found")
```

- The following regular expression matches at least `hit`, `hat`, `hut`, and `that`.

```
x = re.search('h[iau]t', line)
```

Character Classes

- Note that the ^ character is used as a negation when it is the first character within the `[ ]`.

```
x = re.search('[^0-9]', line)
```

  - The above regular expression matches anything but a digit.

- There are also some character class shorthand notations.

Character Class	Meaning
`\d`	Equivalent to `[0-9]` or `[0123456789]`
`\D`	Equivalent to `[^0-9]` or `[^0123456789]`
`\s`	Any whitespace character
`\S`	Any non-whitespace character
`\w`	Equivalent to `[a-zA-Z0-9]`
`\W`	Equivalent to `[^a-zA-Z0-9]`

  - The example below matches any two consecutive digits followed by a whitespace character followed by two consecutive digits.

re4.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("enter a string: ")
    if line == "q": break
    x = re.search('\d\d\s\d\d', line)
    if x:
        print("Match Found")
```

Quantifiers

- There are times when you wish to match a certain quantity of either a character or a character class.
  - The regular expression language can express this using a quantifier.

- The various available quantifiers and their meanings can be seen in the table below.

Quantifier	Meaning
`*`	Zero or more
`+`	One or more
`?`	Zero or one
`{m,n}`	Minimum `m` and maximum `n`
`{m,}`	Minimum `m` and maximum unbounded
`{m}`	Exactly `m`

- A few examples of their uses are shown below.

Expression	Meaning
`x{2,4}`	2 to 4 consectutive `x`'s
`[ab]{3,5}`	3 to 5 `a`'s or `b`'s
`a{2}`	Exactly 2 `a`'s
`\d{5}`	Exactly 5 digits
`\w+`	1 or more word characters
`ab?c`	`abc` or `ac`

re5.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("Enter a string ")
    if line == "q": break
    x = re.search('\d{5}', line)
    if x: print("Match Found")
```

The Dot Character

- A special character that is often used in conjunction with quantifiers is the dot (.) character.
  - The dot matches any character.
  - Therefore, the first match below is more restrictive than the second one, because the second will match any one character between the `h` and the `t`, whereas the first only matches an `i`, `a`, or `u` between the `h` and `t`.

```
x = re.search('h[iau]t', line)
x = re.search('h.t', line)
```

Greedy Matches

- By default, matches are greedy.
  - That is, when there is a choice, the longest match will be chosen.
  - For example, given the pattern below, a match will occur if the target string is preceded and followed by a single space.

    ```
    line = "this is the way to do it"
    x = re.search(' .* ', line)
    ```

    - The above regular expression can be described as a space, followed by anything, followed by a space.
    - The "anything" could be nothing and could also include a blank.
  - Therefore, the pattern matches any of the following.

    ```
    | is |
    | is the |
    | is the way |
    | is the way to |
    | is the way to do |
    ```

    - Because matches are greedy, the longest match will be chosen.
    - This is true unless you make the regular expression non-greedy.
    - This is accomplished by using the question mark as shown below.

      ```
      x = re.search(' .*? ', line)
      ```

- Whether a regular expression is greedy or not has impact when remembering portions of a match.
  - This concept will be seen in more detail shortly.

Grouping

- You can use the | symbol to specify alternatives.
  - For example, if you wished to match `mike`, `chris`, or `susan`, you could code as follows.

`alternatives.py`

```
#!/usr/bin/env python3
import re
while True:
    line = input("Enter a string ")
    if line == "q": break
    x = re.search('mike|chris|susan', line)
    if x: print("Match Found")
```

- Or, notice the following use of the | symbol.

```
x = re.search('www\.(a|b|c)\.com' line)
```

  - The above regular expression matches either of the following.
    - `www.a.com`
    - `www.b.com`
    - `www.c.com`

- The parentheses are used for two different purposes.
  - Grouping
  - Capturing a match for future use
    - Each set of parentheses in a regular expression causes that portion of the match to be remembered.

- You can retrieve each remembered portion of a match by using the `re.group()` method.

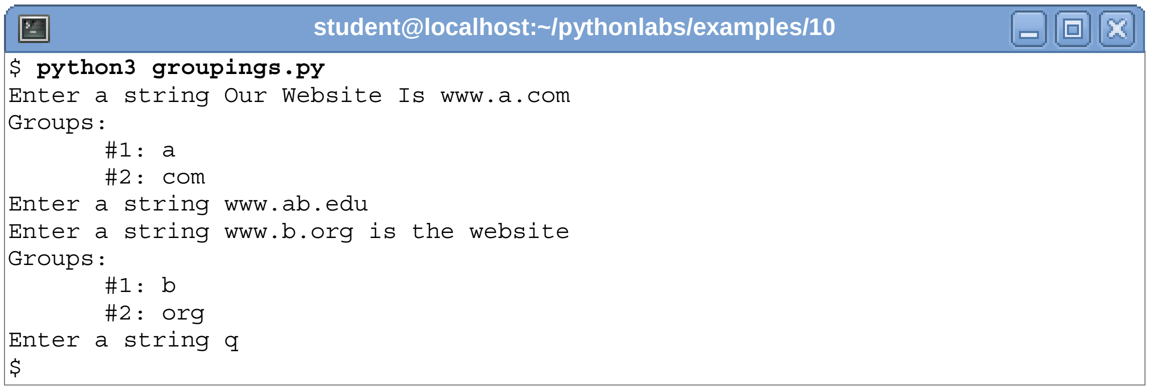
Grouping

- Here is an example featuring the remembered portion of matches.

groupings.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("Enter a string ")
    if line == "q": break
    x = re.search('www\.(a|b|c)\.(com|edu|org)',
                  line)
    if x:
        print("Groups:")
        print("\t#1:", x.group(1))
        print("\t#2:", x.group(2))
```

- The output of the above program is shown below.



```
$ python3 groupings.py
Enter a string Our Website Is www.a.com
Groups:
        #1: a
        #2: com
Enter a string www.ab.edu
Enter a string www.b.org is the website
Groups:
        #1: b
        #2: org
Enter a string q
$
```

Matching at Beginning or End

- You can use ^ at the beginning of a regular expression if you want the pattern to match only at the beginning of the target string.

match_begin.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("> ")
    if line == "q": break
    x = re.search('^\d{2,5}', line)
    if x: print("Starts with 2 to 5 digits")
```

- You can place a $ at the end of a regular expression if you want the pattern to match only at the end of the target string.

match_end.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("> ")
    if line == "q": break
    x = re.search('\d{2,5}$', line)
    if x: print("Ends with 2 to 5 digits")
```

  - The following example incorporates both the ^ and the $ so that the regular expression matches the entire string.

match_whole.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("> ")
    if line == "q": break
    x = re.search('^\d{5}(-\d{4})?$', line)
    if x: print("Is a ZipCode")
```

Match Objects

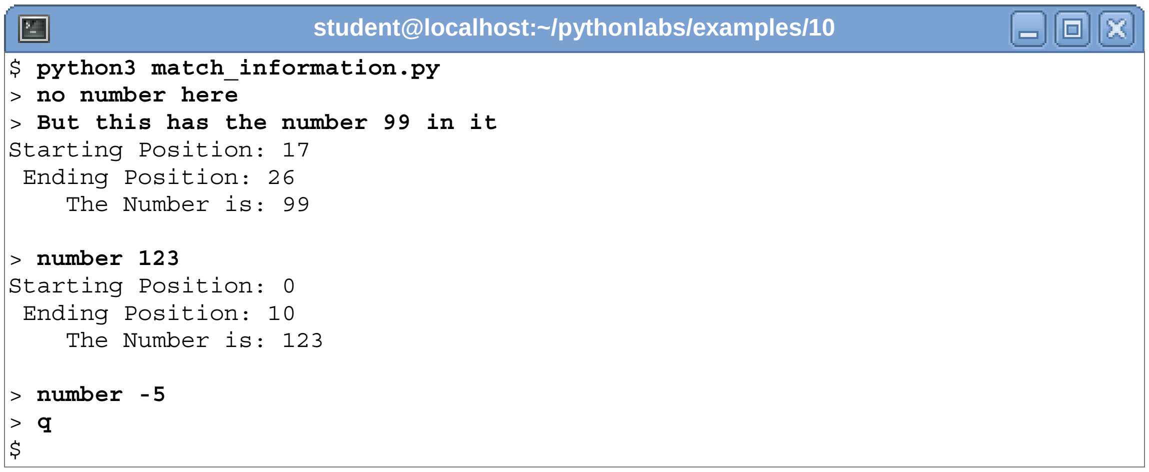
- The `re.search()` method has been used in the previous examples to determine if a string matches a pattern.
  - There is an `re.match()` method that is also available.
    - The method `re.match()` differs from `re.search()` in that it returns a match object if a match occurs at the beginning of the string.
- With either method, if there is a match, you can use the following methods on the resulting match object.
  - `start()`
    - Returns the starting position of the match
  - `end()`
    - Returns the ending position of the match + 1
  - `group(`*`index`*`)`
    - Returns the remembered pattern by *`index`*
- The program on the next page finds lines containing the string "number *xxx*”, where *xxx* is a series of digits.
  - Furthermore, the following information is supplied.
    - The starting position of the match
    - The ending position of the match
    - The number portion of the match that is remembered

Match Objects

match_information.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("> ")
    if line == "q": break
    x = re.search('number ([0-9]+)', line)
    if x:
        print("Starting Position:", x.start())
        print(" Ending Position:", x.end())
        print("    The Number is:", x.group(1))
        print()
```

- The output of the above program is shown below.



student@localhost:~/pythonlabs/examples/10

```
$ python3 match_information.py
> no number here
> But this has the number 99 in it
Starting Position: 17
 Ending Position: 26
    The Number is: 99

> number 123
Starting Position: 0
 Ending Position: 10
    The Number is: 123

> number -5
> q
$
```

Substituting

- The `re` module also gives you the ability to make substitutions via the `re.sub()` method.
  - The method takes three arguments.
    - A regular expression
    - The replacement string
    - The target string (the string to be matched)
- For example, the program below substitutes each lower case `the` with upper case `THE`.

substitutions.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("Enter a string ")
    if line == "q": break
    result = re.sub('the', 'THE', line)
    print(result)
```

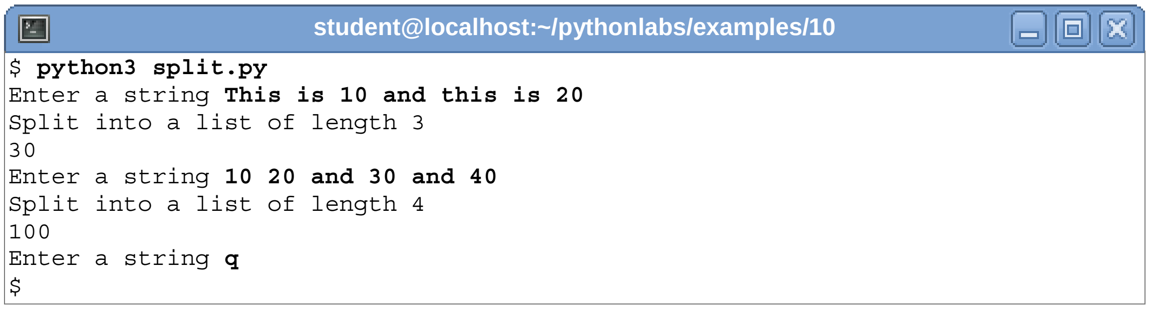
Splitting a String

- Strings can also be split based on regular expressions.
  - The result of the split will be a list of strings.
  - The idea is that `split` will use the regular expression as the delimiter between the strings that you want.
- As an example, if you wanted to split a line on non-digit characters (because you wanted to process the digits on the line), then you could proceed as follows.

split.py

```
#!/usr/bin/env python3
import re
while True:
    line = input("Enter a string ")
    if line == "q": break
    total = 0
    x = re.split('\D+', line)

    if x:
        print("Split into a list of length", len(x))
        for number in x:
            if len(number) != 0:
                total += int(number)
        print(total)
```

  - The output of the above program is shown below.



```
$ python3 split.py
Enter a string This is 10 and this is 20
Split into a list of length 3
30
Enter a string 10 20 and 30 and 40
Split into a list of length 4
100
Enter a string q
$
```

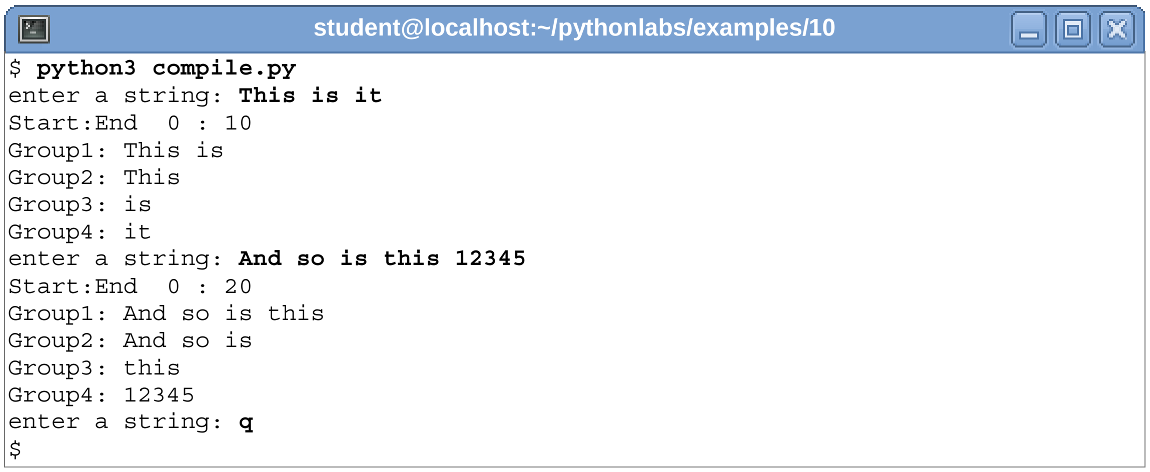
Compiling Regular Expressions

- If you need to match many strings against a single regular expression, it will be more efficient if you first compile the regular expression.
  - After doing so, the compiled regular expression can be matched against the strings in question.

compile.py

```
#!/usr/bin/env python3
import re

comp = re.compile( '((.*) (.*)) (.*)' )

while True:
    line = input("enter a string: ")
    if line == "q": break
    x = comp.match(line)
    if x:
        print("Start:End ", x.start(),":", x.end())
        print("Group1:", x.group(1))
        print("Group2:", x.group(2))
        print("Group3:", x.group(3))
        print("Group4:", x.group(4))
```

  - The output of the above program is shown below.



```
$ python3 compile.py
enter a string: This is it
Start:End  0 : 10
Group1: This is
Group2: This
Group3: is
Group4: it
enter a string: And so is this 12345
Start:End  0 : 20
Group1: And so is this
Group2: And so is
Group3: this
Group4: 12345
enter a string: q
$
```

Flags

- All three methods, `search, match,` and `compile` can take an extra argument.
  - The extra argument is called a **flag**.
    - Its purpose is to slightly alter the behavior of the search.
- For example, to make matches case insensitive, you can use either of the following flags.
  - `re.I`
  - `re.IGNORECASE`

`ignoring_case.py`

```
#!/usr/bin/env python3
import re
while True:
    line = input("Enter a string ")
    if line == "q": break
    x = re.search('hello', line, re.IGNORECASE)
    if x: print("Found a match:", line)
```

- Matches are with respect to strings, and strings can have more than one line within them.
  - For example, the string below has multiple new lines within it.

    `line = "This string\nhas three\nlines\n"`

    - The following would not yield a match because Python would look to see if the `g` was at the end of the string.

      `x = re.search('g$', line)`

Flags

- The following flags can be used if you wanted to check to see if the match was at the end of any line (not the end of the string).
  - `re.M`
  - `re.MULTILINE`

multiline.py

```
#!/usr/bin/env python3
import re

line = "This string\nhas three\nlines\n"

x = re.search('g$', line, re.M)
if x:
    print("MATCH")
else:
    print("NO MATCH")
```

Exercises

1. Write a program that reads a line at a time and determines whether the input consists solely of an integer number that is positive or negative.
    - Specify whether it is positive or negative.
2. Repeat Exercise 1, but this time use a floating point number.
    - A floating point number should have at least one digit to the left and to the right of the decimal point.
    - Specify whether the number is positive or negative.
3. Write a program that reads lines from the user one at a time to see if they are formatted according to the following criteria.
    - Correctly formatted lines should consist of a four character part number, any number of spaces or tabs, and a description.
    - The four characters should consist of two digits and two uppercase characters.
    - For each correctly formatted line, print the two digits, the two characters, and the descriptions.
        - Print all of these pieces of information on separate lines.