Python 3 Programming

Student Guide

Version 2.0

presented by
Hands On Technology Transfer, Inc.
Chelmsford, Massachusetts (USA)

Copyright © 2013-2016 Hands On Technology Transfer, Inc.
All Rights Reserved

Table of Contents

Python 3 Programming

Course Introduction

Course Goals and Objectives

Major Course Goals

Learn how to use the Python programming Language

Understand the syntax and organization of Python programs

Know the capabilities and limitations of Python

Write clear, understandable, commented code

Understand how Python can be used to write general purpose applications

Understand how Python can be used to write GUI applications

Understand how Python can be used to write Web applications

Specific Objectives

Write Python programs

Work with Python's built-in objects

Read and Write files

Communicate with Database's

Create structured data with Lists, Tuples, Sets and Dictionaries

Use exceptions to gracefully handle errors

Test program features using Test Driven Development

Write applications with Graphical User Interfaces

Write clear and concise Regular Expressions

Organize code with Functions and Classes

Module 1

Understanding Python

Module Goals and Objectives

Major Goals

Understand why Programs are developed with Python

Learn where to obtain Python

Install Python

Run Python Programs

Understand the Python Help System

Specific Objectives

Be able to describe how Python differs from other Programming Languages

Install Python on a Windows Operating System

Run Python programs from the shell

Run Python Programs interactively

Use the help system interactively

Request specific help system topics

Section 1–1

Understanding Python

Python Overview

- **Python is a interpreted, general-purpose, high-level, object-oriented programming language**
  - Emphasizes code readability
  - Flexible and readable syntax allows programmers to use fewer lines of code to express programming concepts
  - Used for a wide variety of programming tasks
- **Python supports multiple programming paradigms**
  - Although Python is an Object-Oriented programming language, Python supports object-oriented, functional, and imperative programming styles
- **Python is an Open Source Programming Language**
  - Freely usable and distributable
    - The Python license is administered by the Python Software Foundation

History of Python

- **Python was conceived by Guido van Rossum in the late 80's as a scripting language for the Amoeba Operating System**
  - Guido van Rossum is also known as the Benevolent Dictator For Life (BDFL)
- **Python has a basis in many programming languages including C, Modula-3 and the educational programming language ABC**
- **Python was released for internal use in 1990 at the National Research Institute for Mathematics and Computer Science (CWI) in the Netherlands**
- **Python was released in February, 1991 to the Usenet group alt.sources**
- **Python is named for the British Comedy Group "Monty Python's Flying Circus"**
- **Python has evolved through multiple versions over the last few decades**
  - Python 2.6/7 and Python 3+ are popular releases of Python
  - Python 3 is the future of Python

Why use Python?

- **Can be used to solve a number of computing problems**
- **Large Developer base**
  - Active support from a large and dynamic programming community
- **True Cross-Platform Programming language**
  - Runs well on Linux/Unix, Windows, Mac, Mobile Devices, Super Computers
- **Has a large standard library**
  - Data Type Management
  - File and Directory Access
  - Operating System and Inter Process Communication
  - Structured Markup Processing
- **Software Quality**
  - Python code is readable which leads to ease of maintenance and reusability
  - Python code is also uniform which makes it easier to understand

Who uses Python

- **Python is used for a wide range of application development including Games, Web Development, Financial Applications , Scientific and Mathematical Applications, Framework Development, etc...**
  - http://wiki.python.org/moin/OrganizationsUsingPython
- **Google**
  - Uses Python for a number of their application frameworks
  - Philosophy of "Python where we can, C++ where we must"
  - Employed Guido van Rossum until late 2012
    - Guido spent 50% of his time while at Google working on the Python programming language
- **Applications such as Maya, GIMP, Paint Shop Pro, Cinema 4D**
  - Use Python as a Scripting Language
- **Python is a standard component in most Linux Operating Systems**
- **Rackspace's Cloud SDK has a Python implementation known as Pyrax**
- **YouTube is predominantly written in Python**

Advantages of using Python

- **Python is Easy to learn**
  - Python is based off of the teaching language ABC
- **Python is easy to read**
  - Python requires the use of white space and indentation as part of its syntax
  - The use of white space as part of Pythons syntax dramatically increases its readability
- **Python is designed for rapid application development**
  - Python programs take a fraction of the amount of time to develop when compared to comparable C or Java Programs
- **Python has a well-planned design and consistent evolution of features**
- **May be integrated with other languages**
  - Takes advantage of strengths and weaknesses of multiple languages
- **Python programs are portable**
  - The CPython implementation of Python is written in ANSI C
  - CPython compiles and runs on almost every major platform

Python was designed to be easy to use

- **Python has very simple syntax rules**
  - Uses white space as part of the core syntax
  - The use of white space as part of the core syntax makes Python
    - Easier to read
    - Easier to maintain
    - Easier to modify
- **Data Types are associated with Objects**
  - Variables may be assigned any of Python's existing objects
  - Python's objects represent the most commonly used data types in programs such as, Numbers, Strings, Lists, Dictionaries, Files, etc...
- **Python is Dynamically Typed**
  - Python doesn't require complicated data type and size declarations in your code
  - Python keeps track of the types of objects your program uses when your program runs
- **Python uses automatic memory management**
  - Python uses a reference-based garbage collection system that reclaims objects when they are no longer being used

Python Readability vs. Perl Readability

- **The following code demonstrates the readability differences between Perl and Python**
  - It doesn't matter that we don't know Python yet. Which set of code looks easier to read?

Example

```
# Perl version
sub pair_sum {
    my($a1, $a2) = @_;
    my(@res) = ();
    @l1 = @$a1;
    @l2 = @$a2;
    for($i=0; $i < length(@list1); $i++) {
        push(@res, $l1[$i] + $l2[$i]);
    }
return(\@res);
}

# Python version.
def pair_sum(l1, l2):
    res = []
    for i in range(len(l1)):
        res.append(l1[i] + l2[i])
    return res
```

The Zen of Python

- **Python's Design goals may be summarized by the Zen of Python**

- **PEP 20 (Python Enhancement Proposal) written by Tim Peters describes Guido van Rossum's guiding principles for Python development**

  – http://www.python.org/dev/peps/pep-0020/

1. Beautiful is better than ugly.
2. Explicit is better than implicit.
3. Simple is better than complex.
4. Complex is better than complicated.
5. Flat is better than nested.
6. Sparse is better than dense.
7. Readability counts.
8. Special cases aren't special enough to break the rules.
9. Although practicality beats purity.
10. Errors should never pass silently.
11. Unless explicitly silenced.
12. In the face of ambiguity, refuse the temptation to guess.
13. There should be one-- and preferably only one --obvious way to do it.
14. Although that way may not be obvious at first unless you're Dutch.
15. Now is better than never.
16. Although never is often better than *right* now.
17. If the implementation is hard to explain, it's a bad idea.
18. If the implementation is easy to explain, it may be a good idea.
19. Namespaces are one honking great idea -- let's do more of those!

- **For a special treat, type `import this` in an interactive shell**

Python Implementations

- **An "implementation" of Python should be taken to mean a program or environment which provides support for the execution of programs written in the Python language**

- **CPython**
  - Default and most-widely used implementation of Python
  - Written in C
  - Runs on most Operating Systems

- **Jython**
  - Compiles into Java byte code
  - Can then be executed by every Java Virtual Machine implementation.
  - May use Java class library functions from the Python program

- **IronPython**
  - Written in C#
  - Allows Python programs to be run on the .NET Common Language Runtime

List of Python Implementations

- **PyPy**
  - Written in Python
  - Python Interpreter and Just in Time Compiler
  - JIT compiles Python into machine code at run time
  - Primary focus is fast and efficient code compatible with the CPython compiler
    - Often used as a replacement for CPython when programs need to run faster
  - Successor to the original PsychoJIT
- **Stackless**
  - Enhanced version of CPython geared towards concurrency
  - Makes Python easier to port to small stack architectures
  - Uses microthreads which are lightweight, and efficient alternatives to Python's built-in multi-tasking tools
- **Pyjamas**
  - Standalone Python-to-JavaScript Compiler
  - Contains Ajax Framework and Widget Toolkit
  - Developers can write client-side applications in Python that are compiled to JavaScript
  - Pyjamas is a port of the Java-based Google Web Toolkit to Python

Python 3

- **Python 3 (or Py3K or Python 3000) is the first version of Python that is not backwards compatible with previous versions**
  - May cause problems when running Python 2 programs or Python 3 programs that use modules that rely on Python 2
- **The changes to Python 3 syntax were designed to make Python**
  - More readable
  - More consistent
  - Less ambiguous
- **The syntax changes between Python 2 and Python 3 are fairly small**
  - Small changes to syntax can cause big problems
- **Python 3 ships with the `2to3` library**
  - `2to3` is a Python program that reads Python 2.x source code and applies a series of fixers to transform it into valid Python 3.X code

What Python doesn't do well

- **Python isn't the fastest language**
  - Python is an interpreted programming language
  - Python Programs are semi compiled into an internal byte-code
  - The byte-code is then executed by the Python interpreter
- **C, Java, and Perl all have larger collections of libraries**
  - Python does not have the most extensive collection of libraries
  - Python is extensible, so any module that doesn't exist, you could quickly create yourself
- **Python does not check variables at compile time**
  - Variables are treated as labels that reference objects
  - It is possible to have a variable refer to different objects at different points in your code

Section 1–2

Obtaining and Installing Python

Obtaining Python

- **Python can be downloaded from the Python Website**
  - http://www.python.org/download
- **Current Production versions of Python are**
  - Python 2.7.x
  - Python 3.3.x
- **If you are using Python for the first time, install a 3.x version of Python**
- **Install Python 2.6/7.x if you are working with existing projects that rely on legacy libraries or 3rd party software that doesn't support Python 3**
- **Multiple installers for Python are found on the Download page on the Python website**
  - 32/64-bit Windows Installers
  - Multiple 32/64-bit installers for multiple versions of Mac OS X
  - Installation files for Unix and Linux Platforms
- **Links to Python Implementations and PGP keys may also be found on the Download page**

Installing Python on Windows

- **Installing Python on Windows is a straightforward process**
- **Download the appropriate Windows Installer Package (MSI) for your CPU type from the Downloads page on the Python site**
- **Execute the MSI**
- **Choose which users can use Python**

_

Installing Python on Windows

- **Choose the Destination Directory**

Installing Python on Windows

- **Choose Customization settings for Python**
  - You may choose settings at this point such as
    - Installing TCL/TK
    - Adding Python to the Operating System Search Path
    - Install Documentation, etc...
  - 

Installing Python on Windows

- **Head out for a cup of coffee while the installer completes its job**

- **When the Installer is finished, launch Python**
  - Start > All Programs >Python 3.x > Python (Command Line)
  - Start > Run > Cmd > Python

- **For further documentation, refer to the Python website**
  - http://docs.python.org/3/using/windows.html

Installing Python on Mac

- **Python 2.x is installed on Mac OS X 10.x by default**
  - For instance, Python 2.7 is installed on Mac OS X 10.8 by default
  - To use the existing Python interpreter go to Applications > Utilities > Terminal
  - You may install Python 3 side by side with the 2.x installation
- **To obtain Python for Mac go to the downloads page on the Python.org site**
  - http://www.python.org/download/
- **Obtain a version of Python that matches your OS X version and your Processor or the "Universal Binary" file that runs natively on Intel and PPC architectures**
  - Download the Disk image file
  - Double Click to mount the image file
  - Run the installer inside the image file
- **If Python is successfully installed, a new Python subfolder will be added to the Applications folder**
- **For further information about installing on Mac consult the Python website**
  - http://docs.python.org/3/using/mac.html#getting-and-installing-macpython

Installing Python on Linux

- **Most Linux distributions come with Python installed**
- **It is possible that Python 3 and/or IDLE is not installed as part of your Linux Operating system**
  - To check which version of Python is installed type `python -V` in the shell
  - You may also invoke the interactive interpreter by typing `python`
    - The interactive interpreter reports the version of Python as part of its invocation
- **While compiling Python from source is possible, it is not a procedure that is intended for novices.**
  - Download a release from `http://python.org/download/releases/` and extract the files
    - run ./configure script
    - make
    - make install
  - This will install python in a standard location `/usr/local/bin` and its libraries are installed in `/usr/local/lib/pythonXX` where XX is the version of Python that you are using
  - For more information about building from source check out `http://docs.python.org/3/using/unix.html#building-python`

Installing Python on Linux

- **Optionally you may use the package manager on your computer to install Python**

- **Check the `Python.org` website to see if a precompiled version exists for your distribution or use your package manager to search for Python packages**

- **For further information about installing Python on Linux or how to compile Python from source consult Python.org website**

  – http://docs.python.org/3/using/unix.html

Section 1–3

Running Python Programs

The Python Interpreter

- **When the Python Package is installed it generates a number of components including**
  - The Python Interpreter
  - The Python Standard Library
- **An interpreter is a program that executes other programs**
- **The Python interpreter reads your programs and executes your programs' instructions**
- **Python programs are primarily text files that contain Python commands**
  - Python programs have a `.py` extension
  - Technically, the .py extension is only necessary for Python Programs that are imported, but most Python developers maintain the file extension for consistency
- **Python code may also be compiled into C or another programming language, depending on which implementation of Python you are using**
  - Jython compiles Python code to Java Byte Code
  - IronPython can compile Python code to MSIL code to work with the .NET framework

How Python runs a Program

- **When a program is executed, Python reads the source code and compiles the source code into byte code**

- **Byte code is a low-level , platform-independent representation of your code**
  - The statements in your source code are translated into optimized sets of step by step instructions for the PVM
  - If your Python program is imported, then the Python interpreter will create and save the compiled byte code into a `.pyc` file

- **Once the program has been parsed and compiled, the compiled code is passed to the Python Virtual Machine (PVM) for execution**

- **The Python Virtual Machine is the runtime engine of Python**
  - Always present with all installations of Python

- **The PVM is the component that "runs" Python programs**

Executing Python

- **Python code may be executed a number of ways**
  - Script Mode
  - Interactive Mode
- **Scripts may be executed from the Command Line**
  - If you have multiple versions of Python installed, you may specify which version you want to execute

Example

```
HOTT@HandCannon ~
$ python -V
Python 2.7.3

HOTT@HandCannon ~
$ python3 -V
Python 3.2.3
```

- **Python may run in Interactive mode**
- **Most Python IDE's allow execution of the Python Interactive mode**
- **Utilities such as py2exe and PyInstaller compile Python code to a binary Executable**
  - Also known as "Frozen Binaries"

Executing Python from the Command Line

- **The syntax for invoking a Python script from the command line is:**

Syntax

```
python [option*]
[ scriptfilename| -c command| -m module| - ] [arg*]
```

Example

```
C:\HOTTPython\Mod01\Solutions>python hello.py
Hello, World!
```

- **If Python is not in your operating search path then modify the python statement to read `/path/to/python` where `/path/to/` is the location where Python is installed on your Operating System**

- **If Python is not in your Operating System search path you may add the path to Python to your Operating System search path**

Example

```
#In windows at the command prompt type:
path %path%;C:\Python # or whatever your path to Python is

#In sh or ksh bash
PATH="$PATH:/usr/local/bin/python"

#in bash type
export PATH="$PATH:/usr/local/bin/python"
```

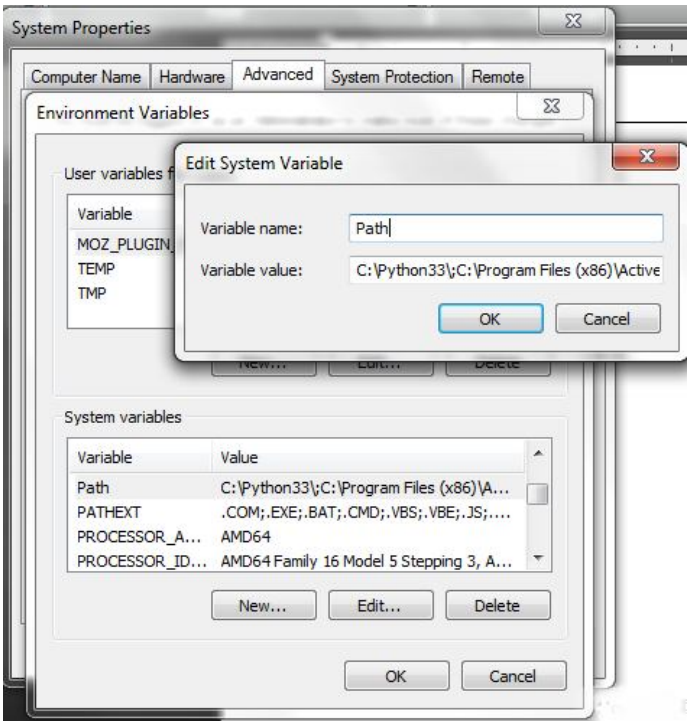
Python Environmental Variables

- **Environmental Variables are system settings that live outside of Python as part of the operating system**

- **May be used to customize the interpreter's behavior each time it runs**

- **The following table is a summary of the Main Python environmental variables**

Environmental Variable	Explanation
PATH(or path)	System shell search path (for finding "python")
PYTHONPATH	Python module search path (for imports)
PYTHONSTARTUP	Path to Python interactive startup file
TCL_LIBRARY, TK_LIBRARY	GUI extension variables (tkinter)



Environmental Variables Explanation

- **The PATH setting lists a set of directories that the operating system searches for executable programs**

- **The PYTHONPATH variable is used to locate module files when you import them in a program**
  - The following example sets the PYTHONPATH on a Windows System

Example

```
set PYTHONPATH=%PYTHONPATH%;C:\My_python_lib
```

- **PYTHONSTARTUP should be set to the pathname of a Python Program you want executed every time the interactive interpreter starts**

- **TCL_LIBRARY, TK_LIBRARY**
  - You may need to set the names of the source library directories of the Tcl and Tk systems if you want to use the GUI Libraries
  - These configuration settings should be unnecessary on Windows systems because tkinter support is installed alongside Python

Python Command Line Options

- **The following table is a partial list of commonly used Python command line options**

-b	Issues warnings for calling str() with a bytes or bytearray object, and comparing a bytes or bytearray with a str
-B	Do not write .py[co] files on imports
-c cmd	Specifies string be run as a command
-d	Turns on parser debugging output
-E	Ignores Python environment variables described ahead
-h	Prints help message and exit
-i	Enters interactive mode after executing a script. Useful for postmortem debugging
-m	Runs library module as a script: searches for module on sys.path, and runs it as a top-level file
-o	Optimizes generated byte-code (create and use .pyobytecode files). Currently yields a minor performance improvement
-v	Prints a message each time a module is initialized, showing the place from which it is loaded; repeats this flag for more verbose output
-V	Prints Python version number and exit
-	Reads Python Commands from STDIN

Running Python3 on Linux with option Example

```
HOTT@HandCannon ~
$ python3 -h
usage: python3 [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-q     : don't print version and copyright messages on interactive startup
-R     : use a pseudo-random salt to make hash() values of various types be
         unpredictable between separate invocations of the interpreter, as
         a defence against denial-of-service attacks
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : unbuffered binary stdout and stderr; also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ':'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>:<exec_prefix>).
               The default module search path uses <prefix>/pythonX.X.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONHASHSEED: if this variable is set to 'random', the effect is the same
   as specifying the -R option: a random value is used to seed the hashes of
   str, bytes and datetime objects.  It can also be set to an integer
   in the range [0,4294967295] to get hash values with a predictable seed.
```

Running Python 3 on Windows with Option example

```
C:\HOTTPython>python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-b     : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B     : don't write .py[co] files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd : program passed in as string (terminates option list)
-d     : debug output from parser; also PYTHONDEBUG=x
-E     : ignore PYTHON* environment variables (such as PYTHONPATH)
-h     : print this help message and exit (also --help)
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod : run library module as a script (terminates option list)
-O     : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO    : remove doc-strings in addition to the -O optimizations
-q     : don't print version and copyright messages on interactive startup
-s     : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S     : don't imply 'import site' on initialization
-u     : unbuffered binary stdout and stderr; also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v     : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-V     : print the Python version number and exit (also --version)
-W arg : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x     : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt : set implementation-specific option
file   : program read from script file
-      : program read from stdin (default; interactive mode if a tty)
arg ...: arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';'-separated list of directories prefixed to the
               default module search path.  The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
   to seed the hashes of str, bytes and datetime objects.  It can also be
   set to an integer in the range [0,4294967295] to get hash values with a
   predictable seed.
```

Python Program Specification Arguments

- **`ptyhon scriptfilename`**
  - Denotes the name of a Python program to execute as the main, topmost file of a program execute (e.g., pythonmain.py)

- **`python -c command`**
  - Specifies a Python command (as a string) to execute

Example

```
C:\HOTTPython>python -c "print('spam ' * 3)"
spam spam spam
```

- **`python -m command`**
  - Runs library module as a script: searches for module on sys.path, and runs it as a top-level file

- **`python -` or `python`**
  - Reads Python commands from stdin(the default)
  - Enters interactive mode if stdin is a tty

- **`arg*`**
  - Indicates that anything else on the command line is passed to the scriptfile or command

Example: Executing Python Script with Script Arguments

Example

```
# This is code from the shell

$ python3 argv.py Hello World
argv.py
Hello
World
```

Example

```
#Source File argv.py

#!/usr/local/bin/python

import sys
for arg in sys.argv:
    print(arg)
```

Changing Permissions for Python Programs on Linux

- **To easily use Python scripts on Unix, you need to make them executable**

Example

```
$ chmod +x script
```

- **Put an appropriate Shebang line at the top of the script**
  - The Shebang line is the path to the Python interpreter
  - You may also use the `env` program to try and locate the path to the Python interpreter

Example

```
#!/usr/local/bin/python

#!/usr/bin/env python
```

- **Your Python program may now be executed by using `./program.py`**

Example

```
$ ./argv.py A B C
./argv.py
A
B
C
```

Running Python in Interactive Mode

- **The Python Interpreter has the ability to run in an interactive mode**
- **In interactive mode a command line shell gives immediate feedback for each statement while running previous statements**
  - As new commands are entered into the interactive prompt, the new commands are executed with the previous commands
- **To start the interactive mode type `python` into a shell without any arguments**
- **In interactive mode, Python prompts for commands using the Primary Prompt >>>**
- **For command continuation lines , Python prompts with the Secondary Prompt . . .**

Example

```
$ python
Python 2.7.3 (default, Dec 18 2012, 13:50:09)
[GCC 4.5.3] on cygwin
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Example: Issuing Commands in Python Interactive Mode

- **The following example uses the Primary Prompt to issue commands interactively to Python**

Example

```
$ python
Python 2.7.3 (default, Dec 18 2012, 13:50:09)
[GCC 4.5.3] on cygwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello, World!")
Hello, World!
>>> 2+2
4
>>> x = 2 + 2
>>> print(x)
4
>>> exit()
```

Interactive Mode Primary and Secondary Prompt Example

- **The following example shows the use of the Primary and Secondary Prompts in the Python Interactive mode**

Example

```
$ python
Python 2.7.3 (default, Dec 18 2012, 13:50:09)
[GCC 4.5.3] on cygwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> k = "Sir Lancelot"
>>> q = "Holy Grail"
>>> c = "Blue"
>>> if( k == "Sir Lancelot" and q == "Holy Grail" and c ==
"Blue"):
...     print("Go on.")
...     print("Off you go.")
...
Go on.
Off you go.
```

- **To exit the secondary prompt send an extra new line to the interpreter after your last command**

Opening a Python program in Interactive Mode

- **You may also enter into an interactive session after executing a script**
  - This is useful if you want to perform some postmortem debugging
  - Also useful to prototype new code
- **To open an interactive session, use the `python -i` argument**
  - The `-i` argument tells python to enter into interactive mode after executing a script

Example

```
python -i scriptname.py
```

Example

```
C:\HOTTPython>python -i helloworld.py
Hello, World!
>>> print("Hello again")
Hello again
>>> 2+2
4
>>> exit()
```

Section 1–4

Python Basic Skills

Creating Variables

- **To create a variable, place an identifier that follows Python's naming convention on the left side of the assignment operator and the object you want to reference on the right hand side**

- **Variables in Python are really references**
  - More on that subject a little later in the course

- **The values on the right side of the assignment operator are really objects**
  - We will discuss this concept a little later as well

Example

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43)
[MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
information.
>>> a = "Hello"
>>> b = "World"
>>> c = 7
>>> d = 123.45
>>> print(a, b, c, d)
Hello World 7 123.45
>>>
```

Strings

- **In Python, Strings are sequences of characters**

- **To create a string object, bind a sequence of characters with Single Quotes ' or Double Quotes "**

Example

```
"Hello, World"
'Well, Hello, yourself!'
```

- **To create a variable with a string object, simply assign the string object to a variable**

Example

```
>>> a = "Hello"
>>> b = 'World!'
>>> print(a, b)
Hello World!
```

Numbers

- **Python supports many types of Number objects**
  - Integers
  - Floating Point Values
  - Complex Numbers
  - Fractions
  - Booleans
- **To create a variable with a number object, simply assign the number to a variable**

Example

```
>>> a = 1
>>> b = 2
>>> print(a+b)
3
>>> print(a/b)
0.5
```

- **Python also supports a standard Mathematical operator set so you may perform Arithmetic and Comparison operations with your Number objects**

Python 3 `print()` function

- **In Python 3 printing takes the form of a built-in function**
  - In Python 2 and earlier, `print` is a statement and has a different syntax from the Python3 `print()` function.
    - This can cause syntax failures when porting code from Python 2.x to Python 3.x

- **The `print()` function takes a list of objects as its argument, appends word separators and line separators, and redirects the output to a `file` object (with `sys.stdout` being the default `file` object)**

Syntax

```
print(value [,value]* [,sep = string] [,end = string] [,
file = file])
```

- **Each value is an object to be printed**

- **`Sep` is an optional string to place between values**
  - Defaults to a space

- **`End` is a string to place at the end of the text printed**
  - Defaults to a newline '\n'

- **`File` is a file-like object that the text is written to**
  - Defaults to sys.stdout

Example: The Python 3 `print()` function

- **Example**

```
#!/usr/bin/Python3

p1 = 'Graham Chapman'
p2 = 'Eric Idle'
p3 = 'Terry Gilliam'
p4 = 'Terry Jones'
p5 = 'John Cleese'
p6 = 'Michael Palin'

print(p1,p2,p3,p4,p5,p6)              # Print MPFC
print()                               # Creates a blank line
print()                               # Creates a blank line
print(p1,p2,p3,p4,p5,p6, sep = ',') # Print MPFC as CSV
print(p1,p2,p3,p4,p5,p6, sep = ',', end = '\nSPAM SPAM
SPAM\n')
# Print MPFC with some spam at the end
```

Output

```
*** Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013,
00:06:53) [MSC v.1600 64 bit (AMD64)] on win32. ***
>>>
Graham Chapman Eric Idle Terry Gilliam Terry Jones John
Cleese Michael Palin


Graham Chapman,Eric Idle,Terry Gilliam,Terry Jones,John
Cleese,Michael Palin
Graham Chapman,Eric Idle,Terry Gilliam,Terry Jones,John
Cleese,Michael Palin
SPAM SPAM SPAM
```

The Python 2 `print` Statement

- **The Python 2, `print` is a statement and not a built-in function**
  - In Python 2, the `print` statement does not wrap its arguments in parenthesis

- **The Python 2 `print` statement displays the printable representation of its arguments on the `stdout` stream (current value of `sys.stdout`) with spaces added as separators**
  - An optional comma as the last argument for print will suppress the linefeed normally added to the end of the printed arguments

Syntax

```
print [value [,value]* [,]]
```

Example

```
HOTT@HandCannon ~
$ python
Python 2.7.3 (default, Dec 18 2012, 13:50:09)
[GCC 4.5.3] on cygwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> a = "John Cleese"
>>> b = "Michael Palin"
>>> print a, b
John Cleese Michael Palin
>>> print a, b,
```

Gathering input from the user the Python 3 way

- **One of the easiest ways to gather input from the user is to use the `input()` function**

Syntax

```
input([prompt])
```

- **The `input()` function prints the prompt string if the prompt string is given**

- **The `input()` function then reads a line from the `stdin` input stream (`sys.stdin`) and returns it as a string with the trailing `\n` stripped**

Example

```
#!/usr/bin/Python3

first = input("What is your first name?")
last = input("What is your last name?")

print("Your name is " , first, last)

#Output Your name is  HOTT Student
```

Gathering input from the user the Python 2 way

- **In Python 2 the equivalent of the `input()` function is the `raw_input()` function**

Syntax

```
raw_input([prompt])
```

- **The behavior of the `raw_input()` function in Python 2 is the same behavior described for the Python 3 `input()` function**

- **Python 2 does have an input function, but it accepts a string and evaluates the string as Python code**
  - Equivalent to Python 2 `eval(raw_input([prompt]))`
  - Equivalent to Python 3 `eval(input([prompt]))`

Example

```
Python 2.7.3 (default, Dec 18 2012, 13:50:09)
>>> f = raw_input("What is your first name?")
What is your first name?HOTT
>>> l = raw_input("What is your last name?")
What is your last name?Student
>>> print "Your name is " ,f , l
Your name is HOTT Student
```

Example: `input()`

Example

```
#!/usr/bin/Python3
# inputExample.py
print("Stop. Who would cross the Bridge of Death must
answer me these questions three, ere the other side he
see.")
print("Ask me the questions, bridgekeeper. I am not
afraid.")
name = input("What... is your name?")
quest = input("What... is your quest?")
color = input("What... is your favourite colour?")

if color == 'blue':
    print("Go on. Off you go.")
else:
    print("Into the Volcano!")

#Output
#Running program from Windows Command Prompt

c:\HOTTPython\Mod01\Examples>python inputExample.py
Stop. Who would cross the Bridge of Death must answer me
these questions three,
ere the other side he see.
Ask me the questions, bridgekeeper. I am not afraid.
What... is your name?My name is Sir Lancelot of Camelot
What... is your quest?To seek the Holy Grail
What... is your favourite colour?blue
Go on. Off you go.
```

Python's Standard Library

- **Python comes with a large collection of utility modules known as the Standard Library**
  - The Standard Library contains over 200 modules
- **The modules in the Standard Library contain platform-independent support for common programming tasks**
  - OS Interfaces
  - Internet and Network Programming
  - GUI Construction
- **Modules provide a convenient way to group related sets of components into a self-contained package**
- **Modules are just Python files**
- **Modules are imported into Python programs by using the `import` statement**

The `import` statement

- **The `import` statement imports modules as a whole**

Syntax

```
import module [,module]*
import module as name [,module as name]
```

- **Modules contain attributes which may be accessed through a qualified path statement (`module.attribute`)**
  - Assignments to variables at the top level of the module create the module's attributes
- **The `as` clause of the import statement assigns a variable name to the imported module object**
  - This is useful to create a shorter synonym for the imported module
- **The `from` statement imports variable names from a module into the namespace of the current program**

Syntax

```
from module import name [,name]*
from module import *
```

  - This is useful so variables may be used without a fully qualified name
  - May cause namespace collisions

Example: Importing Modules

Example

```
#!/usr/bin/Python3
import sys
# This is the sys module from the Standard Library
import inputExample
# This is a call to the inputExample.py Program
# Code for this Program may be found on the slide
# Example: input()

print(inputExample.color)
print(inputExample.name)
print(inputExample.quest)

print("The module search path is " , sys.path)
print("The Current Operating System is", sys.platform)


"""
C:\HOTTPython\Mod01\Examples>python import.py
Stop. Who would cross the Bridge of Death must answer me
these questions three,
ere the other side he see.
Ask me the questions, bridgekeeper. I am not afraid.
What... is your name? Sir Lancelot
What... is your quest? Holy Grail
What... is your favourite colour? blue
Go on. Off you go.
blue
Sir Lancelot
Holy Grail
The module search path is
['C:\\HOTTPython\\Mod01\\Examples', 'C:\\Windows\\sys
tem32\\python33.zip', 'C:\\Python33\\DLLs',
'C:\\Python33\\lib', 'C:\\Python33',
 'C:\\Python33\\lib\\site-packages']
The Current Operating System is win32
"""
```

Section 1–5

Getting Help in Python

Documentation at docs.python.org

- **Major documentation for Python may be found at `http://docs.python.org`**
- **Documentation for specific releases may be accessed from the Python documentation portal**
  - http://docs.python.org/release/3.2/index.html
- **For questions about using modules consult the Standard Library Reference**
- **For comprehensive details about the implementation of the language consult the Language Reference**
- **The Python Usenet group is comp.lang.python**
- **For a list of mailing lists and groups consult `http://www.python.org/community/lists/`**

The `help()` function

- **The `help()` function allows a developer to invoke an interactive help system or return specific documentation**
  - `help()` is one of the interfaces to the standard `PyDoc` tool

syntax

```
help([object]|['string'])
```

- **If no argument is given, the interactive help system starts on the interpreter console**
  - To receive a list of all modules, keywords or topics enter the strings "modules", "keywords" or "topics" into the interactive help prompt
  - To receive help for a specific module, keyword or topic, enter the specific module name, keyword or topic into the interactive help prompt
- **If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console**
- **If the argument is any other kind of object, a help page on the object is generated**

Example: Using the `help(object)` Function

Example

```
C:\Users\>python
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43)
[MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
    flush=False)

    Prints the values to a stream, or to sys.stdout by
    default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the
    current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a
    newline.
    flush: whether to forcibly flush the stream.
```

Example: `help()` Interactive Mode

Example

```
>>> 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> topics

Here is a list of available topics.  Enter any topic name to get more help.

ASSERTION           DELETION            LITERALS            SEQUENCES
ASSIGNMENT          DICTIONARIES        LOOPING             SHIFTING
ATTRIBUTEMETHODS    DICTIONARYLITERALS  MAPPINGMETHODS      SLICINGS
ATTRIBUTES          DYNAMICFEATURES     MAPPINGS
SPECIALATTRIBUTES
AUGMENTEDASSIGNMENT ELLIPSIS            METHODS
SPECIALIDENTIFIERS
BASICMETHODS        EXCEPTIONS          MODULES             SPECIALMETHODS
BINARY              EXECUTION           NAMESPACES          STRINGMETHODS
BITWISE             EXPRESSIONS         NONE                STRINGS
BOOLEAN             FILES               NUMBERMETHODS       SUBSCRIPTS
CALLABLEMETHODS     FLOAT               NUMBERS             TRACEBACKS
CALLS               FORMATTING          OBJECTS             TRUTHVALUE
CLASSES             FRAMEOBJECTS        OPERATORS           TUPLELITERALS
CODEOBJECTS         FRAMES              PACKAGES            TUPLES
COMPARISON          FUNCTIONS           POWER               TYPEOBJECTS
COMPLEX             IDENTIFIERS         PRECEDENCE          TYPES
CONDITIONAL         IMPORTING           PRIVATENAMES        UNARY
CONTEXTMANAGERS     INTEGER             RETURNING           UNICODE
CONVERSIONS         LISTLITERALS        SCOPING
DEBUGGING           LISTS               SEQUENCEMETHODS

help>
```

Module Summary

- **Python3 is a General Purpose, Object-Oriented Programming Language**
- **Python3 is not backwards compatible with previous versions of Python**
- **Multiple versions of Python may be installed on a single computer**
- **Python runs on a variety of Operating Systems including the predominant Unix, Windows and Mac Operating Systems**
  - Python ships as a utility on Unix, Linux and most Mac Operating Systems
- **Python has multiple execution modes**
  - Script Mode
  - Interactive Mode
- **Arguments may be passed to a Python Program at run time**
- **Use the `help()` function to interact with the Python help system**

Module 2

Writing Expressions in Python

Module Goals and Objectives

Major Goals

Understand Python Syntax

Create Variables in Python

Understand Objects in Python

Specific Objectives

On completion of this module, students will be able to:

Write Expressions in Python

Create Variables

Use Variables in Expressions

Make references to Objects

Use Objects as part of statements and expressions

Query an Object for extra information

Section 2–1

Python Basic Syntax

Python General Syntax

- **Python is a case sensitive language**
  - The identifiers `steve`, `Steve` and `STEVE` are all different
- **Python uses white space to delineate blocks of code**
  - White space is not arbitrary in Python like it is in most programming languages
- **Python supports single-line and multi-line comments**
- **Python supports multiple expressions on a single line**
  - Delimit multiple statements with a semi-colon `;`
  - This practice is not recommended as it detracts from the readability of python
- **Python ignores arbitrary lines containing only white space**
- **Line continuation is supported through the \ character**

Example: Multiple Expressions on a Single Line

Example

```
#!/usr/bin/Python3

import sys; msg ="Hello, World"; sys.stdout.write(msg)


#vs


import sys
msg = "Hello, World"
sys.stdout.write(msg)
```

Line structure

- **A Python program is divided into a number of logical lines**

- **The end of a logical line is represented by the token NEWLINE**

```
* \n
```

Example

```
print("This is line 1")
print("This is line 2")
```

- **Python supports line continuation through the line continuation character **

Example

```
msg = "Hello,"\
      "world"

print(msg)
```

- **Statements contained within `[]`, `{}`, or `()` brackets do not need to use the line continuation character**

Example

```
x = ['a','b','c','d','e','f',
    'g','h','i']
print(x)
```

Comments in Python

- **Python supports Single Line and Multi Line Comments**
  - Single Line comments are shell style comments and are prefaced with a #

Example

```
#This is a Comment
```

- **Multi Line Comments are created with triple quoted strings**
  - You may use double quotes or single quotes

Example

```
'''
This is
a Multi-Line
Comment
'''
```

- **Triple quoted strings may also be used to create docstrings**
  - Docstrings are Python Documentation Strings and will be discussed later in the course

The use of White Space in Python Programs to Create Statements

- **Unlike most programming languages, the use of White Space in Python is not arbitrary**
- **Python uses White Space to create blocks of code**
  - Most programming languages use braces to create blocks of code
- **Blocks of code in Python are known as "suites"**
  - The block of code attached to a function is the "Function Suite"
  - The block of code attached to a class is the "Class Suite", etc..
- **By default, Python uses 4 spaces of indentation to create a code block**
  - The second most popular method of indentation is to indent by using a tab
- **Technically, the amount of indentation in code is variable, as long as the amount of indentation is consistent within the suite**
- **Invoking `python.exe` with the `-t` option issues warnings about code that illegally mixes tabs and spaces**
  - When using `-tt` these warnings become errors

Statements

- **Python has 2 types of statements**
  - Simple statements
  - Compound statements
- **Simple statements fit on a single line**
  - Does not have an attached suite
- **Compound statements have a suite of code**
  - First line of a compound statement is called the header
  - The indented code block is called a suite
    - General rule is to indent by 4 spaces to make a suite

Pseudo-Syntax

```
Header Line:
    Nested Statement Block
```

Example: Use White Space in Python

Example

```
#!/usr/bin/Python3

msg = "Hello, World!"

def alert(p):
    """
    This function prints the
    value of a variable
    to stdout
    """
    print(p)

alert(msg)
help(alert)
```

Output

```
Hello, World!
Help on function alert in module __main__:

alert(p)
    This function prints the
    value of a variable
    to stdout
```

Example: White Space Error in Python Program

Example

```
#!/usr/bin/Python3

msg = "Hello, World!"

def alert(p):
    """
    This function prints the
    value of a variable
    to stdout
    """
    print(p)


alert(msg)
 help(alert)
```

Output

```
C:\HOTTPython\Mod03\Examples>python -tt whitespace_error.py
  File "whitespace_error.py", line 15
    help(alert)
    ^
IndentationError: unexpected indent
```

Section 2–2

Python Identifiers

Python's Lexical Categories of Tokens

- **The Python interpreter reads code as a sequence of characters and translates them into a sequence of tokens**
- **Each token is classified by its lexical category**
  - This process is known as "tokenization"
- **Python has 5 Lexical Categories of tokens**
  - Identifier
    - Names that a Programmer or Python defines
  - Operators
    - Symbols that operate on data and return results
  - Delimiters
    - Perform the operations of grouping, punctuation, assigning variables
  - Literals
    - Values classified by their types. (Numbers, Strings, Boolean)
  - Comments
    - Documentation for Programmers

Working with Python Identifiers

- **Python identifiers are names used to identify**
  - Variables
  - Functions
  - Classes
  - Modules
  - Objects
- **Identifiers are used by developers to refer to the objects they have created**
- **Identifiers must start with an upper or lower case letter or an _ followed by 0 or more letters, underscores or digits**
- **Identifiers are case sensitive**

Example

```
#Create a Variable
x = 0

#Create a Function
def functionname(param):
    function_suite

#Create a Class
class ClassName:
    class_suite
```

Identifier Naming Conventions

- **Python has very few limitations on coding style**
  - The obvious exception is the use of white space to indent code blocks
- **Python does have some preferred stylistic conventions for the naming of identifiers**
  - These conventions may be found in the Python Enhancement Proposal (PEP) 8
- **Variable names and Function names should be all lowercase with underscore separators between words to enhance readability**
- **Class names should capitalize each word**
- **Constants should be all capitalized letters with underscore separators between words**
- **Packages and Modules should have short, lowercase names with underscore separators if necessary**

Example

```
sys                              #Module
get_first_name()                 #Function
full_name = "Bobby Tables"       #Variable
drop = NewStudent(full_name)     #Class
```

The Use of Leading Underscores in Identifiers

- **In Python, Underscores are meaningful**

- **Use a leading underscore for a name to indicate a private identifier**
  - Typically used in a class
  - The leading underscore is a weak "internal use" indicator
  - For Example, `from X import *` does not import objects whose name starts with an underscore

Example

```
class Student:
    _count_student = 0
```

- **Single trailing underscores are used to avoid conflicts with Python Keywords**

Example

```
class Vehicle():
    _classid = 8675309
```

The Use of Leading Underscores in Identifiers *cont'd*

- **Use double leading underscores when naming private class attributes**
  - This style invokes Python's internal name mangling mechanism to create private variables
    - Python internally prefaces these type of variables internally with an underscore and the class name

Example

```
class Person():
    __id = 8675309

print( dir(Person))
['_Person__id', '__class__', '__delattr__', '__dict__',
...
, '__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
```

- **Names with both double leading *and* double trailing underscores are used by Python-generated "magic" objects or attributes that live in user-controlled name spaces**
  - Never create names with this format—only use them as documented

Example

```
print(__name__)
# __main__
```

Python Keywords

- **Keywords are identifiers in Python that have a predefined meaning**

- **Python has many internal identifiers that are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers**

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

- **Because Python is a dynamic language, this list may change over time**
  - When in doubt, consult the Python documentation
  - The keyword module has an `iskeyword()` method that may be used to test if a string is a keyword

Example: Identifying Keywords

Example

```
#!/usr/bin/Python3
import sys
import keyword

print("Python version: ", sys.version_info)
print("Python keywords: ", keyword.kwlist)

if keyword.iskeyword('elif'):
    print('elif is a keyword')
else:
    print('elif is not a keyword')
```

Output

```
Python version:  sys.version_info(major=3, minor=3,
micro=2, releaselevel='final', serial=0)

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

elif is a keyword
```

Section 2–3

Creating Variables in Python

Working with Variables

- **Variables in Python are created by placing a label on the left side of the assignment operator and an object on the right side of the assignment operator**

Example

```
x = 7
y = "HOTT Python"
```

- **Variables are created in Python when your code first assigns the variable a value**
- **Variables may reference any type of Object in Python**
- **Variables never have any type information associated with them**
  - Type information pertains to the objects the variables point to, not the variable names themselves
- **When variables appear in an expression, the variable is replaced with the object the variable currently refers to**

Variables and References

- **In Python, variables are really references to objects**
- **This means that variables are treated as labels that are associated with an object**
- **Variables are always linked to an object and never another variable**
- **Objects however may be linked to other Objects**

Variable Example

Example

```
#!/usr/bin/Python3

msg = "HOTT Python"
msg2 = msg
print("Variable msg's id is", id(msg))
print("Variable msg2's id is", id(msg2))
print("msg and msg2 point to the same object")


msg = "HOTT Python: Live at the Hollywood Bowl"
print("Variable msg's id is", id(msg))
print("Variable msg2's id is", id(msg2))
print("msg and msg2 point to different objects")

Output
Variable msg's id is 89197936
Variable msg2's id is 89197936
msg and msg2 point to the same object
Variable msg's id is 89281088
Variable msg2's id is 89197936
msg and msg2 point to different objects
```

Python's Dynamic Typing Model

- **Python is a Dynamically typed programming language**
- **There are no type declarations in Python**
  - In Python, data types for variables are determined automatically at run time
  - Python keeps track of types automatically instead of requiring declaration code
  - Programmers are only allowed to perform operations on an object that are valid for that object type
- **Variables are declared and initialized when they are needed, not ahead of time**
- **Declaring a variable in Python without defining a value, yields an error**
- **Python variables are always pointers to objects, not labels of changeable memory areas**
  - Pythonistas prefer to refer to variables as "*names*"
- **Setting a variable to a new value does not alter the original object, but rather causes the variable to reference an entirely different object**

Module 3

Working With Numbers

Module Goals and Objectives

Major Goals

Understand the Number object

Work with Numeric Functions

Specific Objectives

On completion of this module, students will be able to:

Describe and use the Numeric Object types

Perform conversion operations on numbers

Use numeric operators in expressions

Use the Decimal and Fraction modules

Understand Python's order of operations

Section 3–1

Number Object Overview

Overview of Numeric Data Types

- **The Python Number object is not a single object but a category of similar data types**

- **Python supports a standard list of Numeric Literals**
  - Integer Objects
  - Floating Point Objects
  - Boolean Objects
  - Complex Number Objects

- **Python also supports the creation of Numeric objects through a function oriented interface**
  - The Decimal object may be used by importing the `decimal` module
  - The Fraction object may be used by importing the `fractions` Module

Overview of Operators and Functions

- **Python supports a robust list of built-in numeric operators**
  - Standard Arithmetic Operators
  - Logical Operators
  - Comparison Operators
  - Augmented Assignment Operators
  - Bitwise Operators
- **Many of Python's operators are overloaded and change their operations given their context of usage**
  - For instance the Modulo operator % may be used to return the remainder of a division operation, or to format strings
  - The `*` operator may be used to make copies of sequences
- **Python also supports a robust list of built in numeric functions**
  - `abs(), round(), oct(), hex(),` etc...
- **Python's standard library has a number of modules to provide numeric and math-related functions and data types**
  - numbers, math, decimal, fractions, random, cmath

Uses for the Number object

- **The Number object is the basis of all of the numeric expressions you will write in your programs**

- **Numeric expressions are also the basis of Python's implicit processing and shorthand notations**
  - We will be discussing these concepts as the course progresses

- **Pythons Number object is robust and flexible enough that it is used, heavily, as the basis of Scientific and Engineering modules and applications**
  - NumPy adds a fast, compact, multidimensional array facility to Python
  - SciPy is an open source library of scientific tools for Python
  - OpenOpt is a framework for numerical optimization and systems of linear/non-linear equations
  - SpaceFuncs is a tool for 2D, 3D, N-dimensional geometric modeling with possibilities of parameterized calculations, numerical optimization and solving systems of geometrical equations with automatic differentiation
  - See: `http://wiki.python.org/moin/NumericAndScientific` for more information about Scientific, Numeric and Engineering modules in Python

Section 3–2

Operators and Expressions

Operator Precedence

- **Operator precedence refers to the order in which operators are processed when the order is not explicitly controlled with parentheses**

- **Python has a robust operator set, and consequently a robust order of operations**

Operator	Meaning
`(...), [...], {...}`	Tuple, list, set and dictionary creation
`s[i], s[i:j]`	Indexing and slicing
`s.attr`	Attributes
`f(...)`	Function calls
`+x, -x, ~x`	Unary operators
`x ** y`	Power (right associative)
`x * y, x / y, x // y, x % y`	Multiplication, division, floor division, modulo
`x + y, x - y`	Addition, subtraction
`x << y, x >> y`	Bit-shifting
`x & y`	Bitwise and
`x ^ y`	Bitwise exclusive or
`x | y`	Bitwise or
`x < y, x <= y,`	Comparison, identity, and sequence membership tests
`x > y, x >= y,`	
`x == y, x != y`	
`x is y, x is not y`	
`x in s, x not in s`	
`not x`	Logical negation
`x and y`	Logical and
`x or y`	Logical or
`lambda args: expr`	Anonymous function

Simple Arithmetic

- **Python has a number of basic arithmetic operators that work on all numeric types**

Operation	Description
`x + y`	Addition
`x - y`	Subtraction
`x * y`	Multiplication
`x / y`	Division
`x // y`	Truncating division (Floor division)
`x ** y`	Power (x^y)
`x % y`	Modulo (x mod y)
`-x`	Unary minus
`+x`	Unary plus

Example

```
>>> x = 3
>>> y = 3
>>> x + y
6
>>> x - y
0
>>> x * y
9
>>> x / y
1.0
>>> x // y
1
>>> x ** y
27
>>> -x
-3
```

The Division Conundrum

- **In Python3.x the `/` operator performs *true* division**
  - Returns a floating point result that includes any remainder, regardless of the operand type
- **In Python 2.x the `/` operator performs *classic* division**
  - If both operands are integers, then truncating integer division is performed
  - Otherwise, floating point division is performed, retaining any remainder
- **The `//` operator performs the same operation in both versions of Python**
  - Truncating division for integers
  - Floor division for floating point numbers

The Division Conundrum *cont'd*

- **To use division operations consistently in Python 2.x and 3.x**
  - If your program depends on truncating integer division then use the `//` operator in both Python 2.x and 3.x
  - If your program requires floating point results for integers then use the `float()` function to guarantee that one of the operands is a floating point number

Example

```
a = b // c          #Always truncates with an int result
a = b / float(c)    #Float division with a remainder
```

The // operator

- **The // operator is informally called truncating division**
- **The // operator actually performs floor division**
  - The operator truncates the result down to its floor
  - The closest whole number below the true result
- **The net effect of the result is to round down, not to strictly truncate**
  - This issue matters as it relates to negative values
  - Positive results have the effect of truncation
  - Negative results have the floor effect

Example

```
import math

math.floor(2.5)           #2
math.floor(-2.5)          #-3
math.trunc(2.5)           #2
math.trunc(-2.5)          #-2

#Python3.x true division
5/2 , 5 / -2              #(2.5, -2.5)

#Python2.x classic division
5/2 , 5 / -2              #(2.5, -3)

#Works the same in 2.x and 3.x
5 // 2 , 5 // -2          #(2, -3)
5 // 2.0 , 5 // -2.0      #(2.0, -3.0)
```

Examples: Division

- **Division using Python 3.x**

Example

```
>>> 10 / 4
2.5
>>> 10 / 4.0
2.5
>>> 10 // 4
2
>>> 10 // 4.0
2.0
```

- **Division using Python 2.x**

Example

```
>>> 10 / 4
2
>>> 10 / 4.0
2.5
>>> 10 // 4
2
>>> 10 // 4.0
2.0
>>>
```

Bitwise Operations

- **Bitwise operators compare two numbers bit by bit and set the corresponding bit in the result to 1 or 0**
  - Bitwise Operations may only be performed on Integers

Operation	Description
`x << y`	Left shift. The left operands value is moved to the left by the number of bits specified by the right operand
`x >> y`	Right shift. The left operands value is moved to the right by the number of bits specified by the right operand
`x & y`	Bitwise `and`. Copies a bit to the result if it exists in both operands
`x | y`	Bitwise `or`. Copies a bit if it exists in either operand
`x ^ y`	Bitwise `xor` (exclusive or). Copies the bit if the bit is set in one operand but not the other
`~x`	Bitwise negation. Negates the value of the bit

Example: Bitwise Operators

Example

```
C:\Users\HOTT>python
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43)
[MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> x = 16
>>> y = 4
>>> print(bin(x))
0b10000
>>> print(bin(y))
0b100
>>> print(bin(x & y))
0b0
>>> print(bin(x | y))
0b10100
>>> print(bin(x ^ y))
0b10100
>>> print(bin(x << 2))
0b1000000
>>> print(bin(x >> 2))
0b100
```

Comparison Operators

- **Python's Comparison Operators compare the values of objects and return a true or a false**
- **Comparison operations are supported by all objects**
- **Comparison Operations may be arbitrarily chained together**

Operation	Description
`x < y`	Returns true if x is less than y
`x > y`	Returns true if x is greater than y
`x == y`	Returns true if x is equal to y
`x != y`	Returns true of x is not equal to y
`x <> y`	Returns true of x is not equal to y (Python 2.x)
`x >= y`	Returns true if x is greater than or equal to y
`x <= y`	Returns true if x is less than or equal to y
`is`	Object identity
`is not`	Negated Object identity

Example: Comparison Operators

Example

```
C:\Users\HOTT>python
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43)
[MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> x = 2
>>> y = 3
>>> x == y
False
>>> x != y
True
>>> x < y
True
>>> x > y
False
>>> x <= y
True
>>> x >= y
False
>>> x is y
False
>>> x is not y
True
>>> x = y
>>> id(x)
505910896
>>> id(y)
505910896
>>> x is y
True
>>> x is not y
False
```

Chained Comparison Operations

- **Python supports the chaining of Comparison operators to perform range tests**

- **Chained comparisons are a type of shorthand notation for larger Boolean expressions**

Example

```
#!/usr/bin/Python3

x = 1
y = 3
z = 5
print("This statement x < y < z ")
print(x < y < z)
print("Is equivalent to this statement x < y and y < z")
print(x < y and y < z)
```

Output

```
This statement x < y < z
True
Is equivalent to this statement x < y and y < z
True
```

Boolean Operators

- **Boolean operators compare 2 expressions and return a Boolean true or false depending on the result of the comparison**

Operator	Description
`x or y`	Return true if x or y is true; otherwise return false
`x and y`	Return true if x and y are true; otherwise return false
`not x`	If x is false, return 1; otherwise, return 0

- **The `or` operator is a short-circuit operator**
  - `or` only evaluates the second argument if the first argument is False

- **The `and` operator is a short-circuit operator**
  - `and` only evaluates the second argument if the first argument is True

Example

```
>>> x = 1
>>> y = 3
>>> x == 1 and y == 3
True
>>> x == 2 or y  == 3
True
>>> not x
False
```

Assignment and Augmented Assignment Operators

- **The assignment operator = creates object references**

Example

```
x = 7
```

- **Python also provides a set of augmented assignment operators that may be used anywhere an ordinary assignment operation is used**

Operation	Description
`x += y`	`x= x + y`
`x -= y`	`x= x - y`
`x *= y`	`x= x * y`
`x /= y`	`x= x / y`
`x //= y`	`x = x // y`
`x **= y`	`x= x ** y`
`x %= y`	`x= x % y`
`x &= y`	`x= x & y`
`x |= y`	`x= x | y`
`x ^= y`	`x= x ^ y`
`x >>= y`	`x= x >> y`
`x <<= y`	`x= x << y`

Example

```
>>> x = 1
>>> x+=x
>>> x
2
```

The Attribute and Function Call Operators

- **The dot (.) operator is used to access the attributes of an object**

- **More than one (.) operator can appear in an expression**

Example

```
import sys
import inputExample
print(inputExample.color)
print("The Current Operating System is", sys.platform)
```

- **The function call operator `()` is used to make a call on a function**

- **Each argument to a function is an expression**
  - All arguments' expressions are fully evaluated from left to right
    - Known as *applicative order evaluation*

Example

```
#!/usr/bin/Python3
def spam(msg):
    print((msg + " ") * 3)

spam("Spam")
```

Conditional Expressions

- **Python has a conditional expression equivalent to the C ternary operator `?:`**

Syntax

```
x if [expression] else y
```

- **The `[expression]` is first evaluated**
  - If `[expression]` is true then `x` is evaluated and its value is returned
  - Otherwise `y` is evaluated and its value is returned

Example

```
#!/usr/bin/Python3
a = 1
b = 2
min = a if a <= b else b
print(min)                  #prints 1

#This if statement is functionally equivalent to the
#Conditional Expression
if a <=b:
    min = a
else:
    min = b
print(min)                  #prints 1
```

Section 3–3

Numeric Subtypes

Numeric Subtype Overview

- **Python supports 4 built-in numeric subtype objects that may be created using a literal notation**
  - Integers
  - Floating Point Numbers
  - Booleans
  - Complex Numbers
- **Python has standard library modules which extend the built-in numeric list with 2 more numeric subtype objects**
  - The `fractions` module provides support for rational number arithmetic
  - The `decimal` module provides support for fast correctly-rounded decimal floating point arithmetic

Literal	Interpretation
`1245, 0 , -1245, 86753098675309`	Integers (Unlimited Size)
`1.23 , 8. , 8.67e-10, 8.67e+100`	Floating Point Numbers
`True, False, bool(x)`	Boolean Numbers
`8+6j , 7.5 + 3.0j , 9J`	Complex Numbers
`Decimal('8.67'),Fraction(1,3)`	Decimal and Fraction Extensions
`0o123 , 0x9ab , 0b00111`	Python3.x Octal, Hex and Binary literals
`0123,0o123, 0x9ab , 0b00111`	Python2.x Octal, Hex and Binary literals

Integers and Floating Point numbers

- **Python 3.x integers are signed integers of unlimited length**
- **Python 2.x supports 2 types of signed integers**
  - normal integers (typically 32 bit)
  - long integers of unlimited length
    * An integer may end in an `l` or an `L` to force it to become a long integer
    * Integers are automatically converted to long integers when their values overflow their allocated bits
- **Floating point numbers have a decimal point**
  - May also be written as an optional signed exponent introduced by an `e` or a `E`
  - Floating point numbers are often implemented as C doubles in Cpython

Literal	Interpretation
`1245, 0 , -1245, 86753098675309`	Integers (Unlimited Size)
`1.23 , 8. , 8.67e-10, 8.67e+100`	Floating Point Numbers

Example

```
>>> 1 + 7.675309
8.675309
```

Octal, Hex, and Binary Literals

- **Integers may be coded in**
  - Decimal (base 10)
  - Hex (base 16)
  - Octal (base 8)
  - Binary (base 2)
- **All literal coding styles produce integers, just using an alternative syntax style to do so**
- **Hexadecimal literals start with a leading `0x` or `0X`**
  - Followed by hexadecimal digits `[a-f0-9]`
- **Octal literals start with a leading `0o` or `0O` (a zero followed by an upper or lower case letter `O`)**
  - Followed by the digits [0-7]
  - Octal literals in Python 2.X may be coded with a leading `0`
- **Binary literals start with a leading `0b` or `0B` followed by `1`'s or `0`'s**
- **Python also has a number of conversion functions that may be used to convert numbers between their bases**

Boolean Numbers

- **Python has an explicit Boolean type called `bool`**
- **The `bool` type has 2 pre-assigned, built-in names**
  - True
  - False
- **True and False are instances of `bool`**
  - `bool` is a subclass of the built in integer type
  - True and False behave exactly like integer 1 and integer 0
    - Except True and False have custom printing logic so they represent themselves as True and False instead of 1 and 0 when being printed
- **Most developers treat True and False as if they are integers**

Example

```
>>> x = True
>>> type(x)
<class 'bool'>
>>> isinstance(x, int)
True
>>> print(x)
True
x == 1
True
>>> x is 1
False
>>> x + 1    #(interesting)
2
```

Complex Numbers

- **Mathematically, a complex number is a number of the form `A+Bj` where `j` is the imaginary number, equal to the square root of `-1`**

- **Complex numbers are quite commonly used in electrical engineering**

- **Complex numbers are a composite quantity made of two parts: the real part and the imaginary part, both of which are represented internally as float values**

- **You can retrieve the two components using attribute references**
  - complexnum.real
  - complexnum.imag

Example

```
>>> x = 3+4j
>>> x.real
3.0
>>> x.imag
4.0
```

- **The standard library `cmath` module provides access to mathematical functions for complex numbers**

The `complex()` function

- **Complex numbers may also be created by using the `complex()` function**

Syntax

```
complex([real [,imag]])
```

- **The `imag` argument defaults to `0`**

- **If both arguments are omitted then `0j` is returned**

Example

```
>>> x = complex(3,4)
>>> y = complex(4,5)
>>> x.real, y.real
(3.0, 4.0)
>>> x.imag, y.imag
(4.0, 5.0)
>>> x + y
(7+9j)
```

Decimal Type

- **The `Decimal` object was introduced with Python 2.4**
- **`Decimals` are created by calling a function within the imported `decimal` module**
  - There is no literal syntax for creating a decimal
- **Functionally, `Decimals` are like floating point numbers**
  - Except they have a fixed number of decimal points
  - Decimals are fixed-precision floating point numbers
- **Decimal numbers can be represented exactly**
  - In contrast, numbers like 1.1 and 2.2 do not have exact representations in binary floating point

Example

```
>>> 1.1
1.1
>>> 2.2
2.2
>>> 1.1+2.2
3.3000000000000003    # (For serious, float?)
```

- **The exactness of the Decimal object carries over to expressions as well**

Using the decimal Module

- **The first step to using the `decimal` module is to import the decimal module itself**

Syntax

```
import decimal
from decimal import *
```

- **Decimal objects may be created with the Decimal Constructor**

- **Decimal instances can be constructed from integers, strings, floats, or tuples**

Syntax

```
decimal.Decimal(int|float|string|(tuple))

Decimal(int|float|string|(tuple))
# if you imported the entire decimal namespace
```

- **The Decimal constructor will return a Decimal object**

Example

```
from decimal import *
x = Decimal('1.1')
y = Decimal('2.2')
z = x + y
print(z)

# prints 3.3
```

decimal.getcontext()

- **Contexts are environments for arithmetic operations**
  - They govern precision, set rules for rounding, determine which signals are treated as exceptions, and limit the range for exponents

- **The decimal `getcontext()` function allows you to see the decimal objects current context or set new values for the current context**

Syntax

```
decimal.getcontext()
decimal.getcontext().attribute = value
```

- **The context `prec` attribute is an integer in the range [`1`, `MAX_PREC`] that sets the precision for arithmetic operations in the context**
  - `MAX_PREC` is 999999999999999999 on 64-bit machines
  - `MAX_PREC` is 425000000 on 32-bit machines

Examples: `decimal.getcontext()`

Example

```
from decimal import *
print(decimal.getcontext())
#output
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999,
Emax=999999, capitals=1, clamp=0, flags=[],
traps=[DivisionByZero, Overflow, InvalidOperation])
```

Example

```
from decimal import *
getcontext().prec=3
x = Decimal('1.1243')
y = Decimal('2.2324')
z = x + y
print(z)
# prints 3.36

print(getcontext())
# prints Context(prec=3, rounding=ROUND_HALF_EVEN, Emin=-
999999, Emax=999999, capitals=1, clamp=0, flags=[Inexact,
Rounded], traps=[DivisionByZero, Overflow,
InvalidOperation])
```

The Fraction type

- **Python 2.6 and 3.0 introduced a new numeric type, Fraction**
- **Fraction implements a rational number object**
  - Fraction maintains an explicit numerator and an explicit denominator
  - This helps to avoid some of the inaccuracies and limitations of floating point math
- **Fraction is a cousin of the Decimal object**
- **Fraction, like Decimal, provides methods to get exact results to floating point mathematical expressions**
  - Fraction is a slower object than Float however
  - The code to work with Fraction is more verbose than float because there is no literal constructor for Fraction
- **To create a Fraction**
  - Import the `fractions` module
  - Pass a numerator argument and a denominator argument to the Fraction constructor

Working with Fractions

- **Import the `fractions` module**

syntax

```
import fractions
from fractions import Fraction
```

- **Pass a numerator and a denominator to the Fraction constructor**

- **Fractions may also be created by passing the constructor a**
  - Decimal
  - Float
  - String
  - Another Fraction

syntax

```
Fraction(n,d)
fractions.Fraction(n,d)
Fraction(decimal|float|'string'|fraction)
fractions.Fraction(decimal|float|'string'|fraction)
```

- **The Fraction constructor will return a Fraction object**

Examples: Fraction Object

Example

```
#!/usr/bin/Python3
from decimal import *
from fractions import Fraction

print(0.1 + 0.1 + 0.1 - 0.3) # 5.551115123125783e-17
print(Fraction(1,10) +
      Fraction(1,10) +
      Fraction(1,10) -
      Fraction(3,10) ) # prints 0
# From IDLE
# >>> Fraction(1,10) + Fraction(1,10) + Fraction(1,10) -
# Fraction(3,10)
# Fraction(0, 1)

print(Decimal('0.1') +
      Decimal('0.1') +
      Decimal('0.1') -
      Decimal('0.3')) # prints 0.0

# From IDLE
#>>> from decimal import *
#>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1') -
#Decimal('0.3')
#Decimal('0.0')
```

- **Often times, Fractions and Decimals can return more intuitive and/or more accurate results than floating point objects**
  - This is usually achieved through using rational representation of numbers and limiting precision

Section 3–4

Numeric Conversion Functions

Numeric Conversion Function Overview

- **It may be necessary to perform conversions between built-in data types**
- **Python makes it relatively easy to convert between built-in data types**
- **To convert between types, use the type name as a function to perform the data type conversion**
- **Python also supplies built-in functions to perform specialized conversions**
- **All of the conversion functions discussed in this section return a new object representing the converted value**

The `int()` function

- **The `int()` function converts a number or a string to a plain integer**

Syntax

```
int(number | string [base])
```

- **Conversions of floating point numbers to integers truncates towards 0**

- **The base argument may be passed only of the first argument is a string**
  - If base is passed 0 then the base used for conversion is determined by the string's contents
  - Otherwise, the value passed for base is used for the base conversion of the string
    * Values for base may be 0, 2..36
    * 2, 8, 10 , and 16 are common bases for conversion
  - The string may be preceded by a sign and surrounded by white space

- **If no arguments are passed to `int()`, `int()` returns 0**

The `float()` and `bool()` functions

- **The `float()` function converts a number or a string to a floating point number**

- **If no arguments are passed then 0.0 is returned**

Syntax

```
float([x])
```

- **The `bool()` function converts a value to a `boolean` object**
  - If an argument is false or omitted, the `bool()` function returns false
  - Otherwise the `bool()` function returns true

- **`bool()` is a subclass of int**
  - The only instances of `bool` are True and False

Syntax

```
bool([x])
```

The `round()` Function

- **Python's built-in `round()` function returns a floating point number rounded to a specified number of digits after the decimal point**

Syntax

```
round(number[,n])
```

- **The `round()` function accepts up to 2 arguments**
  - The number to round (`number`)
  - The number of decimal places (`n`)
    - The default number of decimal places to round to is `0`

Example

```
#!/usr/bin/Python3

j = 867.5309

print(round(j)) #868
print(round(j,1)) #867.5
print(round(j,2)) #867.53
```

- **For further information about Python's handling of floating point values, consult the "Floating Point Arithmetic: Issues and Limitations" at `docs.python.org`**

The `complex()` function

- **The `complex()` function builds and returns a complex number object**

Syntax

```
complex([real[,imag]])
```

- **The `complex()` function takes up to 2 arguments**
  - real
  - imag
- **If `imag` is omitted then the argument defaults to 0**
- **If both arguments are omitted then `complex()` returns `0j`**
- **In lieu of using the `complex()` function to construct complex numbers, the complex number literal syntax may be used**
  - real+imagj
    - `1+2j`
  - real+imagJ
    - `1+2J`

The `bin()`, `hex()` and `oct()` functions

- **The `bin()` function converts an integer number to a binary (base 2) string**
  - The result is a valid Python expression

Syntax

```
bin([x])
```

- **The `hex()` function converts an integer number to a hexadecimal (base 16) string**

Syntax

```
hex([x])
```

- **The `oct()` function converts a number to an octal (base 8) string**

Syntax

```
oct([x])
```

The `str()` and `repr()` functions

- **The `repr()` and `str()` functions are designed to return a string representation of an object**

- **The goal of the `repr()` function is to create an unambiguous string**

- **`repr()` produces, if at all possible, a string that looks like a valid Python expression that could be used to recreate an object with the same value**
  - The results of `repr()` should be able to be used in an `eval()` function to parse the results as valid Python code

Syntax

```
repr(object)
```

- **The goal of the `str()` function is to compute an "informal" string representation of an object**

- **The goal of the `str()` function is to create a readable string**
  - This concept differs from the `repr()` function because `str()` does not have to produce valid Python code

Syntax

```
str(object)
```

Example: `repr()` and `str()`

Example

```
#!/usr/bin/Python3

import datetime
today = datetime.datetime.now()
print(str(today))
print(repr(today))
x = repr(today)
print(eval(x))

# Output
2013-07-27 22:41:52.970360
datetime.datetime(2013, 7, 27, 22, 41, 52, 970360)
2013-07-27 22:41:52.970360
```

Module Summary

- **Python has built-in number objects that may be constructed with literal syntax**
  - Integers
  - Floating Point
  - Boolean
  - Complex
- **There are some differences between the way Python 2.x and Python3.x store integers**
- **Division is performed differently in Python 2.x and 3.x**
  - Classic Division vs. True Division
- **Python has 2 powerful modules to extend the built-in number objects**
  - `decimal`
  - `fractions`
- **Python supports a wide array of numeric conversion functions**
- **Python's `math` and `cmath` modules extend Python's built-in operators set**

Module 4

Working with Strings

Module Goals and Objectives

Major Goals

Work with string literals

Use functions and methods to manipulate strings

Specific Objectives

Create string literals using all quoting styles

Understand sequences

Use sequence operations to process string data

Format strings using the format method and string formatting expressions

Use functions to manipulate strings

Use string methods to interact with strings

Section 4–1

String Object Overview

The String Object

- **In Python, a string is an ordered collection of characters**
- **Strings are also immutable objects**
- **Strings in Python 3.X are Unicode**
  - The `str` object is used to handle Unicode text
  - The `bytes` object is immutable and handles binary data
  - The `bytearray` object is a mutable version of the bytes object
- **In Python 2.X:**
  - The `str` object handles 8-bit text and 8-bit binary data
  - The `unicode` object represents Unicode text
- **While this module focuses on the `str` object, awareness of the `bytes` object and the `bytearray` object will be useful when we discuss the processing of Files**
  - Files may be processed as Unicode text with the `str` objects
  - Files may also be processed in binary mode with the `bytes` or `bytearray` objects
- **Strings in Python are used to handle every kind of data, from, Standard Input from the user, to processing the contents of files on disk, to storing the raw bytes used for network data transfers and media files**

String Tools

- **Python supports a large collection of string-based tools**
- **Python has a powerful literal syntax for creating strings and parsing string sequences**
- **Python has a robust operator set for processing strings**
- **Python provides a set of string object methods that implement common string processing tasks**
- **Python ships with modules as part of its standard library that perform advanced processing tasks such as**
  - Building and parsing XML data
  - Performing pattern matching with Regular Expressions
- **Because strings are also sequences, sequence operators and functions that process sequences may be used on strings**

Section 4–2

String Literals

Creating Strings

- **There are a number of ways to create strings in Python**
- **One of the most used method of creating strings is to use the quote operators**
- **Strings may be created in Python by binding any character sequence with quote delimiters**
  - Single Quotes `' '`
  - Double Quotes `" "`
  - Triple Quotes `""" """` or `''' '''`
    - Also known as Block Strings
    - Also the basis of Docstrings
- **Single and Double quoted strings perform the same function in Python**
  - You simply need to make sure you bind your strings with a matched set of operators
- **Block Strings have a bit of syntactic sugar added to them to give them a different behavior than single or double quoted strings**

Single and Double Quoted Strings

- **Either single (') or double (") quotes may be used as string delimiters**
  - There is no functional difference between either type of string
  - Most Python developers prefer single quoted strings because they are easier to read

Syntax

```
' character sequence '
" character sequence "
```

Example

```
#!/usr/bin/Python3
m1 = 'And Now for Something Completely Different'
m2 = 'Monty Python and the Holy Grail'
m3 = "Monty Python's Life of Brian"
m4 = 'Monty Python Live at the Hollywood Bowl'
m5 = "Monty Python's The Meaning of Life"
print("Monty Python's first movie was " + m1)
```

- **Besides readability, another motivating factor in choosing which type of string delimiters to use is the ability to embed a quote character of the other variety in a string without having to escape the character**

Example

```
m3 = "Monty Python's Life of Brian"    #Versus
m3 = 'Monty Python\'s Life of Brian'
```

Escape Sequences

- **Escape Sequences allow characters that aren't easily typed on a keyboard to be embedded in a string**
- **The \ character followed by one or more characters in the string literal are replaced by a single character**
- **The \ character may also be used to disable any special meaning of a character in a string literal**
  - This is useful if you need to embed quotes in your string literal that are the same type of quote that you are using as a delimiter
- **If the \ operator is used before a character with no special meaning, then the \ and that character are maintained as part of the string literal**

Table of Escape Sequences

Escape Sequence	Meaning
\	Newline Ignored (continuation line)
\\	Backslash (stores one \)
\'	Single quote (stores ')
\"	Double quote (stores ")
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\xhh	Character with hex value hh(at most 2 digits)
\000	Character with octal value 000(up to 3 digits)
\0	Null: binary 0 character (doesn't end string)
\N{id}	Unicode database ID
\uhhh	Unicode character with 16-bit hex value
\Uhhhhhhhh	Unicode character with 32-bit hex value

Example

```
m2 = 'Monty Python and the Holy Grail'
print('\055' * 30) # Some Dashes
print('\x0A' + m2 + '\x0A') # Some new lines
print('\055' * 30) # Some Dashes

#Output
------------------------------

Monty Python and the Holy Grail

------------------------------
```

Raw Strings

- **Raw strings disable the behavior of the backslash character \ in a string literal**
- **In a raw string, a character following a backslash is included in the string without change, and all backslashes are left in the string**
- **This is a very useful behavior when you have to include a string literal sequence that uses a lot of back slashes**
  - Windows File Paths
  - Regular expressions
- **To create a raw string, simply append the raw prefix to a string**
  - The raw prefix is a lowercase `r` or an uppercase `R`

Syntax

```
r'Character Sequence'
R'Character Sequence'
```

Example

```
path = r'c:\newFlyingCircus\reallyfunnyskit.omg'
print(path) # c:\newFlyingCircus\reallyfunnyskit.omg
path = 'c:\newFlyingCircus\reallyfunnyskit.omg'
print(path) #Output
c:
ewFlyingCircus
eallyfunnyskit.omg
```

Block Strings

- **Block Strings are strings that are delimited by triple quotes (' ' ') or (""")**
- **Block strings are used to create multi line strings**
- **They are useful any time you need multiline text in your program**
  - XML
  - JSON
  - HTML
- **Essentially, the block string honors the white space placed in the string literal**
  - New lines are converted to \n
  - Extra spaces are honored
- **Single and Double quotes may be used without escaping them**
- **Escape sequences are honored in the block string**
- **Triple quoted strings are also used for documentation strings (docstrings) , which we will be discussed later**
- **Block Strings may also be used as multi-line comments**

Example: Block String

example

```
#!/usr/bin/Python3
ni = '''
Head Knight: The Knights Who Say Ni demand a sacrifice!
King Arthur: Knights of Ni, we are but simple travelers who
seek the enchanter who lives beyond these woods--
Knights who say Ni: NI! NI! NI! NI!
King Arthur: Oh, ow!
Head Knight: We shall say "Ni" again to you, if you do not
appease us.
King Arthur: Well, what do you want?
Head Knight: We want a shrubbery!! [jarring chord]
'''

'''
I think I want to print here
Or maybe some other fancy Python
stuff like repeating Ni!
'''
print(ni)
```

Output

```
Head Knight: The Knights Who Say Ni demand a sacrifice!
King Arthur: Knights of Ni, we are but simple travelers who
seek the enchanter who lives beyond these woods--
Knights who say Ni: NI! NI! NI! NI!
King Arthur: Oh, ow!
Head Knight: We shall say "Ni" again to you, if you do not
appease us.
King Arthur: Well, what do you want?
Head Knight: We want a shrubbery!! [jarring chord]
```

Section 4–3

String Operations

String Concatenation and Repetition

- **Concatenation is the process of adding 2 strings together end to end**
  - Python supports a number of ways to concatenate strings together
- **Strings may be concatenated together by using the concatenation operator +**
- **Adjacent space-separated strings will be concatenated together**
- **The line continuation character may be used to separate a string across multiple lines**

Syntax

```
a = 'string' + 'string'
b = 'string' 'string'
c = 'string'\
    'string'
```

- **The repetition operator * is used to repeat a string**
  - The left operand is a string and the right operand is an integer value representing the number of times to repeat the string

Syntax

```
string * int
```

Example: Concatenation and Repetition

Example

```
#!/usr/bin/Python3
a = 'Hello' + 'World'
b = 'Hello' 'World'
c = 'Hello'\
    'World'
x = '*' * 50
print(x)
print(a)
print(x)
print(b)
print(x)
print(c)
print(x)
print('Knights' 'Who' 'Say' 'Ni')
print(x)
```

```
**************************************************
HelloWorld
**************************************************
HelloWorld
**************************************************
HelloWorld
**************************************************
KnightsWhoSayNi
**************************************************
```

The `len()` function

- **The `len()` function is used to return the number of items in a collection object**
- **A collection object may be a sequence or a mapping**
- **Since strings are sequences, the `len()` function may be used to report the number of characters in a string**

Syntax

```
len(object)
```

Example

```
#!/usr/bin/Python3
p = input("What is the password")
lp = len(p)
if lp < 8 or lp >15:
    print("Password is not between 8 and 15 characters")
```

String Indexing and String Slicing

- **Because strings are ordered collections of characters (sequences , components of the string may be accessed by position**

- **Slicing is a form of indexing that returns an entire section of a sequence**
  - A number of languages use the term "substring" to mean "slice" when talking about strings

- **Slicing is a behavior of sequences, so the discussion we are having here about strings will also carry over to other sequence objects such as lists and tuples**

- **Indexing and slicing is performed by attaching a set of square brackets to a sequence**

Syntax

```
#Indexing Syntax
seq[i]

#Slicing Syntax
seq[i:j]

#Extended Slicing Syntax
seq[i:j:stride]
```

Indexing Rules

- **The syntax `seq[i]` returns the sequence component found at the offset i**

- **Sequences have `0`-based offsets for counting**
  - `seq[0]`

- **The last item in the sequence is 1 item less than the length of the sequence**
  - `len(seq) - 1`
  - `seq[len(seq) - 1]`

- **Python supports negative indexes which count backwards from the end of the sequence**
  - `seq[-1]`

Example

```
#!/usr/bin/Python3
m = "Monty Python's The Meaning of Life"
print('First character in m is ',  m[0])
print('Last character in m is ',  m[len(m) - 1])
print('Last character in m is ',  m[-1])
print('Last 4 characters in m are ',  m[-4:])
```

Slicing Strings

- **Slicing syntax is one of the most convenient and effective methods of returning substrings from string literals**
  - Slicing may be used to return entire sections of strings in a single step
    - May also be used to extract columns of data, trim leading or trailing strings, copy or reverse strings
- **To work with slices, index a sequence with a pair of colon (:) separated offsets**
  - Python will return everything from the first offset , up to but not including the second offset
  - The left offset is used as the inclusive lower bound of the slice
  - The right offset is used as the non inclusive upper bound

Syntax

```
seq[i : j]
```

Example

```
#!/usr/bin/Python3
m = "Monty Python's The Meaning of Life"
print(m[0:5])
#Output
Monty
```

Slice Operations

- **`seq[i : j]`**
  - Fetches everything between `i` and `j`, non-inclusive of `j`
- **`seq[i :]`**
  - Omitting the second offset fetches everything from the first offset to the end
- **`seq[: j]`**
  - Omitting the first offset returns everything from the beginning of the sequence to the second offset, non-inclusive of that offset
- **`seq[:-j]`**
  - Omitting the first offset and using a negative second offset returns everything from the front of the sequence to the position of the negative offset, non inclusive of that offset, counting from the end of the sequence
- **`seq[-i:]`**
  - Using a negative first offset and omitting the second offset returns everything from the first offset, (counting from the end of the sequence) to the end of the sequence
- **`seq[:]`**
  - Fetches everything. Makes a top-level copy

Example: Slicing Operations

Example

```
#!/usr/bin/Python3
m = "Monty Python's The Meaning of Life"
n = m[:]
print(m[0:5])    # Monty
print(m[15:])    # The Meaning of Life
print(m[:14])    # Monty Python's
print(m[:-7])    # Monty Python's The Meaning
print(m[-4:])    # Life
print(n)         # Monty Python's The Meaning of Life
```

Example

```
#!/usr/bin/Python3
import sys

x = sys.argv
print("The user called the script " ,
    x[0],
    "with the following arguments",
    x[1:])
```

Output

```
C:\HOTTPython\Mod05\Examples>python sysargs.py John Graham
Terry Eric Terry Michael

The user called the script sysargs.py with the following
arguments ['John', 'Graham', 'Terry', 'Eric', 'Terry',
'Michael']
```

Extended Slicing

- **Python's slicing syntax supports a 3rd offset known as the stride**

- **The stride is used as a step argument**
  - By default Python strides through your sequence, one item at a time from left to right
    * The default stride is +1
  - Stride supports positive and negative integer values
    * Negative strides count backwards , right to left
    * Negative strides also reverse the order of the offsets

Syntax

```
seq[i:j:k]
```

- **The extended slice sequence should extract items in sequence from `i` through `j - 1` by `k`**

Example

```
#!/usr/bin/Python3
x = '123456789'
print(x[::2])   #13579
print(x[1::2])  #2468
print(x[::-1])  #987654321
#Returns a reversed version of the string. WHAT!
print(x[9:4:-1])    #9876
```

String Conversion Tools

- **A number of string conversion tools have already been discussed in the section "Numeric Conversion Functions" via the `Numbers` module**

- **Character conversions may be performed by using functions `chr()` and `ord()`**

- **The `chr(n)` function returns a one character string whose Unicode code point is the integer supplied as the argument `n`**
  - This function is the inverse of the `ord('c')` function

- **The `ord('c')` function returns an integer code point value of the one character argument `c`**
  - For 7-bit ASCII code for ASCII characters
  - For Unicode, this is the Unicode code point of a one-character Unicode string

Example

```
#!/usr/bin/Python3
print(ord('M')) #77
print(chr(77))  #M
```

Testing Strings with the `in` and `not in` Operators

- **The `in` operator and the `not in` operator may be used to test the membership of a substring within a string**

- **The in and not in operators are sequence operators**
  - Membership may be tested for any of the sequence types

Syntax

```
sub in string
```

- **Returns True if sub is found in string; otherwise returns False**

Syntax

```
sub not in string
```

- **Returns True if sub is not found in string; otherwise returns True**

Example

```
#!/usr/bin/Python3
j = 'John Cleese'
print('J' in j)          #True
print('j' in j)          #False
print('SPAM' not in j)   #True
```

Section 4–4

Formatting Strings

Formatting Strings

- **Python has a number of string methods and sequencing operations that allow for some advanced processing of strings**
- **Python also supports 2 very robust methods for performing data type specific formatting and substitutions in a single step**
- **String Formatting Expressions**
- **String `format()` Method**
- **String formatting expressions have been available since Python's creation and are based on the C programming language's "printf" model**
  - String formatting expressions are used heavily in Python programs
- **The string format method was introduced with Python 2.6/3.0 and is based off of the C#/.NET format method**
  - The format method shares some functionality with string formatting expressions

String Formatting Expressions

- **Python uses the % operator to perform string formatting**
  - The % operator is an overloaded operator
    - Given numbers, the % operator returns the remainder of a division operation
    - Given a string and a tuple, % will replace the template items in the string with the objects in the tuple

Syntax

```
'string' % (object [,object])*
```

- **The string on the left of the % operator should be a format string containing one or more conversion targets**
  - Conversion targets are string formats that start with a `$`

- **The right hand argument should be a tuple of objects to insert into the format string on the left hand side**
  - Each ordinal position in the tuple is embedded in the conversion targets in the string

Example

```
print('%s , %s , %s !' % ('spam','spam','spam'))
# spam , spam , spam !
```

Conversion Target Syntax

- **Conversion targets support a robust syntax that allows you to create some very flexible and adaptable format strings**

- **The general syntax for a conversion target is**

Syntax

```
%[(keyname)][flags][width][.precision][length]type
```

  - Keyname should be used to index a dictionary that is supplying values to the format string

  - Conversion flags which affect the result of some conversion types

Flag	Meaning
`'#'`	The value conversion will use the "alternate form"
`'0'`	The conversion will be zero padded for numeric values
`'-'`	The converted value is left adjusted
`' '`	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion
`'+'`	A sign character ('+' or '-') will precede the conversion (overrides a "space" flag)

  - Minimum field width (optional)

  - Precision (optional), given as a '.' (dot) followed by the precision

  - Length modifier (optional)

  - Conversion type

Conversion Type List

Conversion	Meaning
`'d'`	Signed integer decimal
`'i'`	Signed integer decimal
`'o'`	Signed octal value
`'u'`	Obsolete type – it is identical to 'd'
`'x'`	Signed hexadecimal (lowercase)
`'X'`	Signed hexadecimal (uppercase)
`'e'`	Floating point exponential format (lowercase)
`'E'`	Floating point exponential format (uppercase)
`'f'`	Floating point decimal format
`'F'`	Floating point decimal format
`'g'`	Floating point format. Uses lowercase exponential format if exponent is less than -4 or exceeds the format template's precision, decimal format otherwise
`'G'`	Floating point format. Uses uppercase exponential format if exponent is less than -4 or exceeds the format template's precision, decimal format otherwise
`'c'`	Single character (accepts integer or single character string)
`'r'`	String (converts any Python object using `repr()`)
`'s'`	String (converts any Python object using `str()`)
`'%'`	No argument is converted, results in a '%' character in the result

Example: String Formatting Expressions

Example

```
#!/usr/bin/Python3
msg = 'Bright Side of Life'
print('Take a walk on the %s !' % msg)

i = 8675309
j = 'Integers:===%d===%-10d===%010d' % (i,i,i)
print(j)

f = 8.675309
j = 'Floats: %e | %f | %g' % (f,f,f)
print(j)

msg = """
Hey %s!
It appears that today
you are %d, years old!
"""

print(msg % ('Jenny',8675309))

#Output
Take a walk on the Bright Side of Life !
Integers:===8675309===8675309   ===0008675309
Floats: 8.675309e+00 | 8.675309 | 8.67531
Hey Jenny!
It appears that today
you are 8675309, years old!
```

The `format()` Method

- **The `format()` method is an alternative to string formatting expressions**
  - The `format()` method is a replacement for the string formatting expressions
- **The `format()` method takes a subject string as a template and takes 0 or more arguments that represent values to be substituted according to the template**

Syntax

```
template.format(*args, **kwargs)
```

- **The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`**
- **Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument**
- **Returns a copy of the string where each replacement field is replaced with the string value of the corresponding argument**

Example: `format()` Method

Example

```
#!/usr/bin/Python3
msg = 'Bright Side of Life'
temp = 'Take a walk on the {0} !'
fmsg=temp.format(msg)
print(fmsg)

msg = 'Bright Side of Life'
temp = 'Take a walk on the {} !'
fmsg=temp.format(msg)
print(fmsg)

msg = 'Bright Side of Life'
temp = 'Take a walk on the {m} !'
fmsg=temp.format(m = msg)
print(fmsg)

#Output
Take a walk on the Bright Side of Life !
Take a walk on the Bright Side of Life !
Take a walk on the Bright Side of Life !
```

Section 4–5

String Methods

String Method Introduction

- **Python supplies a number of robust expressions and functions that may be used to process strings**
  - However, these are generalized operations, not custom tailored to the string object
- **Strings provide a set of built in methods that implement more sophisticated processing tasks**
- **These tasks include**
  - Parsing Text
  - Formatting strings
  - Testing strings
  - Changing strings

String Method Syntax

- **Methods are functions that are associated with particular objects**

- **Methods are technically "attributes" of an object**
  - Most methods in Python combine attribute fetches and method calls

- **Attribute fetches retrieve the value of an attribute**
  - The attribute operator (.) is used to access the attributes of an object

Syntax

```
object.attribute
```

- **Method calls invoke a function, passing in 0 or more comma separated values, returning the functions result**

Syntax

```
object.method(arg [,arg]*)
```

Testing String Contents with `isx()` Methods

- **Python has a large set of methods that test the contents of a string**
  - Return true if the string matches the test
  - Return false of the string does not match the test

- **`str.isalnum()`**
  - Returns true if all characters in the string are alphanumeric and there is at least one character, false otherwise

- **`str.isalpha()`**
  - Returns true if all characters in the string are alphabetic and there is at least one character, false otherwise. Alphabetic characters are those characters defined in the Unicode character database as "Letter"

- **`str.isdecimal()`**
  - Returns true if all characters in the string are decimal characters and there is at least one character, false otherwise

- **`str.isdigit()`**
  - Returns true if all characters in the string are digits and there is at least one character, false otherwise

Testing String Contents with `isx()` Methods

- **`str.isidentifier()`**
  - Returns true if the string is a valid identifier according to the language definition
- **`str.islower()`**
  - Returns true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise
- **`str.isnumeric()`**
  - Returns true if all characters in the string are numeric characters, and there is at least one character, false otherwise. Numeric characters include digit characters, and all characters that have the Unicode numeric value property
- **`str.isprintable()`**
  - Returns true if all characters in the string are printable or the string is empty, false otherwise. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator"
- **`str.isspace()`**
  - Returns true if there are only whitespace characters in the string and there is at least one character, false otherwise

Testing String Contents with `isx()` Methods

- **`str.istitle()`**
  - Returns true if the string is a title cased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise

- **`str.isupper()`**
  - Returns true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise

Example

```
#!/usr/bin/Python3
x = input("Please type in a string of Upper Case Alpha
Characters: ")
if x.isalpha() and x.isupper():
    print("You may pass!")
else:
    print("Off to the volcano!")

#Output
C:\HOTTPython\Mod05\Examples>python istest.py
Please type in a string of Upper Case Alpha Characters:
HELLO
You may pass!

C:\HOTTPython\Mod05\Examples>python istest.py
Please type in a string of Upper Case Alpha Characters:
nope
Off to the volcano!
```

Changing a String's Case

- **Python has a number of string methods that can alter the case of a string**
  - Each of these strings returns an altered copy of the string
- **`str.capitalize()`**
  - Return a copy of the string with its first character capitalized and the rest lowercased
- **`str.lower()`**
  - Return a copy of the string with all the cased characters converted to lowercase
- **`str.swapcase()`**
  - Return a copy of the string with uppercase characters converted to lowercase and vice versa
- **`str.title()`**
  - Return a title cased version of the string where words start with an uppercase character and the remaining characters are lowercase
- **`str.upper()`**
  - Return a copy of the string with all the cased characters converted to uppercase

Example: Changing String Case

Example

```
#!/usr/bin/Python3
m1 = 'And Now for Something Completely Different'
print(m1.capitalize())
print(m1.upper())
print(m1.lower())
print(m1.title())
print(m1.swapcase())

#Output
And now for something completely different
AND NOW FOR SOMETHING COMPLETELY DIFFERENT
and now for something completely different
And Now For Something Completely Different
aND nOW FOR sOMETHING cOMPLETELY dIFFERENT
```

Strip Spaces from Strings

- **Python supports a few methods that may be used to remove spaces or other undesired characters from strings**

- **`str.strip([chars])`**
  - Return a copy of the string with the leading and trailing characters removed

- **`str.lstrip([chars])`**
  - Return a copy of the string with the leading characters removed

- **`str.rstrip([chars])`**
  - Return a copy of the string with the trailing characters removed

- **Each of these methods supports the `[chars]` argument**
  - The chars argument is a string specifying the set of characters to be removed
    - If omitted or None, the chars argument defaults to removing whitespace
    - The chars argument is not a prefix or suffix; rather, all combinations of its values are stripped

Example: `Strip()`

Example

```
#!/usr/bin/Python3
roomy = '   Roomy         '
url = 'www.traininghott.com'
print(roomy.strip())      #Roomy
print(url.strip('cmo.w')) #traininghott
```

Join and Split Strings

- **The `join()` method may be used to produce a string from an `iterable` object**

Syntax

```
str.join(iterable)
```

- **Returns a string which is the concatenation of the strings in the iterable `iterable`**

- **The split method may be used to split a string into an `iterable` object**

Syntax

```
str.split(sep=None, maxsplit=-1)
```

- **Return a list of all the words in the string, using `sep` as the separator**
  - The default is to split on whitespace when left unspecified

- **Optionally, split may limit the number of splits to maxsplit**
  - If maxsplit is not specified or -1, then there is no limit on the number of splits

Example: `join()` and `split()`

Example

```
mpfs = ('John Cleese','Terry Gilliam','Eric Idle',
    'Terry Jones','Michael Palin','Graham Chapman')

delim = ' , '
jmpfs = delim.join(mpfs)

print(mpfs)
#('John Cleese', 'Terry Gilliam', 'Eric Idle', 'Terry
#Jones', 'Michael Palin', 'Graham Chapman')

print(jmpfs)
# John Cleese , Terry Gilliam , Eric Idle , Terry Jones
#  , Michael Palin , Graham Chapman

smpfs = jmpfs.split(' , ')

print(smpfs)
#['John Cleese', 'Terry Gilliam', 'Eric Idle', 'Terry
#Jones', 'Michael Palin', 'Graham Chapman']
```

Replace String Contents

- **The string `replace()` method may be used to create a copy of a string with old content replaced with new content, optionally limiting the number of times content is replaced**

Syntax

```
str.replace(old, new[, count])
```

- **Returns a copy of the string with all occurrences of substring `old` replaced by `new`**
  - If the optional argument `count` is given, only the first `count` occurrences are replaced

Example

```
#!/usr/bin/Python3
lunch = 'SPAM SPAM SPAM'
print(lunch)
#SPAM SPAM SPAM
dinner = lunch.replace('SPAM','Steak')
print(dinner)
#Steak Steak Steak
snackThirty = dinner.replace('Steak', 'Big Ol\' Steer', 1)
print(snackThirty)
#Big Ol' Steer Steak Steak
```

Module Summary

- **The String object supports text and binary data**
- **Strings are sequence objects**
- **Quote operators may be used to create string literals**
- **Strings may be indexed and sliced**
- **`%` and `format()` are powerful templating systems for formatting strings**
- **Python's string object has a robust set of built-in methods**

Module 5

Building Structured Data with Lists and Tuples

Module Goals and Objectives

Major Goals

Understand the differences between Lists and Tuples

Understand the purpose of Lists

Work with Lists

Understand the purpose of Tuples

Work with Tuples

Specific Objectives

On completion of this module, students will be able to:

Build, Access, and Destroy Lists

Use sequence expressions with Lists

Sort and Reverse Lists

Use List methods to build stacks

Build, Access and Destroy Tuples

Access Tuple Metadata

Maintain data integrity with Tuples

Section 5–1

Overview of Lists and Tuples

Lists and Tuples

- **Lists are mutable sequences of objects**
- **Lists are extremely flexible, ordered collections of objects**
  - The list is also one of the main work structures in Python
- **Tuples are immutable sequences of objects**
- **Like the list, tuples are ordered collections of objects**
- **The immutability of the tuples provides a level of integrity because the tuple won't be changed by the program**
- **Guido van Rossum has been quoted as saying that he sees "..a tuple as a simple association of objects and a list as a data structure that changes over time.."**

Section 5–2

Working with Lists

Overview of Lists

- **Lists are typically used to store ordered collections of objects**
  - Lists are Python's most flexible ordered collection type
- **Functionally, lists are ordered groups of arbitrary objects**
  - Lists maintain ordering among the elements they contain
- **Elements in a list are accessed by an integer-based index**
  - Lists maintain a relationship called a "mapping" between indexes and an element
    - Each index maps to an element
- **Lists are extremely flexible**
  - Lists are variable in length
  - May store heterogeneous objects
  - Other data structures may be arbitrarily nested within a list
- **Lists are mutable sequences**
  - Lists respond to all of the sequence operations
- **Lists in Python are Arrays of object references**
  - When an object is assigned to a data structure, Python stores a reference to that object

Creating Lists

- **Python supports a number of techniques to create lists, including methods and literal syntax**

- **The `list()` constructor creates a list of all items in any iterable**
  - If no arguments are passed then an empty list is returned

syntax

```
list([iterable])
```

example

```
# A Tuple of numbers is used as the iterable argument for
the list constructor and a list is returned
x = list((1,2,3)) # [1, 2, 3]
```

- **Because lists are dynamic, the preferred method to create a list is to use the list literal `[]`**
  - The list literal operator `[]` may be used to create lists, access elements, for slicing and list comprehension operations

Syntax

```
[]
```

Example

```
x = [1,2,3]
```

Example: Creating Lists

Example

```
mpfc =[ 'Graham Chapman',
        'Eric Idle',
        'Terry Gilliam',
        'Terry Jones',
        'John Cleese',
        'Michael Palin']
print(mpfc)
```

Output

```
['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
```

Example

```
mps = 'Graham Chapman, Eric Idle, Terry Gilliam, Terry
Jones, John Cleese, Michael Palin'
mpfcl = list(mps.split(','))

print(mpfcl)
```

Output

```
['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
```

Creating Empty Lists

- **A list that contains no elements is an empty list**

- **To create an empty list, assign a variable a set of empty brackets `[]`, or assign a `list()` constructor with no arguments**

Syntax

```
x = []
x = list()
```

- **Empty lists may be useful if you intend on loading objects into the list through list methods**

- **Empty lists are also useful to release resources currently being consumed by an existing list**

Example: Empty Lists

Example

```
#!/usr/bin/Python3

mpfc =[ 'Graham Chapman','Eric Idle','Terry Gilliam','Terry
Jones','John Cleese','Michael Palin']

print(mpfc)

mps = 'Graham Chapman,Eric Idle,Terry Gilliam,Terry
Jones,John Cleese,Michael Palin'
mpfcl = list(mps.split(','))

print(mpfcl)

mpfc = []
mpfcl = list()

print(mpfc)
print(mpfcl)
```

Output

```
['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
[]
[]
```

Working With List Elements

- **Lists are mutable, so lists support operations to access or change list elements in place**

- **Individual list elements may be accessed by using the square brackets to slice the list along an index or a range of indexes**
  - Lists in Python use 0 based indexes

Syntax

```
x[int]
```

- **Because lists are mutable, list elements may also take assignments**
  - Python replaces the object reference at the designated offset with a new one

Syntax

```
x[int] = expr
```

- **Accessing an index that is out of the current range of indexes generates an error**

```
x = []
x[0] = 'a'

IndexError: list assignment index out of range
```

Accessing Lists With Slices

- **Slicing syntax is one of the most convenient and effective methods of returning elements from lists**
  - Sliced may be used to return entire sections of lists in a single step
- **To work with slices, index a sequence with a pair of colon `[ : ]` separated offsets**
  - Python will return everything from the first offset, up to but not including the second offset
  - The left offset is used as the lower bound of the slice
  - The right offset is used as the non inclusive upper bound

Syntax

```
seq[i : j]
```

Example

```
mpfc =[ 'Graham Chapman','Eric Idle','Terry Gilliam','Terry
Jones','John Cleese','Michael Palin']
print(mpfc[0:2])
# ['Graham Chapman', 'Eric Idle']
```

Review: Slice Operations

- **`list[i : j]`**
  - Fetches everything between `i` and `j`, non-inclusive of `j`
- **`list[i :]`**
  - Omitting the second offset fetches everything from the first offset to the end
- **`list[: j]`**
  - Omitting the first offset returns everything from the beginning of the sequence to the second offset, non-inclusive of that offset
- **`list[:-j]`**
  - Omitting the first offset and using a negative second offset returns everything from the front of the sequence to the position of the negative offset, non inclusive of that offset, counting from the end of the sequence
- **`list[-i:]`**
  - Using a negative first offset and omitting the second offset returns everything from the first offset (located by counting backwards from the end of the sequence) to the end of the sequence
- **`list[:]`**
  - Fetches everything. Makes a top level copy

Example: Slicing Operations

Example

```
mpfc =['Graham Chapman','Eric Idle','Terry Gilliam','Terry
Jones','John Cleese','Michael Palin']

print(mpfc[0:2])
# ['Graham Chapman', 'Eric Idle']

print(mpfc[:3])
# ['Graham Chapman', 'Eric Idle', 'Terry Gilliam']

print(mpfc[3:])
# ['Terry Jones', 'John Cleese', 'Michael Palin']

print(mpfc[-3:])
# ['Terry Jones', 'John Cleese', 'Michael Palin']

print(mpfc[::2])
#'Graham Chapman', 'Terry Gilliam', 'John Cleese']
```

Using Slices to Change Lists

- **Slice operations may also be used to change the contents of a list**
  - Slice operations alter entire sections of a list in a single step
- **To replace items, assign a list to a slice of 1 or more items**
  - The number of items in the new list must match the number of items in the slice

Syntax

```
l[i:j] = [elem [,elem]*]
```

- **To insert items into a list, assign a new list to a zero element slice**
  - This is accomplished by setting the start and end offsets to the position in the list where you want to insert the new items

Syntax

```
l[i:i] = [elem [,elem]*]
```

- **To delete items, assign an empty list to a slice**

Syntax

```
l[i:j] = []
```

Example: Using Slices to Change Lists

Example

```
#!/usr/bin/Python3

mpfc =[ 'Graham Chapman','Eric Idle','Terry Gilliam','Terry
Jones','John Cleese','Michael Palin']
mpfcbku = mpfc[:]    #Make a backup of the Flying Circus


mpfc[0:2] = ['Scott Thompson', 'Dave Foley']
#Replace Graham and Eric with a couple of Kids

print(mpfc)
#['Scott Thompson', 'Dave Foley', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']


mpfc[0:0]=mpfcbku[0:2]
#Add Graham and Eric back into the Circus. Keep the Kids

print(mpfc)
#['Graham Chapman', 'Eric Idle', 'Scott Thompson', 'Dave
Foley', 'Terry Gilliam', 'Terry Jones', 'John Cleese',
'Michael Palin']


mpfc[2:4]=[]
#Release the Kids from their contract

print(mpfc)
#['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
```

Copying Lists

- **There are a number of techniques to copy lists**
  - Since lists are mutable, it is often useful to make a copy of the list before manipulating the list
  - The technique is also useful to maintain an original version of the list
- **Use an existing list as the iterable argument for the `list()` constructor and assign the results to a new variable**
- **Use the slicing syntax `list[:]` to generate a top level copy and assign the results to a new variable**
- **Use the sequence `copy()` method to create a top level copy of a list**
  - The `copy()` method creates a shallow copy of a sequence, equivalent to `list[:]`

Syntax

```
sequence.copy()
```

Example: Copying Lists

Example

```
mpfc =[ 'Graham Chapman','Eric Idle','Terry Gilliam','Terry
Jones','John Cleese','Michael Palin']

x = mpfc

print("x's id is ", id(x))
print("mpfc's id is ", id(mpfc))

#x's id is  50910712
#mpfc's id is  50910712

a = list(mpfc)
b = mpfc[:]
c = mpfc.copy()

print("a's id is ", id(a))
print("b's id is ", id(b))
print("c's id is ", id(c))

a's id is  51230320
b's id is  50900904
c's id is  50904760
```

Basic List Operations

- **Lists support all sequence operations and all mutable sequence operations**

- **The + operator may be used to concatenate lists together**

Syntax

```
list + list
```

- **The * operator may be used to repeat a list**

Syntax

```
list * int
```

- **The `in` operator may be used to test for list membership**

Syntax

```
value in list
```

- **The `not in` operator may be used to test if a member is not in a list**

Syntax

```
value not in list
```

Example: Basic List Operations

Example

```
#!/usr/bin/Python3

g1 = ['Graham Chapman','Eric Idle','Terry Gilliam']
g2 = ['Terry Jones','John Cleese','Michael Palin']
mpfc =g1 + g2

print(mpfc)

print('Graham Chapman' in mpfc)          # Well, in spirit.
print('Dave Foley' not in mpfc)

print('Say Ni one more time' , ['Ni ' * 4])

#Output
['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
True
True
Say Ni one more time ['Ni Ni Ni Ni ']
```

The `len()` and `range()` functions

- **The `len()` function returns the number of elements in a sequence**

Syntax

```
len(sequence)
```

- **The `range()` function returns a list of indexes from 0 to `n-1`**

Syntax

```
range(stop)
range(start, stop [,stride])
```

- **`range()` and `len()` are often used in combination to iterate over the indexes of a sequence**

Example

```
#!/usr/bin/Python3

g1 = ['Graham Chapman','Eric Idle','Terry Gilliam']

for i in range(len(g1)):
    print(i , g1[i])

e = 'Eric Idle'
for i in range(len(e)):
    print(i , e[i])
```

Basic Mathematical Functions and Lists

- **The `min(iterable)` function returns the smallest item in a non empty iterable**

- **The `max(iterable)` function returns the largest item in a non empty iterable**

- **The `sum(iterable [,start])` function adds up all the items in an iterable plus an optional integer argument start**
  - Elements in the iterable should be numbers
  - The default value for the `start` argument is `0`. Python adds the values of the iterable to the `start` argument

Example

```
#!/usr/bin/Python3

nums = list(range(10))
print(min(nums))
print(max(nums))
print(sum(nums))
print(sum(nums, 10))


Output
# 0
# 9
# 45
# 55
```

Section 5–3

List Methods

List Methods Overview

- **Along with all sequence operations and mutable sequence operations, lists have a very powerful set of built-in methods**
- **Lists support methods for treating the list as a stack**
- **Lists support methods for sorting operations**
- **Lists may be expanded or contracted**
- **Lists also support basic searching methods**

Adding Elements to a List

- **Lists have a number of methods to add elements to a list**

- **The `append()` method inserts a single element at the end of a list, changing the list in place**

  – This is equivalent to a "push" in most programming languages

Syntax

```
list.append(element)
```

- **The `extend()` method inserts every element in an iterable at the end of a list, changing the list in place**

  – Equivalent to `list[len(list):] = list(elements)`

Syntax

```
list.extend(iterable)
```

- **The `insert()` method inserts an element into a list at an offset**

  – Equivalent to `list[i:i] = [element]`

Syntax

```
list.insert(i, element)
```

Example: Adding Elements to a List

Example

```
#!/usr/bin/Python3

mpfc =[ 'Graham Chapman','Eric Idle','Terry Gilliam','Terry
Jones','John Cleese','Michael Palin']

mpfc.append('Dave Foley')
print(mpfc)

#['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin', 'Dave Foley']

mpfc.extend(['Bruce McCulloch','Kevin McDonald'])
print(mpfc)

#'Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin', 'Dave Foley',
'Bruce McCulloch', 'Kevin McDonald']

mpfc.insert(len(mpfc), 'Mark McKinney')
print(mpfc)

#['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin', 'Dave Foley',
'Bruce McCulloch', 'Kevin McDonald', 'Mark McKinney']

mpfc.insert(0,'Scott Thompson')
print(mpfc)

#['Scott Thompson', 'Graham Chapman', 'Eric Idle', 'Terry
Gilliam', 'Terry Jones', 'John Cleese', 'Michael Palin',
'Dave Foley', 'Bruce McCulloch', 'Kevin McDonald', 'Mark
McKinney']
```

Remove Elements from a List

- **Lists have a number of methods to remove and return elements from a list**

- **Python also has a statement that may be used to delete elements from a list**

- **The `pop()` method removes and returns the last item in a list or a specific index from a list , changing the list in place**

Syntax

```
list.pop([idx])
```

- **The `remove()` method deletes the first occurrence of an element in a list.**
  - Raises an exception if the element is not found

Syntax

```
list.remove(elem)
```

- **The `del()` statement may be used to delete names, elements, slices, etc...**

Syntax

```
del name | del name[i] | del[i:j]
```

Example: Removing Elements from a List

Example

```
#!/usr/bin/Python3

mpfcKids =['Scott Thompson', 'Graham Chapman', 'Eric Idle',
           'Terry Gilliam', 'Terry Jones', 'John Cleese',
           'Michael Palin', 'Dave Foley',
            'Bruce McCulloch', 'Kevin McDonald',
            'Mark McKinney']

mpfcKids.pop()      # Bye Mark
print(mpfcKids)

mpfcKids.pop(-2)    # Later Bruce
print(mpfcKids)

del(mpfcKids[-2:])  # C-Ya Dave and Kevin
print(mpfcKids)

mpfcKids.remove('Scott Thompson')
# We hardly knew ya, Scott
print(mpfcKids)
```

Output

```
['Scott Thompson', 'Graham Chapman', 'Eric Idle', 'Terry
Gilliam', 'Terry Jones', 'John Cleese', 'Michael Palin',
'Dave Foley', 'Bruce McCulloch', 'Kevin McDonald']

['Scott Thompson', 'Graham Chapman', 'Eric Idle', 'Terry
Gilliam', 'Terry Jones', 'John Cleese', 'Michael Palin',
'Dave Foley', 'Kevin McDonald']

['Scott Thompson', 'Graham Chapman', 'Eric Idle', 'Terry
Gilliam', 'Terry Jones', 'John Cleese', 'Michael Palin']

['Graham Chapman', 'Eric Idle', 'Terry Gilliam', 'Terry
Jones', 'John Cleese', 'Michael Palin']
```

Sorting Lists

- **The list `sort()` method may be used to sort a list in place**
  - The `sort()` method sorts ascending by default

Syntax

```
list.sort() | list.sort(key = None, reverse = False)
```

  - The key argument takes a function that is used to compute a comparison value from each list element
  - The reverse function, when set to true, will sort the list elements as if each comparison were reversed

- **The list `reverse()` function reverses a list in place**

Example

```
mpfc =['Graham Chapman', 'Eric Idle', 'Terry Gilliam',
'Terry Jones', 'John Cleese', 'michael palin']
mpfc.sort()
print(mpfc)
# ['Eric Idle', 'Graham Chapman', 'John Cleese', 'Terry
# Gilliam', 'Terry Jones', 'michael palin']

mpfc.sort(key=str.lower)
print(mpfc)
# ['Eric Idle', 'Graham Chapman', 'John Cleese', 'michael
# palin', 'Terry Gilliam', 'Terry Jones']
mpfc.sort(key=str.lower, reverse = True)
print(mpfc)
# ['Terry Jones', 'Terry Gilliam', 'michael palin', 'John
# Cleese', 'Graham Chapman', 'Eric Idle']
```

Searching through a List

- **The list `count()` method searches through a list and returns the number of times an item is found**

Syntax

```
list.count(item)
```

Example

```
eatsteak ="Eat steak eat steak eat a big ol steer Eat steak
eat steak do we have one dear?"
rev = eatsteak.split(' ')
print("The word steak appears in eatsteak",
rev.count('steak'), "times") # 4
```

- **The `index()` method returns the index of the first occurrence of an object in a list**
  - Raises an exception if object is not found

Syntax

```
list.index(obj [,i [,j]])
```

  - Optionally, integers may be passed to `i` and `j - 1` to specify a range of elements to search through

Example

```
print('The first occurence of steak is found at position',
rev.index('steak'))       # Position 1
print('steak between indexes 7 and 10 is found at',
rev.index('steak',7, 11)) # Position 10
```

Section 5–4

Working with Tuples

Tuple Overview

- **Tuples are ordered collections of arbitrary objects**
  - They are sequences like strings and lists
- **Tuples are immutable**
  - Tuples do not support in-place change operations like lists do
  - Because the Tuples cannot be changed in place, they provide a level of integrity for the objects stored in them
- **Like Lists, Tuples are accessed by an integer based index**
  - Python maintains a mapping between the index and the element the index points to
- **Tuples in Python are Arrays of object references**
  - When an object is assigned to a data structure, Python stores a reference to that object
- **Like Lists, Tuples may be nested**

Creating Tuples

- **Python supports a number of ways to create Tuples**

- **Tuples may be created by placing a comma separated list of objects into parentheses**
  - When creating a Tuple of one object, place a trailing comma after the item
  - This trailing comma is an indicator to Python that the parenthesis are being used to create a Tuple and *not* to brace an expression to alter the order of operations

- **Tuples may also be created by assigning a comma separated list of values to a variable**

- **Tuples may also be created using the `tuple()` constructor**
  - The `tuple()` constructor takes an iterable as its argument and returns a tuple

Syntax

```
tuple([iterable])
```

- **To create an empty Tuple or overwrite the contents of a Tuple with an empty Tuple**
  - Assign an empty Tuple
  - Assign an empty `tuple()` constructor

Example: Creating and Emptying Tuples

Example

```
#!/usr/bin/Python3
p1 = 'John','Cleese'
p2 = ('Eric','Idle')
p3 = ('Stig',)
p4 = tuple("Monty")

print(type(p1), type(p2),type(p3),type(p4))
print(p1,p2,p3,p4)

# <class 'tuple'> <class 'tuple'> <class 'tuple'> <class
'tuple'>

# ('John', 'Cleese') ('Eric', 'Idle') ('Stig',) ('M', 'o',
'n', 't', 'y')

p1 = ()
p2 = tuple()
print(type(p1), type(p2),type(p3),type(p4))
print(p1,p2,p3,p4)

#<class 'tuple'> <class 'tuple'> <class 'tuple'> <class
'tuple'>

#() () ('Stig',) ('M', 'o', 'n', 't', 'y')
```

Accessing Tuple Items

- **Individual tuple elements may be accessed by using the square brackets to slice the list along an index or a range of indexes**
  - Tuples in Python use 0-based indexes
- **To work with slices, index a sequence with a pair of colon (:) separated offsets**
  - Slicing operations return *new* tuples
- **All of the sequence access operations discussed in this course work for accessing tuples**

Example

```
#!/usr/bin/Python3
emp = ('Jeremy Clarkson','Richard Hammond','James May',
'The Stig')
HamMay = emp[1:3]
print(emp[0], emp[1])
print(emp[-1])
print(HamMay, emp[-3:])
```

Output

```
Jeremy Clarkson Richard Hammond
The Stig
('Richard Hammond', 'James May') ('Richard Hammond', 'James
May', 'The Stig')
```

Basic Tuple Operations

- **Use the `len()` function to return the length of a tuple**

  Syntax

```
len(iterable)
```

- **The + operator concatenates tuples together, returning a new tuple**

  Syntax

```
tuple + tuple
```

- **The * operator returns a new tuple repeated `n` number of times**

  Syntax

```
tuple * n
```

- **The `in` and `not in` operators may be used to test for tuple membership**

  Syntax

```
value in tuple
value not in tuple
```

Example: Basic Tuple Operations

Example

```
#!/usr/bin/Python3
emp1 = ('Jeremy Clarkson','Richard Hammond')
emp2 =('James May', 'The Stig')
emps = emp1 + emp2 # Top Gear! Unite!

malkovich = 'Malkovich',
beingJohn = malkovich * 3 # You saw the movie, right?

print(emps)
# ('Jeremy Clarkson', 'Richard Hammond', 'James May',
# 'The Stig')

print(beingJohn)
# ('Malkovich', 'Malkovich', 'Malkovich')

print('Jeremy Clarkson' in emps)   #true
print('Malkovich' in beingJohn)    #true
print('Das Stig' not in emps)      #true
```

Basic Mathematical Functions and Tuples

- **The `min(iterable)` function returns the smallest item in a non empty iterable**

- **The `max(iterable)` function returns the largest item in a non empty iterable**

- **The `sum(iterable [,start])` function adds up all the items in an iterable plus an optional integer argument start**
  - Elements in the iterable should be numbers
  - The default value for the `start` argument is `0`. Python adds the values of the iterable to the `start` argument

Example

```
#!/usr/bin/Python3

nums = tuple(range(15))
print(min(nums))
print(max(nums))
print(sum(nums))
print(sum(nums, 10))

Output
# 0
# 14
# 105
# 115
```

Tuple Methods

- **Tuples have 2 built-in methods**
  - `index()`
  - `count()`
- **The `index()` and `count()` methods behave the same way they do with lists**

Example

```
#!/usr/bin/Python3
malk = """There's a tiny door in my office, Maxine.
    It's a portal and it takes you inside John Malkovich.
    You see the world through John Malkovich's eyes...
    and then after about 15 minutes, you're spit out...
    into a ditch on the side of the New Jersey Turnpike."""
print("The word Malkovich appears",
malk.count('Malkovich'), "times") # 2
print('The first occurence of Malkovich is found at
position', malk.index('Malkovich')) # 89
```

- **To `sort()` or `reverse()` a tuple, you have to convert the tuple to a list , perform the operations, then convert back to tuple**

Example

```
emp = ('Jeremy Clarkson','Richard Hammond','James May')
emp1 = list(emp); emp1.sort(); emp = tuple(emp1)
emp1 = []
print(emp)
#('James May', 'Jeremy Clarkson', 'Richard Hammond')
```

Creating Records with Tuples

- **Because tuples are immutable, they are a reliable structure to use to represent records**

- **Tuples may be embedded in tuples like lists may be embedded in lists, and lists may be embedded in tuples, etc...**
  - This flexibility gives the ability to create multi-dimensional structures that can reliably represent the type of data that is read from a table in a database while maintaining the integrity of the data from the original source

- **To create multi-dimensional tuples, simply create a tuple as an item in another tuple**

Example

```
x = (('a' ,'b') ,('c' ,'d'), ('e' ,'f'))
```

  - The example code creates a 2 dimensional structure, the equivalent of a 3 row, 2 column table

2	e	f
1	c	d
0	a	b
	0	1

  - To access the values of the subordinate tuples, add one set of tuple access operators `[]` for each dimension

Example

```
x[0][0]  # a
```

Example: Building Immutable Records with Tuples

Example

```
#!/usr/bin/Python3
employees = (('Jeremy Clarkson','The Host'),
    ('Richard Hammond', 'The Talent'),
    ('James May', 'The Brains'),
    ('The Stig', 'The Muscle'))

for employee in employees:
    for field in employee:
        print(field, ' ' , end ='')
         #This print is attached to the inner for loop

    print()  #This print is attached to the outer for loop

#Output
#Jeremy Clarkson  The Host
#Richard Hammond  The Brains
#James May  The Talent
#The Stig  The Muscle

print(employees[0][0] , "is", employees[0][1] )
# Jeremy Clarkson is The Host
```

Creating Tuples with Named Fields

- **The collections module implements specialized container data types providing alternatives to Python's general purpose built-in containers**
- **One of those specialized data types is the `namedtuple`**
- **Named tuples assign meaning to each position in a tuple and allow for more readable, self-documenting code**
- **Named tuples can be used wherever regular tuples are used**
  - They add the ability to access fields by name instead of position index
- **Named tuple instances are just as memory efficient as regular tuples because they do not have per-instance dictionaries**
- **Named tuples are especially useful for assigning field names to result tuples returned by the `csv` or `sqlite3` modules, or any code that generates records as tuples**

Creating Named Tuples

- **Named tuples must be imported from the collections module**

Syntax

```
import collections

from collections import namedtuple
```

- **An instance of a named tuple object may then be created**

Syntax

```
nto = collections.namedtuple(typename, field_names)
```

  - Returns a new tuple subclass named `typename`
    - The new subclass is used to create tuple-like objects that have fields accessible by attribute lookup as well as being index able and iterable
  - Instances of the subclass also have a helpful docstring `help(namedtupleobject)`
  - The `field_names` are a sequence of strings such as ['x', 'y']
    - Alternatively, `field_names` can be a single string with each fieldname separated by whitespace and/or commas, for example `'x y'` or `'x, y'`
    - Any valid Python identifier may be used for a fieldname except for names starting with an underscore

Accessing Named Tuple Attributes

- **Information in a named tuple may be accessed a number of different ways**

- **To view a string representation of your named tuple, print the namedtuple**

Syntax

```
print(nto)
```

- **Values of fields may be accessed by using the field names as attributes of the named tuple object**

Syntax

```
nto.field_name
```

- **Values of fields may also be accessed by using a 0-based integer index lookup**

Syntax

```
nto[int]
```

- **The Python built-in function `getattr()` method may also be used to return the value of a named tuple attribute**

Syntax

```
getattr(object, attribute)
```

Example: Creating a Record with a Named Tuple

Example

In this example we are going to create a couple of employee records using named tuples. We will then access the named tuples using various methods

```
import collections

Employee =
collections.namedtuple('Employee', 'name eid pos')

print('Type of Person:', type(Employee))

jenny = Employee(name='Jenny', eid='8675309', pos='Muse')
print('Representation:', jenny)

sonic = Employee(name='Sonic', eid='6231991',
pos='hedgehog')
print('Field by name:', sonic.name)

emps = [ jenny, sonic]

print('Fields by index:')
for p in emps:
    print('{} is Employee Number {} and is a{}'.format(*p)
        )


#Type of Person: <class 'type'>
#Representation: Employee(name='Jenny', eid='8675309',
#pos='Muse')
#Field by name: Sonic
#Fields by index:
#Jenny is Employee Number 8675309 and is a Muse
#Sonic is Employee Number 6231991 and is a hedgehog
```

Module Summary

- **Lists are arbitrary sequences of mutable objects**
- **Tuples are arbitrary sequences of immutable objects**
- **Lists are the most flexible data type in Python**
- **Tuples are used to maintain the integrity of their data**
- **All sequence operations may be used with lists**
- **Lists support a number of built-in methods which may alter the list**
- **Some sequence operations work with tuples and most likely return a new tuple**
- **Lists and tuples may be used to create multi-dimensional structures**
- **Named tuples may be used to create field names for tuples**

Module 6

Building Structured Data with Dictionaries and Sets

Module Goals and Objectives

Major Goals

Understand the differences between Dictionaries and Sets

Understand the purpose of Dictionaries

Work with Dictionaries

Understand the purpose of Sets

Work with Sets

Understand the relationship between Dictionaries and Sets

Specific Objectives

On completion of this module, students will be able to:

Create, Access and Destroy Dictionaries

Create, Access and Destroy Sets

Retrieve Keys and Values from Dictionaries

Use Dictionaries to create lookup tables

Create Frozensets

Use View objects

Section 6–1

Overview of Dictionaries and Sets

Dictionaries and Sets

- **Dictionaries are mutable tables of object references**
  - Dictionaries are similar to lists in that respect
- **Unlike lists, which are accessed by an integer based index, dictionaries are accessed by key**
  - This makes the dictionary a flexible structure that may be used to create lookup tables, or create records based off of field / value associations
- **Sets are mutable collections of immutable objects**
- **Sets are typically used for computing mathematical operations such as creating a union of sets or creating an intersection of sets**
  - Sets are also used for removing duplicates from a sequence or performing membership testing
- **Sets behave like a "valueless dictionary"**

Section 6–2

Working with Dictionaries

Dictionary Overview

- **Dictionaries, like lists, are one of the most flexible data types**
- **Dictionaries are mutable collections of objects**
  - Dictionaries are unordered, unlike their list counterparts which are ordered
- **Internally, dictionaries are implemented as dynamically expandable hash tables**
- **Items in a dictionary are stored as key / value pairs**
  - Values in a dictionary are accessed by the key, not a positional index
- **Dictionaries may grow and shrink as needed**
- **Dictionaries may contain any object**
  - May be nested to any depth
- **Each key may only have 1 associated value**
  - That value may be a collection
- **Dictionaries support a robust array of methods to access and process the contents of a dictionary**

Creating Dictionaries

- **While there are a number of ways to create dictionaries the two most common methods are to use the dictionary literal syntax or the `dict()` method**

- **Dictionaries may be created using the dictionary literal operator `{ }`**
  - Values in the dictionary operator should be a comma-separated list of `key: value` pairs

Syntax

```
x = {}
x = {'k1' : 'v' [, 'kn' : 'vn']* }
```

- **Dictionaries may also be created using the `dict()` method**

- **The `dict()` method returns a new dictionary initialized from a mapping , sequence or keyword arguments**
  - If no arguments are given then `dict()` returns an empty dictionary

Syntax

```
dict([mapping | iterable | keywords])
```

Example: Creating Dictionaries

Example

```
#!/usr/bin/Python3

#Literal Syntax
mprd = {'Something Completely Different' : 1971,
        'Holy Grail': 1975,
        'Life of Brian': 1979,
        'Live at the Hollywood Bowl': 1982,
        'The Meaning of Life': 1983
        }
print(mprd)

#Keyword syntax
mpdir = dict(hg=['Terry Gilliam', 'Terry Jones'],
            lhb=['Terry Hughes', 'Ian MacNaughton'],
            ml=['Terry Gilliam', 'Terry Jones'],
            scd='Ian MacNaughton',
            lb='Terry Jones'

)
print(mpdir)

#Iterable syntax
cast = dict([('p1' , 'John Cleese'),
            ('p2', 'Terry Gilliam')])
print(cast)
```

Output

```
{'Something Completely Different': 1971, 'Holy Grail':
1975, 'The Meaning of Life': 1983, 'Live at the Hollywood
Bowl': 1982, 'Life of Brian': 1979}

{'ml': ['Terry Gilliam', 'Terry Jones'], 'scd': 'Ian
MacNaughton', 'hg': ['Terry Gilliam', 'Terry Jones'], 'lb':
'Terry Jones', 'lhb': ['Terry Hughes', 'Ian MacNaughton']}

{'p2': 'Terry Gilliam', 'p1': 'John Cleese'}
```

Dictionary Keys and Values

- **Dictionary keys must be immutable**
  - Strings, Numbers and Tuples are legal data types to use as keys
    - Tuples are useful when a key needs to be a multi-valued key
- **Dictionary keys are unique**
  - Writing values to a key that doesn't exist creates the key
  - Subsequent assignments to an existing key assign a new object if the value of the key is immutable
    - String, Number, Tuple
  - Subsequent assignments to an existing key update the reference if the value is mutable
    - List, Dictionary
- **Reading from a key that doesn't exist produces a KeyError**

Example

```
cast = dict([('p1' , 'John Cleese'),('p2', 'Terry
Gilliam')])
print(cast['p3'])
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print(cast['p3'])
KeyError: 'p3'
```

Accessing Values in a dictionary

- **To access values in a dictionary, use the `[]` access operators and supply a key for the lookup instead of an integer**

Syntax

```
d['key']
```

- **The same syntax may be used to update the value of a key**
  - Remember, if the value is immutable then the key will be used to access a new object
  - If the object is mutable, then you will update the object that the key is referring to

Syntax

```
d['key ' ] = expr
```

- **Values may be removed by assigning `None` or the equivalent of a boolean `false` for the data type of the value**

- **Key / Value pairs are not stored in the dictionary in any particular order**
  - Keys must be used to access an associated value

Example: Accessing Dictionary Values

Example

```
#!/usr/bin/Python3

mpdir = dict(hg=['Terry Gilliam', 'Terry Jones'],
             lhb=['Terry Hughes', 'Ian MacNaughton'],
             ml=['Terry Gilliam', 'Terry Jones'],
             scd='Ian MacNaughton',
             lb='Terry Jones')
cast = dict([('p1' , 'John Cleese'),
             ('p2', 'Terry Gilliam')])

print("The directors of Search for the holy grail are" ,
    mpdir['hg'][0] ,
    'and',
    mpdir['hg'][1])

cast['p1'] = 'Steve'
print(cast['p1'])

mpdir['hg'][1] = mpdir['hg'][1].upper()
print(mpdir['hg'][1])
print(mpdir)
```

Output

```
The directors of Search for the holy grail are
Terry Gilliam and Terry Jones
Steve
TERRY JONES
{'ml': ['Terry Gilliam', 'Terry Jones'], 'scd': 'Ian
MacNaughton', 'hg': ['Terry Gilliam', 'TERRY JONES'], 'lb':
'Terry Jones', 'lhb': ['Terry Hughes', 'Ian MacNaughton']}
```

Deleting Dictionary Elements

- **Every item in a dictionary may be removed if necessary**
- **To remove a value, assign `None` to a key or the equivalent of `None` or `False` for that data type**
  - Such as an empty string `''` or an empty list `[]`
  - This technique is also useful to pre declare a key or indicate that a key is required, yet missing a value

Syntax

```
d['key'] = None
```

- **The `del()` statement may be used to delete a dictionary, or key/value pairs.**

Syntax

```
del dict | del dict['key']
```

- **The dictionary `clear()` method removes all entries from the dictionary**

syntax

```
d.clear()
```

Example: Removing Dictionary Items

Example

```
#!/usr/bin/Python3

cast = dict([('p1' , 'John Cleese'),
             ('p2', 'Terry Gilliam'),
             ('p3' , 'Eric Idle')])
print(cast)
#{'p3': 'Eric Idle', 'p2': 'Terry Gilliam', 'p1': 'John
#Cleese'}

cast['p1'] = None
print(type(cast['p1']))
#<class 'NoneType'>

del(cast['p2'])

cast.clear()
print(cast)
#{}

del(cast)
print(cast)
#Message File Name    Line Position
#Traceback
#     <module>
    C:\HOTTPython\Mod07\Examples\removeItems.py    21
#NameError: name 'cast' is not defined
```

Section 6–3

Dictionary Methods

Using Dictionary Methods

- **Dictionary objects support a robust set of built in methods**
- **Dictionary methods may be used to**
  - Retrieve Keys
  - Retrieve Values
  - Retrieve Key / Value pairs
  - Reliably work with keys without producing syntax failures
  - Merge Dictionaries
  - Create new dictionaries from existing dictionaries
  - Set default keys

Differences Between Python 2.x and Python 3.x Dictionaries

- **Python 3 has made some major changes to the behavior of dictionary methods and has removed a number of Python 2 dictionary methods**
  - Python 3 no longer supports the following methods
    - `dict.iteritems()`
    - `dict.iterkeys()`
    - `dict.itervalues()`
- **In Python 3 the `items()`, `keys()` and `values()` methods return view objects instead of list objects**
- **The Python 2 `dict.has_key()` method has been removed from Python 3 in favor of testing key existence with the `in` operator**
- **Python 3 supports a new dictionary comprehension syntax**
  - Discussed later in this course
  - Python 2.7 supports a back port of the new comprehension syntax
- **Python 2.7 supports a back port of the Python 3 iterable views using the `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` methods**

Python 3 View Objects

- **The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views**
- **Dictionary views are lazy sequences that will see changes in the underlying dictionary**
- **They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes**
- **To force the dictionary view to become a full list, use `list(dictview)` to convert the view to a list**
- **Dictionary views can be iterated over to yield their respective data, and support membership tests**

The keys(), values() and items() methods

- **The `keys()` method returns an iterable view object comprised of all the keys in a dictionary**

Syntax

```
d.keys()
```

- **The `values()` method returns an iterable view object comprised of all the values in a dictionary**

Syntax

```
d.values()
```

- **The `items()` method returns an iterable view object of tuple pairs, one tuple pair per key / value pair in the dictionary**

Syntax

```
d.items()
```

- **The return values of all of these methods may be converted into lists**
  - Creates symmetry with Python 2.x type operations
- **Each of these methods may be used as drivers for loop constructs—such as `for` loops or `while` loops—and may be used as part of the new dictionary comprehensions**
  - Discussed later in this course

Example: keys() , values() and items() Methods

Example

```
#!/usr/bin/Python3

mprd = {'Something Completely Different' : 1971,
        'Holy Grail': 1975,
        'Life of Brian': 1979,
        'Live at the Hollywood Bowl': 1982,
        'The Meaning of Life': 1983
        }
titles = mprd.keys()
dates = list(mprd.values())
dates.sort()
release = mprd.items()
print(titles)
print(release)
print("Monty Python movies were released on all of these
years" , end = ' ')
for year in dates: print(year, end = ' ')
```

Output

```
dict_keys(['The Meaning of Life', 'Something Completely
Different', 'Holy Grail', 'Life of Brian', 'Live at the
Hollywood Bowl'])

dict_items([('The Meaning of Life', 1983), ('Something
Completely Different', 1971), ('Holy Grail', 1975), ('Life
of Brian', 1979), ('Live at the Hollywood Bowl', 1982)])

Monty Python movies were released on all of these years
1971 1975 1979 1982 1983
```

The `get()` method

- **One of the issues in accessing a key is if the key doesn't exist, Python raises an exception**

- **The `get()` method returns the value of a key if the key exists; otherwise `get()` returns none**
  - `get()` also allows for returning a default value other than None

Syntax

```
d.get(key [,default])
```

Example

```
#!/usr/bin/Python3
eng_to_esp = {}
eng_to_esp['one'] = 'Uno'
eng_to_esp['two'] = 'Dos'
eng_to_esp['three'] = 'Tres'

print("The number one in Spanish is", eng_to_esp['one'])

# Lookup of nonexistent key throws a KeyError
#print("The number four in Spanish is", eng_to_esp['four'])
#Message File Name    Line Position
#Traceback
#    <module>    C:\HOTTPython\Mod07\Examples\get.py    8
#KeyError: 'Four'

# Lookup of nonexistent key proves no significant problem
print("The number four in Spanish is",
eng_to_esp.get('four', 'No Translation Found'))

The number one in Spanish is Uno
The number four in Spanish is No Translation Found
```

The `setdefault()` and `fromkeys()` methods

- **The `setdefault()` method returns the value of a key if the key is in the dictionary**
  - Otherwise, `setdefault()` inserts a key with the default value and returns the default value
    - The default defaults to None

Syntax

```
d.setdefault(key [,default])
```

- **The `fromkeys()` method creates a new dictionary from a sequence and a default value for each item in the sequence**
  - The sequence provides the key names
  - The values of the keys are set to a default value
    - The default defaults to None
  -

Syntax

```
d.fromkeys(sequence [, default])
```

Example: `fromkeys()` and `setdefault()`

Example

```
#!/usr/bin/python

basket = ('T-Bone', 'Chuck', 'Porterhouse', 'Rib eye')
meatgroup = {}
meatgroup = meatgroup.fromkeys(basket, 0)
# Or meatgroup = {}.fromkeys(basket, 0)
# if you're feeling up to it
print(meatgroup)

meatgroup['T-Bone'] = 12
meatgroup['Chuck'] = 8
meatgroup['Porterhouse'] = 4

print(meatgroup.setdefault('Porterhouse', 11))
# 4 Set manually
print(meatgroup.setdefault('Rib eye', 11))
# 0 Set by fromkeys()
print(meatgroup.setdefault('TopSirloin', 11))
# 11 as in Crank it to

print(meatgroup)
```

Output

```
{'Rib eye': 0, 'Chuck': 0, 'Porterhouse': 0, 'T-Bone': 0}
4
0
11
{'TopSirloin': 11, 'Rib eye': 0, 'Chuck': 8, 'Porterhouse':
4, 'T-Bone': 12}
```

The `update()` method

- **The `update()` method merges two dictionaries together**
  - The originating dictionary performs an in-place merge of key value pairs from another dictionary object

Syntax

```
od.update(ad)
```

- **The `update()` process is equivalent to the following `for` loop**

Syntax

```
for(k,v) in ad.items: od[k] = v
```

- **Python versions 2.4 and up also support using an iterable of key/value pairs and keyword assignments as arguments for the `update()` method**

Syntax

```
od.update(key = 'value' [,keyn = 'valuen']*)
od.update(  [ ('key','value') [,('key','value')]* ]  )
```

Example: `update()` Method

Example

```
#!/usr/bin/Python3
eng_to_esp = {}
eng_to_esp['One'] = 'Uno'
eng_to_esp['Two'] = 'Dos'
eng_to_esp['Three'] = 'Tres'

colors = {}
colors['red'] = 'Rojo'
colors['white'] = 'Blanco'
colors['blue'] = 'Azul'
colors['green'] = 'Verde'
colors['purple'] = 'Morado'

print(eng_to_esp)
print(colors)
eng_to_esp.update(colors) # Dictionary
print(eng_to_esp)
eng_to_esp.update(yellow = 'Amarillo') # Keyword
print(eng_to_esp)
eng_to_esp.update([('magenta','Rosa')]) # Iterable
print(eng_to_esp)
```

Output

```
{'Three': 'Tres', 'One': 'Uno', 'Two': 'Dos'}
{'white': 'Blanco', 'blue': 'Azul', 'red': 'Rojo',
'purple': 'Morado', 'green': 'Verde'}
{'green': 'Verde', 'white': 'Blanco', 'red': 'Rojo',
'blue': 'Azul', 'purple': 'Morado', 'Two': 'Dos', 'One':
'Uno', 'Three': 'Tres'}
{'green': 'Verde', 'white': 'Blanco', 'red': 'Rojo',
'blue': 'Azul', 'purple': 'Morado', 'Two': 'Dos', 'yellow':
'Amarillo', 'One': 'Uno', 'Three': 'Tres'}
{'green': 'Verde', 'white': 'Blanco', 'red': 'Rojo',
'magenta': 'Rosa', 'blue': 'Azul', 'purple': 'Morado',
'Two': 'Dos', 'yellow': 'Amarillo', 'One': 'Uno', 'Three':
'Tres'}
```

Useful Dictionary Expressions

- **The `len()` method returns the length of a dictionary**
  - Length is based on the number of key / value pairs

Syntax

```
len(dictionary)
```

- **The `in` and `not in` operators may be used to perform membership tests**

Syntax

```
key in dictionary
key not in dictionary
```

- **The `clear()` method removes all key / value pairs from a dictionary**

Syntax

```
d.clear()
```

- **The `copy()` method returns a shallow copy of a dictionary**

Syntax

```
nd = d.copy()
```

Example: Using Dictionaries to Count Words

- **This example counts how many times any word occurs in the Monty Python "Lumber Jack Song"**

Example

```
#!/usr/bin/Python3
occurs = {} # The Dictionary
lj = """I never wanted to do this in the first place!
I...  I wanted to be...

A LUMBERJACK!

(piano vamp)

....The rest of the song...
....Perhaps we should read from a file next time?...
"""
for word in lj.split():
    occurs[word] = occurs.get(word, 0) + 1
    #Creates key and sets value to 0
    #Otherwise retrieve value, add 1 and reassign

for word in occurs:
    print("The word", word, "occurs", occurs[word], "times
    in the Lumber Jack Song")
```

Output

```
The word night occurs 2 times in the Lumber Jack Song
The word sleep occurs 1 times in the Lumber Jack Song
The word Wednesdays occurs 2 times in the Lumber Jack Song
The word around occurs 1 times in the Lumber Jack Song
The word thought occurs 1 times in the Lumber Jack Song
...And a lot more output....
```

Section 6–4

Sets

Overview of Sets

- **Introduced with Python 2.4, Sets are a relatively new data type**
- **Sets are an unordered collection of unique and immutable objects**
  - The objects in a set must be hashable
- **Sets themselves are mutable sequences, so they can expand and contract as necessary**
  - Because they are mutable, a set cannot belong to another set
- **Frozensets are immutable sequences**
  - They cannot change once they have been created
  - Because frozensets are immutable and hashable, they can be members of other sets

Uses for Sets

- **Sets support a number of mathematical and database applications**
- **Sets can be used to filter duplicates out of other collections**
  - Sets are collections of unique items, so Python makes sure that items only exist once in a set
- **Sets may be used to isolate differences in sequences like lists and strings**
  - This makes targeting the difference between 2 sequence items trivial
- **Sets may be used to perform order-neutral equality tests**
  - This is extremely useful when 2 sequences need to be compared for equality, regardless of sequencing
  - Sets are said to be equal if every element of each set is contained in the other set
    - Each set is a subset of each other, regardless of the sequence
- **Database operations such as intersections and unions may also be performed on sets**

Creating Sets

- **Sets may be created by calling the `set()` built-in function with an iterable**
  - The `set()` function returns a new set

Syntax

```
set(iterable)
```

- **Python 2.7 and 3.x support a literal syntax for creating set**
- **To create a set using the literal syntax, place an iterable in the set literal operator `{}`**
  - Sets use the `{}` like dictionaries do
  - Set items are unique, unordered, and immutable like dictionary keys
    - This makes the behavior of a set a lot like a valueless dictionary

Syntax

```
{iterable}
```

- **Frozensets may be created by calling the `frozenset()` built-in function with an iterable**
  - The `frozenset()` function returns a new frozenset

Syntax

```
frozenset(iterable)
```

Example: Creating Sets and Frozensets

Example

```
#!/usr/bin/Python3
steak = set(['Chuck', 'Porterhouse', 'Rib eye'])
meat = set('spam')
print(steak)
print(meat)

steakl = {'Chuck', 'Porterhouse', 'Rib eye'}
meat1 = {'s','p','a','m'}
print(steakl )
print(meat1)

frozensteak = frozenset(['Chuck', 'Porterhouse', 'Rib
eye'])
print(frozensteak)
```

Output

```
{'Chuck', 'Rib eye', 'Porterhouse'}
{'s', 'p', 'a', 'm'}
{'Chuck', 'Rib eye', 'Porterhouse'}
{'s', 'p', 'a', 'm'}
frozenset({'Chuck', 'Rib eye', 'Porterhouse'})
```

Set Operations

- **The `in` and `not in` operators may be used to check item membership in a set**

Syntax

```
item in set
item not in set
```

- **The – operator may be used to calculate a set difference**
  - Returns the items in set1 that aren't in set2

Syntax

```
set1 - set2
```

- **The `|` operator may be used to create a set union**
  - Returns the unique items in set1 and set2

Syntax

```
set1 | set2
```

- **The `&` operator may be used to calculate a set intersection**
  - Returns a new set comprised of unique items that exist in both set1 and set2

Syntax

```
set1&set2
```

More Set Operations

- **The <= operator calculates a set subset**
  - Tests whether every item in set1 is also in set2

Syntax

```
set1<=set2
```

- **The >= operator calculates a set superset**
  - Tests whether every element in set2 is also in set1

Syntax

```
set1 >= set2
```

- **The ^ operator calculates a symmetric difference**
  - Returns a new set of items in set1 or set2 but not in both sets

Syntax

```
set1 ^ set2
```

- **The |= operator updates a set with items from another set**
  - Adds items in set2 to set1

syntax

```
set1 |= set2
```

Example: Set Operations

Example

```
#!/usr/bin/Python3
mpfc = set([ 'Graham Chapman','Eric Idle',
'Terry Gilliam','Terry Jones','John Cleese',
'Michael Palin'])

fcw = set(['John Cleese','Michael Palin'])

fempython= set(['Carol Cleveland', 'Eric Idle'])

print(mpfc - fcw) # Difference
#{'Graham Chapman', 'Terry Jones', 'Eric Idle', 'Terry Gilliam'}

print(mpfc | fempython) # Union
#{'John Cleese', 'Graham Chapman', 'Eric Idle', 'Terry Jones',
#'Terry Gilliam', 'Carol Cleveland', 'Michael Palin'}

print(mpfc & fempython)# Intersect
#{'Eric Idle'}

print(fcw <= mpfc) # Subset
#True

print(fempython <= mpfc) # Superset
#False

print(mpfc ^ fcw) # Symetric Difference
#{'Graham Chapman', 'Terry Jones', 'Eric Idle', 'Terry Gilliam'}

print(mpfc ^ fcw ^ fempython) # Symetric Difference
#{'Carol Cleveland', 'Terry Jones', 'Graham Chapman', 'Terry
#Gilliam'}

mpfc |= fempython # Update
print(mpfc)

{'Carol Cleveland', 'Michael Palin', 'John Cleese',
'Eric Idle', 'Terry Jones', 'Terry Gilliam', 'Graham
Chapman'}
```

Set Methods

- **Sets also support a number of methods which may be used to modify the set**
  - Not for use with frozensets. Some assembly required
- **The `add()` method inserts one item into a set**

Syntax

```
set.add(item)
```

- **The `update()` method performs an in place union on a set**

Syntax

```
set.update(iterable)
```

- **The `remove()` method removes an item from a set**

Syntax

```
set.remove(item)
```

- **The `copy()` method makes a copy of a set**

Syntax

```
set.copy()
```

Example: Set Methods

Example

```
#!/usr/bin/Python3

mpfc = set([ 'Graham Chapman','Eric Idle','Terry
Gilliam','Terry Jones','John Cleese','Michael Palin'])
print(mpfc)

mpfc.add('Carol Cleveland')
print(mpfc)

mpfcf = mpfc.copy()
mpfc.remove('Carol Cleveland')
print(mpfc)
print(mpfcf)

mpfc.update(set(['Gene Simmons', 'Paul Stanley']))
print(mpfc)
#Kiss meets Monty Pythons Flying Circus?
#We're going to need more Scooby Snacks
#And a bigger boat
```

Output

```
{'Michael Palin', 'John Cleese', 'Eric Idle', 'Terry
Jones', 'Terry Gilliam', 'Graham Chapman'}
{'Carol Cleveland', 'Michael Palin', 'John Cleese', 'Eric
Idle', 'Terry Jones', 'Terry Gilliam', 'Graham Chapman'}
{'Michael Palin', 'John Cleese', 'Eric Idle', 'Terry
Jones', 'Terry Gilliam', 'Graham Chapman'}
{'John Cleese', 'Graham Chapman', 'Eric Idle', 'Terry
Jones', 'Terry Gilliam', 'Carol Cleveland', 'Michael
Palin'}
{'Gene Simmons', 'Michael Palin', 'John Cleese', 'Eric
Idle', 'Terry Jones', 'Terry Gilliam', 'Paul Stanley',
'Graham Chapman'}
```

Set Method Analogs for Set Operations

- Python 2.7 and Python 3.x introduce a series of set methods that are analogous to the set operations discussed earlier
- `set1.difference(set2)`; returns a set difference
- `set1.union(set2)`; returns a union of set1 and set2
- `set1.intersection(set2)`; returns a set intersection
- `set1.issubset(set2)`; returns true if set1 is a subset of set2
- `set1.issuperset(set2)`; returns true if set1 is a superset of set2
- `set1.symmetric_difference(set2)`; returns a new set with items from set1 or set2 but not both
- `set1.update(set2)`; updates set1 with the unique values from set2

Example: Set Operation Analog Methods

Example

```
#This is a rewrite of the previous example using methods
#Some Pythonistas consider the method interface to be more
#readable
#!/usr/bin/Python3
mpfc = set([ 'Graham Chapman','Eric Idle','Terry
Gilliam','Terry Jones','John Cleese','Michael Palin'])
fcw = set(['John Cleese','Michael Palin'])
fempython= set(['Carol Cleveland', 'Eric Idle'])

print(mpfc.difference(fcw)) # Difference
#{'Graham Chapman', 'Terry Jones', 'Eric Idle', 'Terry Gilliam'}

print(mpfc.union(fempython)) # Union
#{'John Cleese', 'Graham Chapman', 'Eric Idle', 'Terry Jones',
#'Terry Gilliam', 'Carol Cleveland', 'Michael Palin'}

print(mpfc.intersection(fempython))# Intersect
#{'Eric Idle'}

print(fcw.issubset(mpfc)) # Subset
#True

print(fempython.issuperset(mpfc)) # Superset
#False

print(mpfc.symmetric_difference(fcw)) #Symmetric Difference
#{'Graham Chapman', 'Terry Jones', 'Eric Idle', 'Terry Gilliam'}

print(
fempython.symmetric_difference(mpfc.symmetric_difference(fc
w))) # Symetric Difference
#{'Carol Cleveland', 'Terry Jones', 'Graham Chapman', 'Terry
Gilliam'}

mpfc.update(fempython) # Update
#{'Carol Cleveland', 'Michael Palin', 'John Cleese', 'Eric Idle',
'Terry Jones', 'Terry Gilliam', 'Graham Chapman'}
print(mpfc)
```

Module Summary

- **Dictionaries are mutable tables of object references**
  - Useful for creating data tables or records
- **Sets are mutable collections of immutable objects**
  - Useful for performing Mathematical or Database Operations
- **Dictionaries and sets both have a robust literal syntax for manipulation**
- **Dictionaries and sets both support a full set of methods for performing operations**
- **Frozensets are immutable, unique collections of immutable objects**

Module 7

Controlling the Flow of a Python Program

Module Goals and Objectives

Major Goals

Understand how to control the flow of a Python Program using flow control statements

Understand the syntax for Compound Statements

Use If statements for Conditional Execution

Use For Loops and While Loops for iterative execution

Specific Objectives

Understand compound statements

Use if, elif, and else statements to perform conditional processing

Write loop constructs using while and for

Use the break and continue statements to modify loops

Perform distributed assignments with the assignment operator

Augment loops with distributed assignments

Emulate C-style counting loops with for

Perform parallel sequence processing

Unpack sequences

Section 7–1

Flow Control Overview

About Flow Control

- **Often times it will be necessary in a program to control the order in which statements are executed**
- **Flow Control statements change the order in which statements are executed**
- **A body of code may need to be executed:**
  - If some Boolean condition is met
  - Repetitively until a Boolean condition is met
  - As a continuation of some other statement being executed
  - To execute a specific set of code that has been packaged as a unit
  - To stop the program and prevent any further execution
- **Python has a number of statements that can be used to control the flow of an application**
  - Conditional Statements modify the flow of an application based on the result of a Boolean expression
  - Loop Statements may be used to perform repetitive tasks
  - Python supports sequence-based assignment statements
  - Comprehensions create new Lists, Sets or Dictionaries by applying an expression to each item in another List, Set or Dictionary

Compound Statements

- **A number of Python's flow control statements are compound statements**
- **Compound statements are statements that have nested statements**
  - Nested statements are called a "Suite" or a "Block"
- **Compound statements are comprised of a Header and a nested Suite**

Syntax

```
Header :
    Nested Suite
```

  - Headers are followed by a colon operator (:)
  - Suites are blocks of code indented under the header
    - Suites are indented by 4 spaces by convention
    - Suites may be indented by any amount of space (4 spaces, tab) as long as the amount of indentation is consistent

- **All Compound statements follow the same pattern of a Header followed by a colon followed by a nested Suite**
- **Python uses the indentation of the suite to automatically detect the suite's boundaries**

Example: Compound Statement

Syntax

This is pseudo code to demonstrate the structure of a compound statement

```
if x :
     Some code attached to the if statement
     Some more code attached to the if statement
```

Syntax

For comparison purposes, the following statement written in (Perl, JavaScript, C, C#, etc.....) is the equivalent pseudo code to the Python statement in the previous example

```
if(x)
{
Some Code attached to the if statement
Some more code attached to the if statement
}
```

Syntax

The following examples are pseudo code for nested compound statements written in Python and written in a C-based programming language

```
if x:
     I am attached to the First if statement
     if y:
          I am attached to the Second if statement
     I am also attached to the First if statement
```

Syntax

```
if(x){
I am attached to the First if statement
     if(y){
     I am attached to the Second if statement
     }
I am also attached to the First if statement
}
```

Section 7–2

Conditional Statements

`if` Statements

- **`if` statements are Python's main statement for executing alternative branches of code based on the results of a test**
- **Any statements may be nested within an `if` statement**
- **When the `if` statement executes, a test is performed**
  - If the test result is true then Python executes an associated suite of nested statements
  - Nested statements are indented underneath the `if` header statement

Syntax

```
if test:
    Statement(s)
```

Example

```
usr = input("What is your name?")
if usr:
    print("Hello",usr,"! Welcome to Python!")
```

- **The `if` statement may be written on a single line if there is only 1 line of code in the suite**

Example

```
if usr: print("Hello",usr,"! Welcome to Python!")
```

`if/else` Statements

- **When an `if` statement executes, it performs a test and if that test is true then a nested statement is executed**
  - Otherwise, control is passed to the next line of execution in the program
- **`if` statements may have an optional `else` statement**
- **The `else` statement is a statement that executes if the `if` statements test fails**
- **This allows your `if` statement to execute two different branching paths predicated on the results of the if statement's test**

Syntax

```
if test:
    statement(s)
else:
    statement(s)
```

Example

```
usr = input("What is your name?")
if usr:
    print("Hello",usr,"! Welcome to Python!")
else:
    print("Why didn't anyone name you? Don't worry, Python
        will be your buddy from now on!")
```

Chained Conditionals with `if/elif/else`

- **When an `if` statement needs to branch in more than 2 directions, the `if` statement may be augmented with optional `elif` statements**
  - `elif` is short for else if
- **`elif` statements are executed if the `if` statement fails**
- **`elif` statements, like `if` statements, when executed perform a test**
  - If the results of the test are true then an attached suite of code is executed
  - Otherwise, control is passed to the next `elif`, `else` or to the next line of execution
- **`elif` statements are useful when an if statement test may have multiple cases**

Syntax

```
if testcase1:
    Suite
elif testcase2:
    Suite
elif testcase3:
    Suite
else:
    Suite
```

Example: `if/elif/else` Statements

Example

```
#!/usr/bin/Python3

score =
input("Input a score for the Quiz in the form of an
integer:")

if score.isdigit():
    score = int(score)
else:
    print(score, "isn't 'in the form of an integer' . ")
    quit()

if score >= 90:
    letter = 'A'
elif score >= 80:
    letter = 'B'
elif score >= 70:
    letter = 'C'
elif score >= 60:
    letter = 'D'
else:
    letter = 'F'



print("You received the letter grade",
letter, "for a score of", score)
```

Conditional Expressions

- **Python 2.5 introduced a new syntax for conditional expressions**

- **Like an `if/else` statement, a condition is evaluated**
  - If the condition is true then an expression is evaluated
  - Otherwise another expression is evaluated if the test result is false

Syntax

```
true_expr if condition else false_expr
```

- **The condition expression in the middle is evaluated first**
  - The `true_expr` expression is evaluated only if the condition was true
  - The `false_expr` expression is only evaluated when the condition is false

- **The conditional expression is often used to assign values for a common case or for an exception case**
  - This also means the expression may be read as "Common Case Expression" if the test is true, otherwise "Exception Case Expression"

Syntax

a is assigned object x if condition y is true. Otherwise a is assigned object z

```
a = x if y else z
```

Example: Conditional Expression

Example

```
quest = input("what is your quest?")

if quest:
    print(quest)
else:
    print("I have no quest")

print(quest if quest else "I have no quest")
```

Section 7–3

Assignment Statements

Assignment Statement Refresh

- **The assignment statement assigns an object to a name at runtime**
- **Assignments create object references**
- **Names are created when Objects are first assigned**
  - Names do not need to be pre-declared
- **Names must be assigned before being referenced**
  - Python returns an error if a name is used that does not have an assigned object

Syntax

```
name = object
```

- **The assignment operator is a highly overloaded operator that performs distributive assignments when used with sequence objects**
- **This gives the assignment operator the ability to behave like a loop when performing assignments**
  - This also means that the assignment operator may augment a loop through distributed assignments

Assignment Statement Syntaxes

- **The following table illustrates different assignment operator syntaxes**

- **Some of these syntaxes have been discussed in previous modules; the rest will be discussed in this section**

Assignment Operation	Description
`x = 'value'`	Standard Assignment Form
`x, y = 'value' , 'value'`	Tuple Unpacking Assignment by position
`[x , y] = ['value' , 'value']`	List Unpacking Assignment by position
`a, b, c, d, e = 'value'`	Generalized Sequence Assignment
`a, *b = 'value'`	Extended Sequence Unpacking(Python 3.x)
`x = y = z`	Multiple-Target Assignment
`x +=x`	Augmented Assignment

Sequence Assignments

- **Sequence assignments allow any sequence of names to be assigned any sequence of values**
  - For each item in the right hand value, an assignment is made to the positional correlated item in the left hand value
  - Python allows for multiple types of sequences to be used on both sides of the assignment operations
- **The number of values in the right value need to match the number of names in the left value**
  - Unless Extended Sequence unpacking is being performed
  - Python raises the `ValueError` exception if the wrong number of items are being unpacked
- **Tuple assignments**
  - When a tuple is coded on the left hand side of an assignment operator, Python pairs the objects on the right side with the targets on the left side by position
- **List assignments**
  - When a list is coded on the left hand side of an assignment operator, Python pairs the objects on the right side with the targets on the left side by position

Example: Sequence Assignment Techniques

Example

```
#!/usr/bin/Python3

fname, lname = 'Eric', 'Idle'    # Tuple Assignment
print(fname, lname)
# Eric Idle

fname, lname = lname, fname      # Tuple Swap Values
print(fname, lname)
#Idle Eric

s = 'John Cleese'
[fn, ln] = s.split()             # List Assignment
print(fn, ln)
#John Cleese

a,b,c,d,e = 'Terry'               # String unpacking to
Tuple
print(a,b,c,d,e)
# T e r r y

a,b,c,d = 'Terry'              # String unpacking to Tuple
print(a,b,c,d,e)
'''
Message  File Name    Line Position
Traceback
    <module>
    C:\HOTTPython\Mod08\Examples\sequenceUnpacking.py   16
ValueError: too many values to unpack (expected 4)
'''
#Uh oh.... Gotta fix that....
```

Extended Sequence Unpacking

- **While sequence types for assignments may be mixed, the sequences must be the same size for the assignments to work**

- **Python 3 supports an operator * that allows for more generalized unpacking**

Syntax

```
*name
```

- **A starred name can be used in the assignment target to collect all items in the sequence that are not assigned to other names**

- **Only one starred name may be used per assignment sequence**

Example

```
seq = 'Eric'
a, *b = seq       # a = e, b = ['r', 'i', 'c']
*a, b = seq       # a = ['E', 'r', 'i'], b = c
a, *b, c = seq    # a = e, b = ['r', 'i'] c = c
```

These same techniques may be emulated in Python 2.x with slicing syntax

```
seq = ['E','r','i','c']
a, b = seq[0],seq[1:]
print( a, b )     # a = E , b = ['r', 'i', 'c']
```

Section 7–4

Looping Statements

Loop Overview

- **Looping constructs are statements that repeat actions multiple times**
- **Repeated execution of statements is known as iteration**
- **Python supports 2 main looping constructs**
  - `while` Loop
  - `for` Loop
- **Looping constructs like the `for` loop may operate on any iterable object**
  - Objects in Python are considered iterable if the object is either a physically stored sequence or an object that produces one result at a time in the context of an iteration tool such as a `for` loop

`while` Loops

- **`while` loops are the most general iteration statement in Python**

- **`while` loops continually execute a block of code so long as a test condition is true**
  - When a `while` loop fails, control is passed to the next statement in the program

- **Any statements may be nested in a `while` loop**

syntax

```
while test:
    While Suite
```

Example

```
#!/usr/bin/Python3

x = 0 ; y = 10

while x < y:
    print(x, end = ' ')
    x = x + 1
```

while/else loops

- **`while` loops support an optional `else` statement that is executed if control exits the while loop normally**
  - If the while loop exits due to a break statement, then the else statement is not executed

syntax

```
while test:
    Suite
else:
    Suite
```

Example

```
#!/usr/bin/Python3

x = 1
y = 0
while x <= 10:
    print(x, end = ' ')
    y += x
    x+=1
else:
    print()
    print(y)
```

Output

```
1 2 3 4 5 6 7 8 9 10
55
```

The `break` Statement

- **The `break` statement causes an immediate exit from the loop**

- **Any code that follows the break statement in a loop is not executed if the break statement is executed**
  - So, a loop's else statement would not be executed if a break statement is encountered

Syntax

```
break
```

Example

```
#!/usr/bin/Python3

i = 1;
while i < 3:
    name = input("What is your name?")
    if name == "end": break
    age = input("What is your age?")
    if age == "end": break
    print("Hi", name, "you are", age , "Years old")
    i+=1
else:
    print(name, "next year you will be", int(age) + 1 ,
    "years old! " )
```

The `continue` Statement

- **The `continue` statement immediately jumps to the top of the loop and continues execution of the loop with the next iteration**

- **This is useful if you need to skip portions of loop code or nested loop statements**

Syntax

```
continue
```

Example

```
#!/usr/bin/Python3

x = 10
while x > 0:
    x -=1
    if x % 2 !=  0: continue
    print(x, end = ' ')
```

The `for` Loop

- **The `for` loop is a generic iterator**

- **The for loop steps through the items in any ordered sequence or any other iterable object**

Syntax

```
for item in object:
    Suite
else
    Suite
```

- **When Python executes the `for` loop, it assigns each element in the iterable `object` to `item` one by one and executes the loop suite for each item**

- **The optional `else` block in a for loop works the same way as the optional `else` block in a while loop**
  - The `else` statement is executed if the loop exits normally (i.e. without running into a `break` statement)

- **`for` loops in Python generally execute faster than `while` loops, so `for` loops are the first loop type that most Pythonistas use when writing loops**

Example: `for` Loop

Example

```
#!/usr/bin/Python3

mpfc = ['Graham Chapman' , 'Eric Idle' , 'Terry Gilliam',
'Terry Jones' , 'John Cleese' , 'Michael Palin']

for python in mpfc:
    print(python , end = ' ' )
    if python.endswith('Gilliam'):
        print('Giant foot smashes down!')
else:
    print()
    print('And now for something completely different!')
```

Output

```
Graham Chapman Eric Idle Terry Gilliam Giant foot smashes
down!
Terry Jones John Cleese Michael Palin
And now for something completely different!
```

Iterating through Sequences with `for` Loops

- **To iterate through a `list` with a `for` loop, simply use a `list` as the object argument in the for loop**

Example

```
#!/usr/bin/Python3
serenity = ['Malcolm "Mal" Reynolds', 'Zoe Washburne' ,
'Wash Washburne' , 'Inara Serra' , 'Jayne Cobb' ,
'Kaylee Frye', 'Dr. Simon Tam', 'River Tam', 'Shepherd']
for crew in serenity:
    print(crew , end =' ')
    if(crew.startswith('Malcolm')):
        print('I aim to  misbehave')
    if(crew.startswith('S')):print('The special hell')
    if(crew.startswith('Jayne')):
        print('The man they called Jayne')
#Malcolm "Mal" Reynolds I aim to misbehave
#Zoe Washburne Wash Washburne Inara Serra
#Jayne Cobb The man called Jayne
#Kaylee Frye Dr. Simon Tam River Tam Shepherd The special..
```

- **`Strings` and `tuples` may be used in the `for` loop in the same fashion**

Example

```
#!/usr/bin/Python3
l = ("I", "am" , "a" , "lumberjack", "and", "I'm", "okay")
s = "Lumberjack"
for word in l: print(word, end = ' ')
print()
for letter in s: print(letter , end = ' ')
# I am a lumberjack and I'm okay
# L u m b e r j a c k
```

Iterating through Dictionaries with `for` loops

- **When looping through a `dictionary`, a `for` loop returns each `key`, one `key` at a time from the dictionary**

Example

```
crew ={'Mal': 'Malcolm Reynolds',
        'Jayne': 'Jayne Cobb',
        'Shepherd': 'Derrial Book'}
for c in crew:
    print('Familiar name', c, 'Full name' , crew[c])
```

- **`Tuples` are useful when iterating through a dictionary using the dictionary `items()` method**
  - Recall, the `items()` method returns a tuple that contains a tied key value pair

Example

```
#using the same dictionary
for (Faname, Funame) in crew.items():
    print('Familiar name', Faname, 'Full name' , Funame)
```

Example: Nested Loops

In this example we are going to check to see which crew members are currently on the Serenity by comparing a list of required crew to a list of crew present onboard

Example

```
onboard = ['Mal', 'Zoe', 'Wash', 'Inara', 'Jayne',
             'Kaylee', 'Simon', 'River', 'Shepherd']
required = ['Mal', 'Zoe', 'River', 'Jayne']
found = []
numreq = len(required)
numfound = 0

for req in required:
    for crew in onboard:
        if req.lower == crew.lower:
            print(req, "was found")
            numfound +=1
            found.append(req)
            break
#Optionally
#for req in required:
#    if req in onboard:
#            print(req, "was found")
#            numfound +=1
#            found.append(req)

if numreq == numfound and found == required:
    print("Away team accounted for. Go find a Red shirt!")
#Mixing metaphors?
```

Output

```
Mal was found
Zoe was found
River was found
Jayne was found
Away team accounted for. Go find a Red shirt!
#Make it so, Number One
```

Using `for` Loops as a Counter

- **`for` loops in Python may be constructed to behave like standard C-style counting loops**
- **Python supplies a number of built-in functions that may be used to specialize the iteration of a for loop**
- **The `range()` function creates a series of successively increasing integers**
  - Useful to create a standard C-style counting loop
  - Also useful when indexes and an item's value are required
- **The `zip()` function returns a series of parallel item tuples**
  - Useful to traverse parallel sequences in for loops
- **The `enumerate()` function generates values and indexes of items in an iterable so manual counting is not necessary**

Using `range()` to create a Counter Loop

Syntax

```
range([start], stop [,step])
```

- **The `range()` function returns successive integers between the `start` and the `end` argument**
  - With 1 argument, `range()` returns `0` through `stop -1`
  - With 2 arguments, `range()` returns `start` through `stop -1`
  - With 3 arguments, `range()` returns `start` through `stop -1` counting by `step`
  - `range()` returns an immutable iterable sequence in Python 3.x and a list in Python 2.6/7

Example

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

Example

The following example uses the `range()` function and the `len()` function to perform manual indexing of a list

```
for i in range(len(onboard)):
    print("Crewmember", i , "is", onboard[i])
```

And for every other crewmember.

```
for i in range(0,len(onboard),2):
    print("Crewmember", i , "is", onboard[i])
```

Using the `zip()` Function to Traverse Parallel Sequences

Syntax

```
zip(iterable [,iterable]*)
```

- **The `zip()` function returns a series of tuples where the `i`th tuple contains the `i`th element from each of the iterables supplied as arguments**
  - Requires at least 1 tuple, or the function returns an empty result
  - The resulting tuple is truncated to the length of the shortest argument tuple

- **`zip()` makes parallel traversal of sequences fairly simple since `zip()` returns a tuple of the correlated items found at in each of its arguments**

Example

```
match = list(zip(('a','b','c'),(1,2,3)))

print(match)
[('a', 1), ('b', 2), ('c', 3)]
```

Example: Working with Sequences using `zip()`

Example

```
c1 = ['Mal',  'Wash', 'Jayne', 'Simon', 'Shepherd']
c2 = ['Inara', 'Zoe', 'Kaylee', 'River',]
c3 = ['Team 1','Team 2','Team 3','Team 4']

for (m1,m2,m3) in zip(c1, c2,c3):
    print(m1 , "is matched up with", m2, "in" , m3)
```

Output

```
Mal is matched up with Inara in Team 1
Wash is matched up with Zoe in Team 2
Jayne is matched up with Kaylee in Team 3
Simon is matched up with River in Team 4
```

Example

In this example we are going to create a dictionary with the zip() function

```
c1 = ['Mal',  'Wash', 'Jayne', 'Simon', 'Shepherd']
c2 = ['Inara' ,'Zoe' ,'Kaylee' ,'River',]

match = dict(zip(c1, c2))


for (m1, m2) in match.items():
    print(m1, 'is matched up with', m2)
```

Output

```
Wash is matched up with Zoe
Mal is matched up with Inara
Simon is matched up with River
Jayne is matched up with Kaylee
```

Working with offsets and items using `enumerate()`

syntax

```
enumerate(iterable, start =0)
```

- **The `enumerate()` function returns an iterable enumerate object**
- **The `enumerate()` function returns a tuple containing a count (from `start`) and the corresponding value obtained from the iterable**
- **`enumerate()` is useful for obtaining an indexed series when both the `offset` of an item and the `item` are required in `for` loops**

Example

```
c1 = ['Mal',  'Wash', 'Jayne', 'Simon', 'Shepherd']
pos = {0:'Captain', 1:'Pilot',2:'Muscle',
    3:'Doctor',4:'Spiritual Guidance'}
for (offset, crew) in enumerate(c1):
    print("The", offset , "crewmember is the" , crew, "and
    their position is the", pos[offset])
Output
The 0 crewmember is Mal and their position is the Captain
The 1 crewmember is Wash and their position is the Pilot
The 2 crewmember is Jayne and their position is the Muscle
The 3 crewmember is Simon and their position is the Doctor
The 4 crewmember is Shepherd and their position is the
Spiritual Guidance
```

Section 7–5

Comprehensions

Comprehensions Overview

- **Comprehensions are one of the most powerful tools in Python**
- **Comprehension syntax is derived from a construct in set theory notation that applies an operation to each item in a set**
  - Behaves similar to a `map()` function
  - List comprehensions are one of the most prominent contexts in which the iteration protocol is applied
- **Python 3.x supports Set comprehensions and Dictionary comprehensions as well as List comprehensions**

Benefits of List Comprehensions

- **List comprehensions are very concise to write**
  - This is quite fortunate, because the code pattern of building result lists in Python is very common
- **List comprehensions are used in many contexts, such as parsing lists and files**
- **List comprehensions can run much faster than `for` loop statements because their iterations are performed at C language speed inside of the Python interpreter**
  - List comprehensions can run approximately twice as fast as correlate `for` loops

List Comprehension Syntax

- **List comprehensions look similar to a "backwards" `for` loop**
- **List comprehensions are written in square brackets**
  - Ultimately, List comprehensions are an expression that constructs a new list
- **List comprehensions start with an arbitrary expression that uses a loop variable**
- **The expression is followed by a for loop header that creates the variable that is used in our expression and an iterable object**
  - An optional condition may exist in the list comprehension that may be used to filter the results of the list

Syntax

```
[item for item in iterable]

[expression for item in iterable]

[expression for item in iterable if condition]
```

Comparing List Comprehension syntax to for loops

- **The syntax**

```
l = [item for item in iterable]
```

- **Is roughly equivalent to**

```
l = []
for item in iterable
    l.append(item)
```

- **The Syntax**

```
l = [expression for item in iterable]
```

- **Is roughly equivalent to**

```
l = []
for item in iterable
    l.append(expression)
```

- **The Syntax**

```
[expression for item in iterable if condition]
```

- **Is roughly equivalent to**

```
l = []
for item in iterable:
    if condition:
        l.append(expression)
```

Example: For loops vs. List Comprehensions

Example

In this example we are going to print the items in a list using a for loop and a List Comprehension

```
mpfc =[ 'Graham Chapman','Eric Idle','Terry Gilliam',
        'Terry Jones','John Cleese','Michael Palin']

for py in mpfc:
    print(py , end = ' ')
print()

[print(py , end = ' ') for py in mpfc]


#Graham Chapman Eric Idle Terry Gilliam Terry Jones
#John Cleese Michael Palin
#Graham Chapman Eric Idle Terry Gilliam Terry Jones
#John Cleese Michael Palin
```

Example: For loop vs. List Comprehension to execute expressions

Example

In this simple example, we take a list of input strings and convert them to integers before appending them to a new list

```
istring= ['1', '5', '28', '131', '3']
onum = []
for num in istring:
    onum.append(int(num))
print(onum)

onum2 = [int(num) for num in istring]
print(onum2)

#[1, 5, 28, 131, 3]
#[1, 5, 28, 131, 3]
```

Example: For loop vs. List Comprehension with Filtering

Example

In this example we are going to use a for loop and a list comprehension to calculate all of the leap years between the year 2000 and 2080

```
leapfor = []
for year in range(2000, 2080):
    if (year % 4 == 0 and year % 100 != 0)
         or (year % 400 == 0):
        leapfor.append(year)

print(leapfor)

leapcomp = [y for y in range(2000, 2080)
if (y % 4 == 0 and y % 100 != 0) or (y % 400 == 0)]
print(leapcomp)
```

Output

```
#[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032,
#2036, 2040, 2044, 2048, 2052, 2056, 2060, 2064, 2068,
#2072, 2076]

#[2000, 2004, 2008, 2012, 2016, 2020, 2024, 2028, 2032,
#2036, 2040, 2044, 2048, 2052, 2056, 2060, 2064, 2068,
#2072, 2076]
```

The Iteration Protocol

- **Python's iteration protocol is a way that non-sequence items can take part in iteration loops**
- **The `for` loop uses the iteration protocol to step through items in the object across which it is iterating**
- **Supporting the iterable protocol simply means an object has an `__iter__` method that returns another object that supports the iterator protocol**
  - The for loop calls the object's `__next__` method (run by the `next()` built-in) on each iteration and catches the `StopIterationexception` to determine when to stop looping
- **Any object that supports this model works in a `for` loop and in other iteration contexts such as `Comprehensions`, the `map()` function**
- **File Objects have an iterator that automatically reads one line at a time in a for loop, list comprehension, or other iteration context**

Example: Using Comprehensions with File Iterators

Example

In this example we read through the contents of one of the previous examples, converting the contents to uppercase. We also remove some extra newline characters.

```
[print(line.upper()) for line in open('listcompitem.py')]

[print(line.rstrip().upper()) for line in
open('listcompitem.py')]

#Output 1
MPFC =[ 'GRAHAM CHAPMAN','ERIC IDLE','TERRY GILLIAM',

        'TERRY JONES','JOHN CLEESE','MICHAEL PALIN']



FOR PY IN MPFC:

    PRINT(PY , END = ' ')
.....
#Output 2
MPFC =[ 'GRAHAM CHAPMAN','ERIC IDLE','TERRY GILLIAM',
        'TERRY JONES','JOHN CLEESE','MICHAEL PALIN']

FOR PY IN MPFC:
    PRINT(PY , END = ' ')
```

Example: Search For Files, Using Comprehensions

Example

In this simple example we are going to use a list comprehension to output the results of a search for Python files

```
import os, glob
f = glob.glob('*.py')
print(f)
for file in f:
    print(os.path.realpath(file))

[print(os.path.realpath(f)) for f in glob.glob('*.py')]

#['globlistcomp.py', 'listcompexpr.py', 'listcompfile.py',
#'listcompitem.py', 'listcompleapyear.py']

# Removed Duplicate Results
#C:\HOTTPython\FlowControl\Comprehensions\Examples\globlistcomp.py
#C:\HOTTPython\FlowControl\Comprehensions\Examples\listcompexpr.py
#C:\HOTTPython\FlowControl\Comprehensions\Examples\listcompfile.py
#C:\HOTTPython\FlowControl\Comprehensions\Examples\listcompitem.py
#C:\HOTTPython\FlowControl\Comprehensions\Examples\listcompleapyear.py

#Prints .py files with a size greater than 500 bytes
[print(f) for f in glob.glob('*.py') if os.stat(f).st_size > 500]

#Prints file size and real path for .py files
[print(os.stat(f).st_size, os.path.realpath(f)) for f in glob.glob('*.py')]
```

Dictionary Comprehensions

- **A dictionary comprehension is an expression and a loop with an optional condition enclosed in braces, very similar to a List comprehension**

- **Dictionary comprehensions are like List comprehensions except they create a Dictionary Object**

- **Dictionary comprehensions are useful for initializing dictionaries from a keys list**

Syntax

```
{k: val for k in list}
{k:expr for k in list}
{k: for k in list if condition}
{k:v for k ,v in iterable}
{k:v for key , value in iterable if condition}
```

Example

This example flips key/value pairs

```
orig = {'a': 1, 'b': 2, 'c': 3}
flop = {value : key for key, value in orig.items()}
[print(k , v) for k , v in flop.items()]
1 a
2 b
3 c
```

Example: Dictionary Comprehension

Example

In this example we are going to create a Dictionary of filenames and file sizes from the current working directory. The file name will be the key and the file size of the file will be the value. We will then use a list comprehension to print the contents of the Dictionary object we create

```
import os
fs = {name: os.path.getsize(name) for name in
os.listdir(".")}

[print(k , v) for k , v in fs.items()]

#Output
listcompitem.py 370
dictcompos.py 140
listcompfile.py 127
systemComprehensions.py 267
listcompleapyear.py 526
globlistcomp.py 637
listcompexpr.py 210
```

Set Comprehensions

- **In addition to creating sets by calling `set()`, or by using a set literal, set can also be created using set comprehensions**

- **Set comprehension syntax looks a lot like Dictionary comprehension syntax except we aren't working with key / value pairs**

- **Set comprehensions run a loop and collect the result of an expression on each iteration; a loop variable gives access to the current iteration value for use in the collection expression**
  - The result is a new set created by running the code, with all the normal set behavior

Syntax

```
{expression for item in iterable}
{expression for item in iterable if condition}
```

Example

```
{print(x * x, end = ' ' ) for x in range(10)}
# 0 1 4 9 16 25 36 49 64 81
```

Example: Use Set Comprehensions to Return Unique Results

Example

In this example we are going to use a Set Comprehension to return a distinct list of Science Fiction authors from a namedtuple object

```
from collections import namedtuple
bookinfo = namedtuple("bookinfo", "author title genre")
books = [
bookinfo("Heinlein", "Starship Troopers", "scifi"),
bookinfo("Heinlein", "Stranger in a Strange Land",
"scifi"),
bookinfo("Gibson", "Neuromancer", "scifi"),
bookinfo("Gibson", "Mona Lisa Overdrive", "scifi"),
bookinfo("Herbert", "Dune", "scifi"),
bookinfo("McCarthy", "Blood Meridian", "western"),
bookinfo("McCarthy", "No Country For Old Men", "thriller"),
bookinfo("Follett", "Eye Of The needle", "suspense"),
bookinfo("Huxley", "Brave New World", "scifi"),
]

scifi_author = {
b.author for b in books if b.genre == 'scifi'}
print(scifi_author)


#{'Heinlein', 'Herbert', 'Gibson', 'Huxley'}
```

Module Summary

- **`If` statements may be used to perform conditional processing in Python programs**
- **`For` statements and `While` statements may be used to perform repetitive processing**
- **The assignment operator may be used to perform distributed assignments**
  - Sequences may be unpacked into other sequences
- **The `for` loop iterates through an object by default**
- **`enumerate()`, `zip()` and `range()` may be used to augment how for loops execute**
- **Statements may be nested within other statements**
- **Comprehensions may be used to process any iterable object**
- **List comprehensions are the most common type of comprehension**
- **Python 3 supports set comprehensions and dictionary comprehensions**

Module 8

Creating modular Code with Functions

Module Goals and Objectives

Major Goals

Create functions and use functions

Understand variable scope

Use lambda expressions

Specific Objectives

On completion of this module, students will be able to:

Define reusable, flexible programs using functions

Pass arguments to functions by position, by name, and by unpacking

Encapsulate names in a function's namespace by using local variables

Share values between scopes using global and non local

Understand the basic building blocks of a module

Define and use anonymous functions using lambda expressions

Section 8–1

Overview of Functions

Function Overview

- **Functions are a sequence of statements and/or expressions that are bundled as a single callable unit and perform tasks or return values**

- **Functions are the most basic program structure that Python provides for facilitating code reuse**

- **Functions are commonly used to**
  - Execute frequently used code
  - Organize code for readability and ease of maintenance
  - Perform specific tasks
  - Split programs into procedures that have well-defined roles
  - Minimize redundancy in code

- **Functions may be treated like operators or subroutines**
  - Like operators, functions may return new values
  - Like subroutines, functions may simply run tasks and return None

Section 8–2

Creating Functions

Creating Functions

- **Functions are created with the `def` statement**
- **The `def` statement is a runtime statement**
- **When `def` runs, `def` creates a new function and assigns the function to a name**
- **Because `def` is a statement, `def` may appear anywhere a statement may be used**
  - This means that functions may be dynamically created
- **Typically, functions are defined in a module and are created when the module is imported into the calling script**
- **`def` is a compound statement, so a `def` statement is comprised of a Header and a Body**

The `def` statement

Syntax

```
def name([arg, ...argN]):
    Suite
```

- **The def statement header specifies a function name that is assigned the function object**
- **Arguments are passed to functions by assignment**
- **Each call to a function creates a new, local scope, where assigned names are local to the function call by default**
- **After a function is created, the function may be called by using a calling statement**
  - A function is called by adding parenthesis to your function name anywhere in your program

Syntax

```
name()
```

Examples: Defining and Calling Functions

Example

This first example creates a simple function that outputs a string. The function is then called in a conditional context

```
def nickoftime():
    print("Big Damn Heroes , Sir!")
mal = True
zoe = True
if mal and zoe:
    nickoftime()
```

Example

This example takes 2 arguments, iterates through the arguments and adds up their values, and then prints the result

```
def sum(a, b):
    result = 0
    for i in range(a, b + 1):
        result += i
    print(result)

print("The sum of 1 to 10 is")
sum(1,10)
print("The sum of 1 to 100 is")
sum(1,100)
print("The sum of 1 to 9001")
sum(1,9001)
```

Returning Values from a Function

- **Functions may be designed to return a value**

- **To return a value from a function use the `return` statement**

Syntax

```
return [expression]
```

- **The `return` statement exits the enclosing function and returns an `expression` value as the result of the call to the function**
  - If the `expression` is omitted then the default return value is `None`
  - Tuples are often used to return multiple values from a function

- **Functions without `return` statements also return `None` by default**

Example

```
def multiply(a , b):
    return a * b
print(multiply(2,4))
print(multiply('Steak', 3))
```

Example: Returning a Value from a Function

Example

We can redefine the sum function from earlier to return the result of the calculation, instead of printing the result

```
def sum(a, b):
    result = 0
    for i in range(a, b + 1):
        result += i
    return result

print("The sum of 1 to 10 is" , sum(1,10) )
print("The sum of 1 to 100 is" , sum(1,100) )
print("The sum of 1 to 9001" , sum(1,9001) )
```

Example: Returning Multiple Values from a Function

Example

In this example we will redefine an earlier example that used nested loops to perform an intersection. We will define the intersect logic in a function that takes multiple sequences as arguments and returns a sequence

```
def searchcrew(s1,s2):
    result = []
    for val in s1:
        if val in s2:
            result.append(val)

    return result

onboard = ['Mal', 'Zoe', 'Wash', 'Inara', 'Jayne',
'Kaylee', 'Simon', 'River', 'Shepherd']

required = ['Mal', 'Zoe', 'River', 'Jayne']

found = []
numreq = len(required)


found = searchcrew(required,onboard)
numfound = len(found)

if numreq == numfound and found == required:
   print("Away team accounted for. Go find a Red shirt!")
```

Output

```
Away team accounted for. Go find a Red shirt!
```

Documenting a Function

- **Documenting a function is useful for any programmer who intends on using a function**

- **Documentation in a function is generated by creating a docstring**
  - Programmers may look up the function's documentation using the `help()` function or the function's `__doc__` attribute

- **A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition**

- **All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings**
  - PEP 257 suggests that for consistency, always use """triple double quotes""" around docstrings

- **docstrings may be single line or multi-line**

Syntax

This is an example of a single line docstring

```
def test():
""" Do test and return a list """
```

Writing docstrings

- **docstrings may be single line docstrings or multi-line docstrings**

- **A single line docstring is a phrase ending in a period that describes the function or method's effect as a command i.e. (Do this, Return that)**

- **Multi-Line docstrings consist of a summary line just like a one-line docstring**
  - They are then followed by a blank line, followed by a more elaborate description.
  - The summary line may be used by automatic indexing tools; it is important that it fits on one line and is separated from the rest of the docstring by a blank line

Syntax

```
def test(list, iter):
    """Perform iterative tests on a list.

    Arguments:
    list -- Any list, optionally any sequence
    iter -- Number of iterations for the test
    """
print(help(test))

#test(list, iter)
#    Perform iterative tests on a list.
#
#    Arguments:
#    list -- Any list, optionally any sequence
#    iter -- Number of iterations for the test
#
#None
```

The `pass` Statement

- **The `pass` statement is used to create a nop or noop (No Operation) instruction**

- **The pass statement is useful to test the syntax of a statement or to create a placeholder for the definition for a function before the definition has been created**

- **The pass statement is also useful to create a placeholder for a class statement before the class has been defined**

Syntax

```
pass
```

Usage

```
def test():
    pass
```

Example: Pass Statement

Example

In this example we are going to use a `pass` statement to allow us to prototype the header for the `searchcrew()` function we defined in the slide titled "Example: Returning Multiple Values from a Function" . The function will eventually accept 2 lists for comparison and will return a list of values that are common to both lists.

```
def searchcrew(foo,bar):
    pass

onboard = ['Mal', 'Zoe', 'Wash', 'Inara', 'Jayne',
'Kaylee', 'Simon', 'River', 'Shepherd']
required = ['Mal', 'Zoe', 'River', 'Jayne']
found = []

numreq = len(required)
numfound = 0

#Refactor this code into the searchcrew function
for req in required:
    for crew in onboard:
        if req.lower == crew.lower:
            print(req, "was found")
            numfound +=1
            found.append(req)
            break

#new caller
#found = searchcrew(foo,bar)

if numreq == numfound and found == required:
    print("Away team accounted for. Go find a Red shirt!")
```

Section 8–3

Arguments

Passing Arguments to Functions

- **Objects may be passed to a function as inputs by assigning the objects to arguments in the function**

- **Arguments for a function are defined as part of the function's header**
  - Each call to a function object generates a new, local scope, where assigned names are local to the function call by default

Syntax

```
def functionname(arg,... arg = value, ... *args, ...
**kwargs): Suite
```

- **Functions may be defined with a number of different argument signatures**
  - Signatures include
    - Positional signatures
    - Keyword signatures
    - Positional/keyword/unpacking sequences

- **Arguments are passed to the function by the calling statement as**
  - Positional arguments
  - Keyword arguments
  - Positional/keyword packing/unpacking sequences

Positional Arguments

- **To create a function with positional arguments, supply a comma separated tuple of names as part of the function's header**

Syntax

```
def functionname(x, y,...z):
    Suite
```

- **The function's calling statement supplies positional arguments to the function**

Syntax

```
functionname(1,2)
```

- **Passing the wrong number of argument will raise a `TypeError` exception**

Example: Positional Arguments

Example

This example demonstrates how to call a function with positional arguments. The example also demonstrates the exceptions that are thrown by using the wrong number of arguments

```
def greet(name, greeting):
    """ Greet a user by name """
    print(greeting, name , '!')

greet('Program', 'Greetings')
```

Output

```
Greetings Program !

#greet('Program')
#Message File Name    Line Position
#Traceback
#    <module>
#   C:\HOTTPython\Examples\positionalArguments.py  6
#TypeError: greet() missing 1 required positional argument:
#'greeting'

#greet('Program', 'Greetings', 'Identity Disc')
#Message File Name    Line Position
#Traceback
#    <module>
#   C:\HOTTPython\Examples\positionalArguments.py  11
#TypeError: greet() takes 2 positional arguments but 3 were
#given
```

Keyword Arguments

- **While positional arguments are matched from left to right, Python also allows arguments to be passed by name**
- **Keyword arguments are created by placing key=value pairs into the functions arguments list**
- **Arguments are then matched in the call statement by name**
  - This allows for the passing of values by name, not position
  - The passing of arguments by name is a large convenience because the caller does not have to be aware of argument positions
  - This also allows changes to the functions interface to occur without the fear of breaking positional calling statements
- **Assigning values to arguments also has 2 other benefits**
  - A default value for the argument is created
  - The argument is made optional
- **Positional and named arguments may be combined**
  - By placing optional arguments at the end of the arguments list, a signature of "required" and "optional" arguments may be created

Syntax

```
def functionname(x = val , y = val2, ...): Suite
def functionname(x , y , z = ''): Suite
```

Example: Keyword Arguments

Example

In this example we augment the greet function with a default name and a default greeting. We also create a division function that returns a quotient of half the dividend by default

```
def greet(name = 'Program', greeting = 'Greetings'):
    """ Greet a user by name """
    print(greeting, name, '!')

greet()
# Greetings Program !
greet('Student')
# Greetings Student !
greet(name = 'Mal', greeting = 'No Misbehaving')
# No Misbehaving Mal !

#dividend is a required argument

def div(dividend, divisor = 2):
    """Returns the quotient of a dividend and a divisor.
    Divisor defaults to 2"""

    quotient = dividend / divisor
    return quotient

x = div(10,2)
y = div(3)
print(x,y)
#5.0 1.5
```

Variable Positional Arguments

- **Functions may be defined to collect an arbitrary tuple of arguments**
- **This allows for the development of functions that may take a variable number of arguments**
- **Functions may be defined to take variable arguments by supplying a starred name as one of the arguments `*name`**
- **By supplying a single starred name, all positional arguments are collected into a tuple**
- **By supplying a starred name at the end of an arguments list, the remaining positional arguments may be collected into a tuple**
- **Python 3.x also allows for keyword arguments to be created after a starred name**

Syntax

```
def functionname(*args):
    Suite
```

Syntax

```
def functionname(x, y,*args):
    Suite
```

Unpacking arguments in the calling statement

- **Calling statements may unpack all objects in an iterable as individual arguments by passing a starred name**

- **This allows the caller to pass an arbitrary iterable of arguments into a function**

Syntax

```
functionname(*args)
```

Syntax

```
functionname(x , y , *args)
```

Example: Variable Positional Arguments

Example

In this example we are going to create a simple function to add up all the items in a sequence and return the result

```
def add(x,y,*z):
    result = x + y
    if(z):
        for num in z:
            result += num

    return result

nums = (1,2,3,4,5)
msg = ('And', 'finally,', 'monsieur,', 'a', 'wafer-thin',
'mint')

a = add(2,2)
b = add(2,2,3,4)
c = add(*nums)        # Argument Unpacking
d = add(10,20,*nums) # Argument Unpacking
msgres = add(*msg)    # Argument Unpacking


print(a, b, c, d)
#4 11 15 45

print(msgres)
#Andfinally,monsieur,awafer-thinmint
```

Variable Keyword Arguments

- **.Functions may be defined to collect an arbitrary dictionary of arguments**
- **This allows for the development of functions that may take a variable number of key = value pair arguments**
- **Functions may be defined to take variable arguments by supplying a double-starred dictionary name as one of the arguments `**name`**
- **By supplying a double-starred dictionary name, all keyword arguments are collected into the dictionary**
- **By supplying a double-starred dictionary name at the end of an arguments list, the remaining keyword arguments may be collected into the dictionary**

Syntax

```
def functionname(**kwargs):
    Suite
```

Syntax

```
def functionname(x, y,**kwargs):
    Suite
```

Unpacking Keyword Arguments in the Calling statement

- **Calling statements may unpack all objects in a dictionary as key/value arguments by passing a double-starred dictionary**

- **This allows the caller to pass an arbitrary dictionary of arguments into a function**

Syntax

```
functionname(**kwargs)
```

Syntax

```
functionname(x , y , **kwargs)
```

Example: Variable Keyword Arguments

Example

In this example we will pass an arbitrary tuple of attributes to a function which will perform some light data transformations and return a dictionary of the transformed attributes

```
def userdata(**usr):
    print(usr)
    data = {}
    for (key, value) in usr.items():
        data[key] = value.upper()
    return data

ud = {'name' : 'Kaylee', 'pos' : 'Mechanic', 'phrase'
:"Everything's shiny, Cap'n. Not to fret"}

u1 = userdata(name = 'Mal', pos = 'Captain', phrase = 'I
aim to misbehave')
u2 = userdata(name = 'Wash', pos = 'Pilot', phrase = 'I am
a leaf on the wind. Watch how I soar.')
u3 = userdata(**ud)


print([val for val in u1.values()])
#['CAPTAIN', 'I AIM TO MISBEHAVE', 'MAL']

print([val for val in u2.values()])
#['PILOT', 'I AM A LEAF ON THE WIND. WATCH HOW I SOAR.',
#'WASH']

print([val for val in u3.values()])
#["EVERYTHING'S SHINY, CAP'N. NOT TO FRET", 'MECHANIC',
#'KAYLEE']
```

Section 8–4

Scope

Scope Basics

- **When a name is created or used in a Python program, Python creates or looks up the name in a namespace**
- **A namespace is just a place to collect names**
- **In Python, the term "scope" refers to a namespace—the location of a name's assignment in a program**
- **Functions add an extra namespace layer to your program**
  - This minimizes the potential for namespace collisions among variables that have the same name
- **By default all names assigned inside a function are associated with that function's namespace**
  - This means that all variables assigned in a function are local by default

Syntax

```
def functionname():
    x = 'local'
    y = 'also local'
```

Lexical Scoping Rules

- **Variable scopes are determined at runtime based on the location of variable assignments in a program**
  - This is said to be lexical scoping because the object's scope is dependent on the context of assignment
- **The concept of lexical scoping creates different scopes on top of the Python __builtins__ scope**
  - local
  - nonlocal
  - global
- **If a name is assigned inside a `def`, then the variable is local to the function**
  - Names assigned within a `def` may only be seen within that def
  - Names assigned within a `def` do not clash with names assigned outside the def
- **If a variable is assigned in an enclosing `def`, then the variable is nonlocal to nested functions**
  - The `nonlocal` keyword is new to Python 3.x
- **If a variable is assigned outside of all `def` statements, then the variable is global to the entire file**

Example: Local Scope

Example

In this example we are going to take a look at how to create local variables and use variables from the global space

```
x = 7 # Name x is assigned in the global space
y = 12 # Name y is assigned in the global space

def add():
    result = x + y # Name result is assigned locally
    return result

z=add()
print(z) # 19

def add2(x , y):
    # Names x and y are local to the function
    result = x + y
    return result

a = add2(2,2)
print(a) # 4

def add3():
    x = 4 # Name x is local to add3 and masks global x
    y = 3 # Name y is local to add3 and masks global y
    result = x + y
    return result

b = add3()
print(b) #  7

print(x , y) # Names x and y are still 7 and 12
```

Global Scope

- **Any variables not assigned in a `def` statement are global in scope**
- **It may be necessary to assign a value to a global variable from a function call**
  - Recall that any assignment to a variable in a `def` statement creates a local variable
- **Python supports namespace declaration statements that allow a function to work with variables outside the local scope**
- **The `global` statement allows a function to refer and change names in the global space**

The `global` statement

- **The `global` statement is a namespace declaration**
- **When the `global` statement is used inside of a class or a function, all appearances of a name in the class or function context will be treated as references to a name in the global context**
  - This reference occurs whether the name exists or doesn't exist and whether the name has a value assigned or not
- **The `global` statement allows global variables to be changed within a function or a class**

Syntax

```
global name [,name]*
```

Example: Global variables

Example

This example squares the value of x once executed

```
x = 5
def square():
    global x
    x *= x

print(x)# 5
square()
print(x)# 25

x = 1
def increase():
    global x
    x = x + 1
print(x) # Displays 1
increase()
print(x) # Displays 2
```

The `nonlocal` statement

- **The `nonlocal` statement is a new statement in Python 3.x**
- **The nonlocal statement is a namespace declaration**
- **The nonlocal statement causes all appearances of name in a context to refer to a local variable, of the same name in an enclosing function's scope**
  - This reference occurs whether name is assigned or not
- **Unlike `global`, the name must exist in the enclosing function**
- **Some uses for nonlocal variables are to allow a nested function to alter variables in an outer function or to persist state information across calls to nested functions**

Syntax

```
nonlocal name [,name]*
```

Example: nonlocal variables

Example

This is a simple counter function. In this example an outer function defines 2 child functions and uses the child functions to decrement a nonlocal variable and display the countdown message

```
def countdown(start = 10):
    n = start
    def display():
        print('T-minus %d' % n)
    def decrement():
        nonlocal n # Bind to outer n (Python 3.x only)
        n -= 1
    while n > 0:
        display()
        decrement()

countdown()

#T-minus 10
#T-minus 9
#T-minus 8
#T-minus 7
#T-minus 6
#T-minus 5
#T-minus 4
#T-minus 3
#T-minus 2
#T-minus 1
```

The `locals()` function and the `globals()` function

- **Python supports functions that may be used to return a dictionary of variables**
  - locals()
  - globals()

- **The `locals()` function returns a dictionary containing the local variables of the caller in a key : value format**

Syntax

```
locals()
locvar = locals()
```

- **The `globals()` function returns a dictionary containing the caller's global variables (i.e. the enclosing module's names)**

Syntax

```
globals()
globvar = globals()
```

Example: `locals()` and `globals()`

Example

This example demonstrates the use of the `locals()` function and the `globals()` function

```
x = 7
y = 12

def add():
    result = x + y
    print('Local variables in add() are', locals())
    return result
print(add())
#Local variables in add() are {'result': 19} 19

def add2(x,y):
    result = x + y
    print('Local variables in add2() are', locals())
    return result
print(add2(2,2))
#Local variables in add2() are {'x': 2, 'y': 2, 'result':
#4} 4

def add3():
    x = 4
    y = 3
    result = x + y
    print('Local variables in add3() are', locals())
    print('Global variables are', globals())
    return result
print(add3())
#Local variables in add3() are {'x': 4, 'y': 3, 'result':
#7} 7

#Global variables are {'__loader__': <class
#'_frozen_importlib.BuiltinImporter'>,
#'__package__': None, 'add3': <function add3 at
#0x000000000562E158>, 'x': 7, 'y': 12,
#'__name__': '__main__', '__file__':
#'C:\\HOTTPython\\Examples\\globalsAndLocals.py',
#'__builtins__': <module 'builtins' (built-in)>, '__doc__': None,
#'add2': <function add2 at 0x000000000562E0D0>, 'add': <function
#add at 0x000000000562E048>}
```

Section 8–5

Anonymous Functions

Anonymous Functions Overview

- **Python provides an expression that generates function objects**
- **This expression that creates anonymous functions is known as a `lambda` expression**
- **`lambda` expressions are useful because as expressions they may appear anywhere in Python expressions**
  - In that regard, this makes the `lambda` expression much more flexible than a `def` statement
- **`lambdas` are designed to return a new function object which may or may not be assigned to a name**
  - This is in contrast to the `def` statement which always assigns the new function to the name in the header of the `def` statement
- **A `lambda` expression's body is a single expression, not a suite of statements**
  - This means that lambdas are useful for specialized, dynamic expressions or specialized dynamic functions, opposed to the `lambda's` sibling `def` statement which is tuned for general use
- **`lambdas` are especially useful as a shorthand notation that allows functions to be embedded in the code that uses the `lambdas`**

Creating lambda Expressions

- **To create a lambda expression, use the lambda keyword, followed by an arguments list, followed by an expression that uses the arguments**

Syntax

```
lambda arg , arg , ... : expression
```

- **The `lambda` keyword creates a new function object and returns the object to be called later**
- **The arguments list is equivalent to the arguments list in the `def` statements header**
- **The expression is the implied return value of the subsequent calls to the `lambda`**
- **Because `lambdas` are a single expression, they are not designed to be used for complex calculations**
  - Use a `def` statement for complex functions
- **However, because `lambdas` are expressions, they may be used where a `def` statement cannot, such as**
  - The value of a key in a dictionary
  - Part of the argument list for a function call

Example: lambda Expressions

Example

In this example we are going to use lambda expressions in various contexts

Here we are going to make some basic arithmetic functions dynamically

```
add = lambda x , y : x + y
sub = lambda x , y : x - y
addresult= add(2,2)
subresult = sub(10,3)
print(addresult, subresult) # 4, 7
```

Here we are going to create a jump table or a dispatch table which is a list of functions to be executed on demand. I've heard of late binding, but, late..late binding?!?!

```
res = 0
powtable = [lambda x: x ** 0,
            lambda x: x ** 1,
            lambda x: x ** 2,
            lambda x: x ** 3,
            lambda x: x ** 4,
            lambda x: x ** 5,
            lambda x: x ** 6,
            lambda x: x ** 7,
            ]
print(powtable[2](5)) # 25
for p in powtable:
    print(p(2))
    res += p(2)
#1 2 4 8 16 32 64 128
print(res) # 255
```

And lastly, we'll create a Dictionary of functions. Because that is how we do it.

```
mymath = {
        add:lambda x , y : x + y,
        sub:lambda x , y : x - y
        }
print(mymath[add](2,2), mymath[sub](2,2)) # 4 0
```

Module Summary

- **Functions are defined at runtime using the `def` statement**
- **All function calls have their own local namespace**
- **Variables may be shared across namespaces by using `global` and `nonlocal`**
- **Functions may be passed named and positional arguments**
- **Functions may be created with default (optional) arguments**
- **Calling statements may pass named, positional and unpacked arguments**
- **lambda expressions are used to create flexible, simple anonymous functions**
- **Docstrings may be used to document a function or a class**

Module 9

Input / Output

Module Goals and Objectives

Major Goals

Understand how to Read and Write Files

Understand how to work with the File System using Python

Understand how to parse XML files

Specific Objectives

On completion of this module, students will be able to:

Read Files from the File System

Write Files to the File System

Search for Files

Work with Directories

Use Modules to interact with the System

Read XML files

Parse XML files

Write XML files

Section 9–1

Input / Output Overview

Input / Output Overview

- **Python supports a number of facilities for working with various types of I/O**
- **The 3 main types of I/O that Python works with are**
  - text I/O
  - binary I/O
  - raw I/O
- **Any object that belongs to any of these categories of I/O is known as a `file object`**
  - Other common terms for file objects are *`stream object`* and *`file-like object`*
- **The primary module that provides Python's main facilities for dealing with I/O objects is the `io` module**
- **The most convenient method for interacting with the `io` module is to create file objects using the `open()` function**
- **The `open()` function creates a `file object`**

I/O Types: Text I/O and Binary I/O

- **Text I/O uses and creates `str` objects**
  - Encoding and Decoding is performed transparently
  - Optionally, platform-specific newline characters may be translated

Example

The following examples use the `open()` function to create a file object and the `io` module's `StringIO` class to create an in-memory text stream

```
from io import *
myfile = open('myfile.ext' , 'r', encoding = 'utf-8')
mytxtstream = StringIO('This is in memory text')
```

- **Binary I/O uses and creates `bytes` objects**
  - Binary I/O does not perform any encoding, decoding or newline translation
  - Useful for manipulating non-text data or for manually controlling text transformation

Example

The following example uses the `open()` function and the `io` module's `BytesIO` class to create a binary file object

```
from io import *
mybinfile = open('HOTTLogo.jpg', 'rb')
mybinfile2 = BytesIO(b"This is binary data \x00\x01")
```

I/O Types: Raw I/O

- **Raw I/O is also known as unbuffered I/O**
  - Typically used as a low-level building block for binary and text streams
  - Rarely used to manipulate raw streams from user code
  - Raw streams are created by opening files in binary mode with buffering disabled

Example

This example creates a Raw file object by opening a file in an unbuffered binary mode

```
myrawfile = open('HOTTLogo.jpg', 'rb', buffering = 0)
```

Section 9–2

Working with `file` Objects

Creating `file` Objects

- **The most convenient way to create `file` objects is by using the built in `open()` function**
  - In Python 2.x, the `file()` function is a synonym for the `open()` function
  - In Python 3.x the `file()` function is not available
- **`file` Objects export methods to work with the file object and attributes that may be used to work with the file object's metadata**

Syntax

```
open(path
     [,mode = 'r'
     [,buffering = None
     [,encoding = None     # text mode only
     [,errors = None       # text mode only
     [,newline = None      # text mode only
     ]]]]])
```

- **The `open()` function returns a new `file` object connected to the external file named by the path argument**
- **The path argument is typically a string or a bytes object that gives the path to the file that is to be opened**
- **`open()` raises an IOError if the `open()` function fails**
  - In Python3.3, IOError is an alias of OSError

File Modes Table

- **The `open()` function's `mode` argument is an optional string that specifies the mode in which the file is opened**

Mode	Description
`r`	Opens a file in read mode. File is opened for reading only. This is the default file mode
`w`	Opens a file in write mode. Creates file if file does not exist, overwrites file contents if file does exist.
`a`	Opens a file in append mode. Creates file if file does not exist, appends new content to existing content.
`x`	Opens file for exclusive creation, fails if the file already exists
`b`	Opens file in binary mode
`t`	Opens file in text mode (default)
`+`	Opens file for reading and writing
`U`	Universal newlines mode (for backwards compatibility; should not be used in new code)

- **The default file mode is `'r'` for read mode**

Optional File Modes

- **The `t` (text) and `b` (binary) modes may be combined with `r, w, a` to specify Text I/O or Binary I/O**

Mode	Description
`rt,rb`	Read Text mode (default), Read Binary mode
`wt,wb`	Write Text mode, Write Binary mode
`at,ab`	Append Text mode, Append Binary mode

- **The `+` mode for reading and writing may be combined with `r, w, a` as well**

Mode	Description
`r+t,r+b`	Read/Write Text mode , Read/Write Binary mode
`w+t,w+b`	Write/Read Text mode (truncates), Write/Read Binary mode (truncates)
`a+t,a+b`	Append/Read Text mode, Append/Read Binary mode

`open()` arguments: `buffering` and `encoding`

- **`buffering`**
  - `buffering` takes an integer argument to set the buffering policy of the `file` object
  - By default full buffering is on
    - `0` turns buffering off (binary mode only)
    - `1` sets line buffering
    - `> 1` sets full buffering

Example

```
pic = open('pic.jpg' ,'rb', buffering = 0)
```

- **`encoding`**
  - Sets the encoding type to be used for encoding or decoding data
    - Should be used in text mode only
  - Default encoding is based off of the default encoding of the platform running the Python program
  - The documentation for the `codecs` module has a list of supported encodings
    - `http://docs.python.org/3.3/library/codecs.html`

Example

```
mpfc = open('mpfc.txt' ,'r', encoding = 'utf8')
```

`open()` arguments: `errors` and `newline`

- **`errors`**
  - The errors argument specifies how to handle encoding and decoding errors
    - Pass `'strict'` to raise a ValueError exception if there is an encoding error (the default of `None` has the same effect)
    - Pass `'ignore'` to ignore errors. (Note that ignoring encoding errors can lead to data loss.)
    - Pass `'replace'` to create a replacement marker (such as `'?'`) to be inserted where there is malformed data

example

```
mpfc = open('mpfc.txt' ,'a', encoding = 'utf8', errors =
'strict')
```

- **`newline`**
  - Controls how Universal newlines work (text mode only)
    - Options are `None` (default), `''` , `'\n'` , `'\r'`, `'\r\n'`
  - On input, if `newline` is `None`, universal newlines are enabled
    - Lines that end in `'\n'` , `'\r'`, `'\r\n'` are all translated to `'\n'`
    - If `newline` is `''`, universal newlines are enabled, but line endings are returned to the caller untranslated
  - On output, if `newline` is `None`, any `'\n'` characters are translated to the system default specified by `os.linesep`
  - If `newline` is `''`, no translation takes place

Examples: Creating `File` Objects

Example

In this example we are going to create file objects in various read / write modes

```
mpfc = open('mpfc.txt', 'r')

mpMovies = open('mpMovies.csv' , 'w')

mpPlayers = open('mpfc.txt', 'a')

ffPlayers = open('ffplayers.txt', 'a', encoding = 'utf8')
```

`file` Object Attributes

- **`file` Objects have a few attributes that may be used to expose metadata about the file**

- **`file.closed`**
  - Returns true if the file object has been closed

- **`file.mode`**
  - Returns the mode string passed to the open function

- **`file.name`**
  - Returns a string name of the corresponding external file

- **`file.encoding`**
  - returns the encoding type of the file object

Example

```
mpfc = open('mpfc.txt', 'r')
print(mpfc.name)      #mpfc.txt
print(mpfc.mode)      #r
print(mpfc.closed)    #False
print(mpfc.encoding)  #cp1252
```

Closing `file` Objects

- **To close a `file` object use the `close()` method**

Syntax

```
file.close()
```

- **`close()` flushes and closes the file stream**
  - The `close()` method has no effect if the file is already closed
  - Once the file is closed, any operation on the file (i.e. reading or writing) will raise a ValueError
- **As a convenience, it is allowable to call this method more than once; only the first call, however, will have an effect**
- **The file object's `closed` attribute will return true if the file has been closed**

Example

```
mpfc = open('mpfc.txt', 'r')
mpfc.close()
print(mpfc.closed)    #True
```

Reading from Files

- **Python supports a number of methods and techniques that may be used to read from a file**
- **The `read()` method allows for an entire file or a specified number of bytes to be read from a file**
- **The `readline()` method reads a line from a file**
- **The `readlines()` method reads an entire file into a list**
- **Files are their own iterator objects, so files may also be processed in `for` loops**

The `read()` method

- **The `read()` method may be used to read an entire file or to read a specified number of bytes from a file**

Syntax

```
file.read()
file.read(n)
```

- **To read an entire file at once, use the `read()` method with no arguments**
  - In text mode, line endings are translated to `'\n'` by default
  - In binary mode, the result string may contain non-printable characters
  - In Python 3.x text mode decodes to a Unicode `str` object and binary mode returns unaltered content as a `bytes` object
  - In Python 2.x content read is always represented as a `str` object

- **To read a specified number of bytes, pass an integer as an argument to the `read()` method**

Example

```
mpfc = open('mpfc.txt', 'r')
x = mpfc.read()
mpfc.close()
print(x)
```

Example: Reading from a file

Example

In this example we are going to write a function to read an entire file or to read the specified number of bytes. Some light error handling with `if` statements will be added to the function. We will replace the if statements with exception handling later in the course

```
import os

def rf(path, b = None):
    if(os.path.isfile(path)):
        f = open(path, 'r')
        if(b is None):
            content = f.read()
        elif(isinstance(b, int)):
            content = f.read(b)
        else:
            content = 'Could not read file'
    else:
        content = path + ' is not a file'

    return content

i = rf('mpfc.txt')    #Read file
print(i)

ib = rf('mpfc.txt',7) #Read 7 bytes from file
print(ib)

ib = rf('mpfc.txt','Steve')   #Read Steve bytes from file?
print(ib)

iyota = rf('toyotathon.txt')  #Is it toyotathon all ready?
print(iyota)
```

Reading a file with `readline()` and `readlines()`

- **The `readline()` method reads the next line of a file**
  - Returns an empty string once end-of-file (EOF) is reached

Syntax

```
file.readline()
```

Example

```
mpfc = open('mpfc.txt', 'r')
print(mpfc.readline())
# Graham Chapman
```

- **The `readlines()` method reads an entire file into a list**
  - Each element in the list is one line from the file

Syntax

```
file.readlines()
```

Example

```
mpfc = open('mpfc.txt', 'r')
print(mpfc.readlines())
#['Graham Chapman\n', 'John Cleese\n', 'Terry Gilliam\n',
#'Eric Idle\n', 'Terry Jones\n', 'Michael Palin']
```

Example: `readlines()`

Example

In this example we create a simple function that reads a file into a list, formats the items in the list and appends the transformed items to a new list which is returned to the caller

```
def modfile(path):
    mod = []
    f = open(path,'r')
    pathlist = f.readlines()

    for item in pathlist:
        item = item.rstrip()
        mod.append(item.upper())

    return mod

modlist = modfile('mpfc.txt')
print(modlist)
```

Output

```
['GRAHAM CHAPMAN', 'JOHN CLEESE', 'TERRY GILLIAM', 'ERIC
IDLE', 'TERRY JONES', 'MICHAEL PALIN']
```

Example: `readline()`

Example

In this example we create a simple function that reads the contents of a file one line at a time in a `while` loop, formats the items, and appends the transformed items to a new list which is returned to the caller

```
def modfile(path):
    mod = []
    f = open(path,'r')
    line = f.readline()

    while(line != ''):
        line = line.rstrip()
        mod.append(line.lower())
        line = f.readline()

    return mod

modlist = modfile('mpfc.txt')
print(modlist)
```

Output

```
['graham chapman', 'john cleese', 'terry gilliam', 'eric
idle', 'terry jones', 'michael palin']
```

File Iterators

- **A `file` object is its own iterator, and a `for` loop may use the built-in iterator to step through the lines in a file automatically**

- **When a file is used as an iterator, typically in a for loop the iterator object's `next()` method is called repeatedly**
  - This method returns the next input line, or raises `StopIteration` when `EOF` is hit

- **Files may be used in other iteration contexts such as comprehensions**

Example

```
for line in inputfile:
Suite
```

Example

```
[line for line in open(path)]
```

- **The iterator syntax `for line in inputfile:` is similar to `for line in file.readlines()`**
  - The iterator version is more efficient because it does not load the entire file into memory

Example: File Iterator

Example

In this example we process the contents of a file in a for loop and in a list comprehension

```
f = open('mpfc.txt', 'r')
for line in f:
    print(line.strip())
f.close()
```

Output

```
Graham Chapman
John Cleese
Terry Gilliam
Eric Idle
Terry Jones
Michael Palin
```

Example

```
print([line for line in open('mpfc.txt', 'r')])
```

Output

```
['Graham Chapman\n', 'John Cleese\n', 'Terry Gilliam\n',
'Eric Idle\n', 'Terry Jones\n', 'Michael Palin']
```

Writing to a File

- **The `write()` method and the `writelines()` method may be used to write contents to a file**

- **The `write()` method writes a string to a file**

Syntax

```
file.write(s)
```

  - In text mode `'\n'` is translated into the platform-specific line end sequence by default
  - In binary mode, the string may contain non-printable bytes
  - In Python 3.x, text mode encodes `str` Unicode strings, and binary mode writes `bytes` strings unaltered

- **The `writelines()` method writes all strings in a list onto a file**

Syntax

```
writelines(l)
```

Example: Writing to Files

Example

```
f = open('mpfc.txt', 'a')
f.write('Carol Cleveland \n')
f.close()
f = open('mpfc.txt', 'r')
print(f.read())
```

Output

```
Graham Chapman
John Cleese
Terry Gilliam
Eric Idle
Terry Jones
Michael Palin
Carol Cleveland
```

Example

```
crew = ['Mal', 'Zoe', 'Wash', 'Inara', 'Jayne', 'Kaylee',
'Simon', 'River' ,'Shepherd']
f = open('crew.txt','w')
f.writelines([c + '\n' for c in crew])
f.close()

#crew.txt
Mal
Zoe
Wash
Inara
Jayne
Kaylee
Simon
River
Shepherd
```

Section 9–3

The `os` and `os.path` modules

About the `os` and `os.path` modules

- **Often it is necessary to work with the file systems underlying a Python program in a platform-independent way**
- **The `os` module is the primary operating system interface in Python**
- **The `os` module has generic operating system support and provides a standard platform-independent operating system interface**
- **The `os` module includes tools to work with**
  - Environments
  - Processes
  - Files
  - Shell Commands
- **The `os.path` module provides useful functions for working with pathnames**
- **Generally, programs that use `os` and `os.path` are portable across most platforms**
  - However, not every operating system supports all methods exported by `os` and `os.path`

os Module Portability Constants

- **The os module automatically sets a number of constant values to the appropriate values used by the platform the Python program is being executed on**

- **These constants are used internally to help build and parse portable directory and search path strings**

Constant	Meaning
os.curdir	This is the string used to represent the current working directory on an OS (e.g. . on Windows and Posix, : on Mac)
os.pardir	This is the string used to represent the parent directory on an OS (e.g. .. on Windows and Posix, :: on Mac)
os.sep	This is the string used to separate directories (e.g. / on Unix, \ on Windows, : for Mac)
os.altsep	This is the alternative separator string or None is returned (e.g. / for Windows)
os.extsep	This is the character that separates a base file name from the file extension (e.g. .)
os.pathsep	This is the character that is used to separate search path components in the PATH and PYTHONPATH shell variable settings (e.g. ; for Windows and : for Unix)
os.defpath	This is the Default search path used by os.exec calls if there is no PATH setting in the shell
os.linesep	This is the string used to terminate lines on the current platform. (e.g. \n for Posix, \r for Mac, or \r\n for Windows)

Example: os Module Portability Constants

Example

This example lists out the values set for the os Module's portability constants on the current platform

```
import os
print(os.name)
print(os.getcwd())

print('Current Dir representation', os.curdir)
print('Parent Dir representation', os.pardir)
print('Directory Separator', os.sep)
print('Alternate Directory Separator', os.altsep)
print('Extension Separator', os.extsep)
print('Search Path Separator', os.pathsep)
print('Default Search Path', os.defpath)
print('Line Terminator', os.linesep)
```

Output

```
nt
C:\HOTTPython\SampleCode\IO
Current Dir representation .
Parent Dir representation ..
Directory Separator \
Alternate Directory Separator /
Extension Separator .
Search Path Separator ;
Default Search Path .;C:\bin
Line Terminator
```

Getting and Changing Directories

- **The `os` module includes a number of methods processing files by their pathnames**
  - The `os.path` module extends this functionality

- **`os.name`**
  - Returns the name of the OS specific modules whose names are copied to the top level of the os namespace (i.e. posix, nt, mac, etc...)
    - To return a string identifying the OS that Python is running on, use `sys.platform`

- **`os.getcwd()`**
  - Returns the current working directory name as a string

Syntax

```
os.getcwd()
```

- **`os.chdir()`**
  - Changes the current working directory for the current process to a new path
    - Successive file operations are relative to the new current working directory

Syntax

```
os.chdir(path)
```

Example: Getting and Changing Directories

Example

This simple example displays the Platform from the sys module, the name of the OS specific modules exported by the os module, reports the current working directory, changes directories and reports the current working directory

```
import os
import sys

print(sys.platform)
print(os.name)
print(os.getcwd())

os.chdir('..')
print(os.getcwd())
```

Win32 Output

```
win32
nt
C:\HOTTPython\SampleCode\IO
C:\HOTTPython\SampleCode
```

Cygwin Output

```
cygwin
posix
/home/HOTT/HOTTPython/IO
/home/HOTT/HOTTPython
```

Listing Directory Contents

- **The `os.listdir()` function returns a list of all entries in a directory path**

- **`os.listdir()` is a fast and portable alternative to using the glob module which is useful for filename expansion**

Syntax

```
os.listdir(path)
```

Example

```
import os

#List contents of Current Working Directory
for file in os.listdir(os.getcwd()):
    print(file)
```

Searching for Files with `glob`

- **Often times, it is useful to allow users to use wildcard characters to perform searches for files in the file system**

- **The Python `glob` module finds all pathnames matching a specified pattern according to rules used by the Unix shell**

- **While glob does not support tilde ~ expansion, the `glob` module does support the use of wild card characters such as `*`, `?` and `[]`**

Syntax

```
import glob
glob.glob(pathname)
```

- **The `glob` module's `glob()` method returns a list of pathnames that match the pathname argument**
  - Pathname takes absolute and relative pathnames
  - The `glob()` method may return an empty list if no pathnames are matched

Wildcard	Meaning
*	Matches any character 0 or more times
?	Matches any single Character
[...]	Matches any single character listed in the brackets
[!...]	Matches any single character not listed in the brackets

Example: Searching for Files with Glob

Example

This example explores searching for files relatively using `glob.glob()` and wildcards

```
import glob

efiles = glob.glob('e*.*')
# List files that start with e and have an extension
print(efiles)
#['Encoding.py', 'errors.py']

ext = glob.glob('*.??t')
#List files that have a 3 character extension ending with a
# t
print(ext)
#['crew.txt', 'ffplayers.txt', 'firefly.txt', 'mpfc.txt']

ogfiles = glob.glob('[og]*')
# List files that start with an o or a g
print(ogfiles)
# ['globing.py', 'open.py', 'openClose.py', 'OsBasics.py',
# 'osListDir.py', 'osPortabilityConstants.py']

nogfiles = glob.glob('[!og]*')
# List files that do not start with an o or a g
print(nogfiles)
#['appendFile.py', ... 'firefly.txt',
#'iteratorComprenehsion.py', 'mpfc.txt', 'mpMovies.csv',
#'pic.jpg', ... 'writeLines.py', '__pycache__']
```

Manipulating paths with `os.path`

- **The `os` module's `os.path` module provides some powerful methods for manipulating and testing path statements**

- **The `os.path.join()` method intelligently (using the operating systems path separator) joins one or more path components to create a path**
  - This is useful when paths need to be created at runtime from string-based data

Syntax

```
os.path.join(pathcomponent1 [,pathcom2 [,...]])
```

- **The `os.path.split(path)` method returns a tuple comprised of 2 components: a head and a tail (head,tail)**
  - The tail component is the last pathname component
  - The head component is everything leading up to the tail
    - Similar to `(os.path.dirname(path), os.path.basename(path))`

- **`os.path.basename(path)` returns the last path component of a path**

- **`os.path.dirname(path)` returns everything leading up to the last path component of a path**

Testing Path Components

- **The `os.path` module provides a number of methods that may be used to test path components**

- **`os.path.isfile(path)`**
  - Returns true if a string path is a regular file

- **`os.path.isabs(path)`**
  - Returns true if a string path is an absolute path

- **`os.path.isdir(path)`**
  - Returns true if a string path is a directory

- **`os.path.islink(path)`**
  - Returns true is a string path is a symbolic link

- **`os.path.ismount(path)`**
  - Returns true if string path is a mount point

Example

```
import os

print(os.path.isdir(os.getcwd())) #True
```

Creating and Removing Directories

- **The `os` module provides functions to create and remove directories**

- **`os.mkdir(path [,mode])`**
  - Makes a directory called `path` with the optionally provided `mode`
    - Default mode is 777

- **`os.makedirs(path, [mode])`**
  - This recursive directory function makes a directory like `os.mkdir()` but also creates all intermediate directories necessary to create the leaf directory
    - Throws an exception if the leaf directory already exists or cannot be created

- **`os.rmdir(path)`**
  - Removes a directory named by the path argument

- **`os.removedirs(path)`**
  - Recursive directory removal function, similar to `os.rmdir()`, but if the leaf path is successfully removed, `removedirs()` will continue to remove directories corresponding to the right most path statements until the whole path is removed or an error is raised
    - Throws an exception if the leaf directory could not be removed

Example: Working with Directories

Example

```
import os

def createdir(base, new):
    newdir = os.path.join(base,new)
    os.mkdir(newdir)
    if os.path.isdir(newdir):
        print('Directory Successfully Created')
        return newdir
    else:
        print('Unsuccessful Creating Directory')
        return False

nd = createdir('c:\HOTTPython', 'Test')
# Directory Successfully Created
head,tail = os.path.split(nd)
f = 'test.txt'
if os.path.isdir(nd):
    os.chdir(nd)
    t = open(f, 'a')
    t.write('Making Files \n')
    t.close()

print(os.listdir(os.getcwd()))
# ['test.txt']
for file in open(f):
    print(file)
#Making Files
os.unlink(f)
print(os.listdir(os.getcwd()))
#[]
os.chdir('..')
print(os.listdir(os.getcwd()))
#['Mod01', ... 'Mod17', 'Test']

os.rmdir(tail)
print(os.listdir(os.getcwd()))
##['Mod01', ... 'Mod17']
```

Module Summary

- **The `open()` function may be used to read and write files to the file system**
- **The os module provides generic Operating System support and a standard platform independent Operating System Interface**
- **The os.path module provides additional file directory pathname related services and additional portability tools**

Module 10

Object-Oriented Programming with Python

Module Goals and Objectives

Major Goals

Understand Object-Oriented Programming

Understand Pythons Object-Oriented Programming Paradigm

Create Object-Oriented Python Programs

Specific Objectives

On completion of this module, students will be able to:

Reuse Code

Create Classes in Python

Instantiate Classes in Python

Create and Use Class Attributes

Create and Use Instance Methods

Implement Inheritance in Python

Overload Built-in Operators

Destroy Objects

Section 10–1

Object-Oriented Programming Overview

Object-Oriented Programming

- **Object-Oriented Programming allows developers to organize their programs around interactions of objects**
  - In Python, everything is an object
- **Objects encapsulate a state and a set of behaviors**
- **A class provides the definition of the structure and the behaviors of objects**
- **Classes are Python's main object-oriented programming tool**
- **Classes are used to create and manage new objects that support inheritance and overloading**

Benefits of OOP

- **Object-Oriented Programming provides many benefits**

- **Code Reuse**
  - Objects created using an OOP may be easily reused in other programs

- **Encapsulation**
  - Once an object is created, it is not necessary to know how the object is implemented in order to use the object
  - Objects may also hide attributes from developers
    - This prevents developers from altering implementation code that should not be altered

- **Maintenance**
  - OOP provides an excellent framework for developing modular code libraries
  - Libraries of code may be modified and adapted as necessary by developers

- **Extensibility**
  - OOP methodologies make it simple to create new objects from existing objects
    - This gives the programmer the ability to use existing behaviors, extend existing behaviors or build new behaviors

OOP Terminology

- **Class**
  - A class is a definition of an object
  - Defines the attributes and behaviors of an object
- **Object**
  - A member of a class
    - Attributes and behaviors are stipulated by the Class
  - Everything in Python is an object
- **Instantiation**
  - The process of creating an object from a class
- **Instance**
  - A specific object member of a class
  - Each object is an instance of a class
- **Class attribute**
  - An attribute that is shared by all instances of a class
- **Instance attribute**
  - Attributes that are attached to a specific instance

OOP Terminology

- **Inheritance**
  - The ability to transfer the attributes and behaviors of one class to another class
- **Encapsulation**
  - Hiding implementation of attributes and behaviors inside of a class from the program that is using the class
- **Polymorphism**
  - The ability for a behavior to react differently to different types of input
- **Abstraction**
  - The ability to model complex systems as smaller more generalized systems specific to problem domains
  - Closely related to the concept of Encapsulation

Section 10–2

Object-Oriented Programming in Python

Objects in Python

- **In Python, data takes the form of Objects**
  - Objects may be built-in objects
  - Objects may be user defined objects
- **Developers may create custom objects by:**
  - Creating and using Classes
  - External language tools e.g. C extension libraries
- **Essentially, objects are custom data types that have custom values and associated sets of operations**
- **Objects are instances of classes**
- **Everything in Python is an object**

Example: Object types in Python

Example

In this example we are going to use the `type()` method to return the type objects associated with each object. The syntax `type(object)` is equivalent to the syntax `object.__class__`

```
#!/usr/bin/python

import sys
import sqlite3


def test(): pass

conn = sqlite3.connect(":memory:")

print(type(1))
print(type(""))
print(type([]))
print(type({}))
print(type(()))
print(type(object))
print(type(test))
print(type(sys))
print(type(conn))
```

Output

```
<class 'int'>
<class 'str'>
<class 'list'>
<class 'dict'>
<class 'tuple'>
<class 'type'>
<class 'function'>
<class 'module'>
<class 'sqlite3.Connection'>
```

Object-Oriented Programming in Python

- **Python's class mechanism allows users to create classes with only slight additions to the Python language syntax and semantics**
  - Python's class mechanisms are built from a mixture C++ and Modula-3
- **Python classes provide all standard features of Object-Oriented Programming**
  - Inheritance and Multiple Inheritance (Inheritance)
  - Method overriding in derived classes (Polymorphism)
- **In Python, classes are Data Types**
- **All of the Data Types built into Python are classes**
- **Classes defined by a developer are custom Data Types**
  - Classes are blueprints for custom Data Types
  - Classes are used to define behaviors and attributes of objects

Creating Classes

- **The `class` statement creates new class objects**
- **According to PEP 8, Classes should be named using Camel Case Syntax**
  - Attributes of classes are not capitalized
- **To create an instance of a class, assign the class to a new variable**
- **Classes automatically inherit from the built-in `Object` class**

Syntax

```
class ClassName(superclass [,superclass]*):
    suite
```

Example

```
#Python 3.x
class Test():
    pass
```

Example

```
class Test():
    pass

x = Test()
```

Example: Creating a Basic Class

Example

```
#!/usr/bin/python

class Test:
    pass

myobj = Test()

print(type(myobj))
print(type(Test))
print(dir(Test))
```

Output

```
<class '__main__.Test'>
<class 'type'>
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

The Class suite

- **The `class` statement introduces new syntax and semantics to the developer, and allows the developer to use features of Python's object system**

Syntax

```
[decoration]
class name [(super [,super]*)]:
    suite
```

- **`class` statements create new class objects**
  - Classes are factories for instance objects
  - Encapsulate attributes and methods for objects
- **The `class` statement creates a class object and assigns it to a name**
- **New classes inherit from each super class in the order supplied as arguments to the class**
- **`Class` statements introduce a new local name scope**
  - This creates a namespace to encapsulate attributes
- **Assignments inside `class` statements create class attributes**
  - All names assigned in the class statement generate `class` object attributes shared by all instances of the class

Class Statement Behavior

- **While the `class` statement may seem similar to class statements in other languages (e.g. C++) , the `class` statement in Python behaves differently from other programming languages**

- **The `class` statement in Python behaves a lot like the `def` statement that is used for creating functions**

- **The `class` statement is an object builder and an implicit assignment**
  - When executed, the `class` statement creates a class object and stores a reference to the class object by using the name supplied in the `class` statement header

- **Classes are built at runtime**
  - Classes do not exist until Python reaches the `class` statement in a program
  - At that point, Python runs the `class` statement and defines the class

Example: Class Attributes

Example

```
#!/usr/bin/python

class Alert:
     msg = "Hello"

a1 = Alert()
a2 = Alert()


print(a1.msg)
print(a2.msg)
```

Output

```
Hello
Hello
```

Creating and Working with Methods

- **Functions are an important concept to understand when dealing with methods**

- **Methods are function objects created by `def` statements that are nested in a `class` statement**

- **Methods are essential to the implementation of Encapsulation in the Object-Oriented Programming paradigm**
  - Abstractly, methods provide behaviors for instance methods to inherit
  - Programmatically, methods work the same way as functions

- **One key exception between functions and methods is that a method's first argument always receives the instance object that made the method call**
  - Python automatically maps instance method calls to a class's method functions
  - Python determines the appropriate class by using its inheritance search procedure

Example

```
instance.method(args,...)
#Is translated to
class.method(instance, args....)
```

The `self` argument

- **The first argument is one of the most significant differences between normal functions and class methods**
  - The first argument of class method calls is assigned a reference to the instance that made the method call
- **By convention, the first argument for class methods is called `self`**
  - It is strongly recommended to adhere to this convention, but technically not necessary
  - Remember it is the first argument of a class method that has significance, not the name of the first argument
- **The self argument provides a hook back to the instance that is the subject of the method call**
  - This provides classes with the ability to manage the data that varies between different instances

Syntax

```
class ClassName():
    def mymethod(self,arg1, args....)
        self.arg1 = arg1
```

The Explicit Nature of the `self` Argument

- **In Python, the `self` argument is always explicit in the code**
- **Methods must always use the self argument to:**
  - Set attributes of the instance
  - Get attributes of the instance
- **The explicit nature of the `self` argument exists by design**
- **The presence of the `self` argument makes it obvious to the developer that they are working with an instance and instance attribute names**
  - As opposed to names in the local or global scopes

Example: Creating and Working with Instance Methods

Example

In this example we are going to create a class with a single instance method. Once the class is instantiated, the instance method may be called with an argument. The instance method will print the argument passed to the screen

```
#!/usr/bin/python

class Alert:
    def message(self, msg):
        self.msg = msg
        print(self.msg)

a = Alert()
a.message('Hello, World')

a2 = Alert()
a2.message('And now for something completely different')
```

Output

```
Hello, World
And now for something completely different
```

Quick Review of Classes and Instances

- **Class Objects**
  - The class statement creates a class and assigns is to a name
  - Class objects support 2 kinds of operations
    - Attribute References
    - Instantiation
  - Assignments within class statements create class attributes
    - Class attributes are inheritable
  - Class methods are nested def statements with a special first argument that receives a reference to the subject instance
- **Instance Objects**
  - Generated from Classes
  - To create an instance object, call a class object like a function
  - Each instance inherits class attributes
    - Has its own attribute namespace
  - Assignments to attributes of the first argument in methods create instance attributes
    - e.g. `self.var = 'value'`

Customizing Instance State with `__init__()`

- **Class instantiation uses function notation**
  - Simply call the class as a function and assign the results to a new object

Example

```
x = TestClass()
```

- **The instantiation operation creates a new empty object**
- **It is useful to instantiate an object that has been customized with a specific initial state**
  - This is useful because objects may be predefined with custom:
    * Attributes
    * Behaviors
- **Python supports a built-in method for this very purpose**
  - __init__()
- **When a class defines an `__init__()` method, instantiating the class automatically executes the `__init__()` method for the new instance**

The `__init__()` Method

Syntax

```
__init__(self [,arg]*)
```

- **The `__init__()` method is used to initialize the attributes of an object**
  - Is invoked on `class(args...)`
- **`__init__()` is the constructor that initializes the new instance, `self`**
  - When run for class calls, the `self` argument is provided automatically
  - The `arg` arguments are the arguments passed to the `class` at instantiation
- **`__init__()` must return no value**

Example: Initializing an Instance with `__init__()`

Example

```
#!/usr/bin/python

class Person:
    def __init__(self, name):
        self.name = name
        print(self.name)

p1 = Person('Gary')
p2 = Person('Roland')
```

Output

```
Gary
Roland
```

Example: Initialized Instance Method

Example

```
#!/usr/bin/python

class Person:
    def __init__(self, name):
        self.name = name


    def print_name(self):
        print(self.name)

p1 = Person('Gary')
p2 = Person('Roland')
p1.print_name()
p2.print_name()
```

Output

```
Gary
Roland
```

Working with Class Attributes in `__init__()`

- **Class attributes may also be assigned to in the `__init__()` method**

- **To assign to class attributes, explicitly refer to the class name and the class**

Syntax

```
def __init__():
    ClassName.class_attribute  = 'x'
```

Example: Working with Class Attributes

Example

```
#!/usr/bin/python

class Person:
    pnum = 0
    def __init__(self, name):
        self.name = name
        Person.pnum +=1


    def print_name(self):
        print(self.name)

p1 = Person('Gary')
print(Person.pnum)

p2 = Person('Roland')
print(Person.pnum)

p1.print_name()
p2.print_name()
```

Output

```
1
2
Gary
Roland
```

Accessing Attributes Using `delattr()` and `getattr()`

- **While it is not recommended, a developer may use a function oriented interface for interacting with an objects attributes**
  - This style is useful for testing, but not for production code

- **`delattr(object, 'name')`**
  - This method deletes the attribute `'name'` from an object
  - Similar to the `del obj.name` call
    - e.g. `delattr(x, 'y')` is similar to `del(x.y)`

- **`getattr(object, 'name'[, default])`**
  - Returns the value of the `'name'` attribute from the object
  - Similar to `obj.name`
    - e.g. `getattr(x,'y')` is similar to `x.y`
  - If the `'name'` attribute does not exist, the default value (if provided) will be returned

Accessing Attributes Using `hasattr()` and `setattr()`

- **`hasattr(object, 'name')`**
  - Returns `true` if `object` has an attribute called `'name'`
  - Returns `false` otherwise
- **`setattr(object, 'name', value)`**
  - Assigns the `value` attribute to the attribute `'name'`
  - Similar to object.name = value
  - e.g. setattr(x, ' y' , z) is similar to x.y = z

Example: Accessing Attributes Using a Function Oriented Interface

Example

```
#!/usr/bin/python

class Person:
    pnum = 0
    def __init__(self, name):
        self.name = name
        Person.pnum +=1
    def print_name(self):
        print(self.name)


p1 = Person('Gary')

print(hasattr(p1, 'name'))
print(getattr(p1, 'name'))
print(getattr(p1, 'position', 'instructor'))
setattr(p1, 'position', 'Instructor')
print(getattr(p1,'position'))
delattr(p1,'position')
```

Output

```
True
Gary
instructor
Instructor
```

Garbage Collection

- **Python has a feature called garbage collection that cleans up unused memory as your program runs**
- **Python uses a reference-based garbage collection system**
  - Python uses a counter to keep track of the number of references currently pointing to an object
  - Once an object's counter reaches 0, memory space is reclaimed
- **In standard CPython (Python) space is reclaimed immediately as soon as the last reference to an object is removed**
- **The automatic garbage collection system allows a developer to use objects without having to allocate or free up space manually**

Destructors

- **The counterpart to the class `__init__()` method is the `__del__()` method**

- **The `__del__()` method is a destructor method**
  - It is called automatically when an instance's memory space is being reclaimed
  - Participates as part of the Python garbage collection process

Syntax

```
class ClassName():
    def __del__():
        #Manual Cleanup Here
```

- **Because Python performs automatic garbage collection, it is important to point out that destructors are not commonly used in Python programs**

Example: Garbage Collection Using a Destructor

Example

```
#!/usr/bin/python

class Person:
    pnum = 0
    def __init__(self, name):
        self.name = name
        Person.pnum +=1
    def print_name(self):
        print(self.name)
    def __del__(self):
        Person.pnum -= 1
        print(self.name, "Removed")

p1 = Person('Gary')
print(Person.pnum)
p2 = Person('Roland')
print(Person.pnum)
p1.print_name()
p2.print_name()
del(p1)
print(Person.pnum)
del(p2)
print(Person.pnum)
```

Output

```
1
2
Gary
Roland
Gary Removed
1
Roland Removed
0
```

Section 10–3

Implementing Inheritance

Inheritance Overview

- **Software design using an object-oriented paradigm focuses on objects and operations on objects**
  - This combines the power of the procedural paradigm with a dimension that integrates data with operations into objects
- **Inheritance extends the power of an object-oriented paradigm by adding the ability to define generalized classes with attributes and methods that will be used by other classes**
- **The inheritance mechanism may be used to create a class that uses, specializes or modifies the behavior of an existing class**
- **The class that provides the initial set of attributes is known as the *base class*, *parent class* or *superclass***
- **The class that inherits from the base class is known as a *derived class*, *child class* or *subclass***
- **Derived classes may use any inherited attributes, redefine any inherited attributes or add new attributes of its own**

Using Class Inheritance

- **To use inheritance in Python, simply supply the name of the base class to the new derived class as an argument of the `class()` method**

Syntax

```
class DerivedClass(BaseClass):
    Code
```

- **The base class must be defined in a scope containing the derived class definition**

- **Arbitrary expressions are allowed in the place of the `BaseClass` argument for the `class()` method**
  - This is useful when the base class is defined in another module

Syntax

```
import modname

class DerivedClass(modname.BaseClass):
  Code
```

Example: Basic Inheritance

Example

```
#!/usr/bin/python

class Base():
    var1 = 1
    var2 = 2

class Derived(Base):
    var2 = 'All your Base'
    var3 = 'Are Belong To Us'

CATS = Derived()

print(CATS.var1)
print(CATS.var2, CATS.var3)
```

Output

```
1
All your Base Are Belong To Us
```

Inheritance and Attribute Resolution

- **Derived classes are built and executed the same way as the base class is**
  - When a derived class is executed, its base class is remembered
    - Remember, base classes automatically inherit from the `Object` class in Python 3
- **The base class is used for resolving attribute references**
  - If an attribute is not found in the current class, Python proceeds to look in the base class for the attribute
  - This process is repeated recursively if the base class is derived from another class
- **This gives derived classes the ability to use commonly defined attributes in the base class**

Example: Implementing Inheritance

Example

```
#!/usr/bin/python

class Vehicle():
    num = 0
    def __init__(self):
        Vehicle.num += 1

class Auto(Vehicle):
    def setcolor(self, color):
        self.color = color
    def getcolor(self):
        return self.color

jag = Auto()
jag.setcolor("Gold")

bmw = Auto()
bmw.setcolor("Silver")

print(Vehicle.num)
print(jag.getcolor())
print(bmw.getcolor())
```

Output

```
2
Gold
Silver
```

Overriding Base Class Methods

- **Occasionally a developer will want to inherit a base class's constructor and override the capabilities of that constructor**

- **The most common coding pattern in Python to perform this task is to manually call the base class and pass the self argument explicitly to the base class**

Syntax

```
class Base():
    def __init__(self):
        pass
class Derived(Base):
    def __init__(self):
        Base.__init__(self):
        pass
```

Example: Customizing Constructors

Example

```
#!/usr/bin/python

class Vehicle():
    num = 0
    def __init__(self):
        Vehicle.num += 1
    def setcolor(self, color):
        self.color = color
    def getcolor(self):
        return self.color

class Auto(Vehicle):
    def __init__(self):
        Vehicle.__init__(self)
        self.numwheels = 4


jag = Auto()
jag.setcolor("Gold")
bmw = Auto()
bmw.setcolor("Silver")

print(Vehicle.num)
print(jag.getcolor(), jag.numwheels)
print(bmw.getcolor(), bmw.numwheels)
```

Output

```
2
Gold 4
Silver 4
```

Extending a Base Class with `super()`

- **New style classes may also extend base classes through the `super()` method**

Syntax

```
super([type[,object-or-type]])
```

- **The `super()` method returns a proxy object that delegates method calls to a parent class of `type`**

- **One of the primary uses for the `super()` method is in a single inheritance class hierarchy**
  - `super()` may be use to refer to a base class without explicitly naming the base class
    - This is useful for accessing inherited methods that have been overridden in a class

- **To extend the `__init__()` method of a base class, the syntax would be similar to the following**

Syntax

```
class Base():
    def __init__(self):
        pass
class Derived(Base):
    def __init__(self):
        super(Derived, self).__init__():
        pass
```

Example: Using `super()` to Customize a Constructor

Example

```
#!/usr/bin/python

class Vehicle():
    num = 0
    def __init__(self):
        Vehicle.num += 1
    def setcolor(self, color):
        self.color = color
    def getcolor(self):
        return self.color

class Auto(Vehicle):
    def __init__(self):
        super(Auto, self).__init__()
        self.numwheels = 4


jag = Auto()
jag.setcolor("Gold")
bmw = Auto()
bmw.setcolor("Silver")

print(Vehicle.num)
print(jag.getcolor(), jag.numwheels)
print(bmw.getcolor(), bmw.numwheels)
```

Output

```
2
Gold 4
Silver 4
```

Section 10–4

Built-in Methods

Operator Overloading Methods

- **Classes intercept and implement built-in operations by providing specially named method functions**
  - Each of these functions begins and ends with 2 underscores (__)
- **These methods are not reserved and may be inherited from super classes**
- **Python locates and calls at most one instance of these methods per operation**
- **Python automatically calls a class's overloading methods when instances appear in expressions and in other contexts**
  - For Example, when a class in instantiated, its `__init__()` method is called

Class Private Names

- **Names defined within a class statement with 2 leading underscores (`__`) only are mangled at compile time to include the enclosing class name as a prefix using the style `_class__`**
  - For example a class with the name `Test` with an `__init__()` method would be mangled to `_Test__init__()`
- **The added class name prefix localizes these names to the enclosed class**
  - This makes the names distinct both in the self instance object and in the class hierarchy
- **This system is helpful to avoid clashes that may arise at runtime for methods and attributes of the same name in the inheritance chain**
- **This system does not provide for strict privacy however, because the attributes may still be accessed through the mangled name**

Operator Overloading Methods `__init__()` and `__del__()`

- **Python's operator overloading methods are so common place to use, we have been working with 2 of them for the entire chapter**

- **`__init__(self [,args])`**
  - This method is invoked on class(args) and initializes the new instance, self

- **`__del__(self)`**
  - This method is invoked on instance garbage collection and cleans up when an instance is freed

Operator Overloading Methods `__str__()` and `__repr__()`

- **The `__str__(self)` method is invoked when calling `str(self)` or `print(self)`**
  - This method returns back a "user friendly" string representation of self

- **The `__repr__(self)` function is invoked when calling `repr(self)` and interactive echo calls**
  - This method returns a low-level "as code" string representation of `self`

Example

```
class Vehicle():
    num = 0
    def __init__(self):
        Vehicle.num += 1
class Auto(Vehicle):
    def __init__(self):
        Vehicle.__init__(self)
        self.numwheels = 4
jag = Auto()
print(repr(Vehicle))
print(repr(Auto))
print(repr(jag))
```

Output

```
<class '__main__.Vehicle'>
<class '__main__.Auto'>
<__main__.Auto object at 0x02AC56D0>
```

Example: Using `__str__()` and `__repr__()`

Example

```
class Vehicle():
    num = 0
    def __init__(self):
        Vehicle.num += 1
    def setcolor(self, color):
        self.color = color
    def getcolor(self):
        return self.color
    def __str__(self):
        return "I am a Vehicle"
class Auto(Vehicle):
    def __init__(self):
        Vehicle.__init__(self)
        self.numwheels = 4
    def __str__(self):
        msg =  "I am an auto and I have {0}
         wheels".format(self.numwheels)
        return msg

print(Vehicle)
x = Vehicle()
print(x)
print(Auto)
y = Auto()
print(y)
```

Output

```
<class '__main__.Vehicle'>
I am a Vehicle
<class '__main__.Auto'>
I am an auto and I have 4 wheels
```

Calling `help()` on a Class

- **Calling the `help()` function on a class returns the class's Docstring and any other documentation that the help system can locate, including:**
  - Module Name
  - Functions
  - Classes
  - Methods
  - Keywords

Syntax

```
help(Class)
```

Example: Using `help()`

Example

```
class Vehicle():
    """ The Vehicle Class is the Base class for other
Vehicle-based Classes"""
    num = 0
    def __init__(self):
        Vehicle.num += 1
    def setcolor(self, color):
        """ Sets the color of an instance of a Vehicle
         Object"""
        self.color = color
    def getcolor(self):
        """ Returns the color of an instance of a Vehicle
         Object"""
        return self.color
help(Vehicle)
```

Output

```
Help on class Vehicle in module __main__:
class Vehicle(builtins.object)
 |  The Vehicle Class is the Base class for other Vehicle
based Classes
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  getcolor(self)
 |      Returns the color of an instance of a Vehicle
Object
 |
 |  setcolor(self, color)
... (Extra output removed here)
 |  num = 0
```

Module Summary

- **The benefits of Object-Oriented Programming are**
  - Code Reuse
  - Encapsulation
  - Maintenance
  - Extensibility
- **Python is an Object-Oriented Programming language**
- **Python classes support an extensible, inheritance hierarchy**
- **Python supports a robust collection of built in class methods**
- **Python implements automatic, reference-based garbage collection**

Module 11

Using Modules in Python Programs

Module Goals and Objectives

Major Goals

Understand how to define and use Modules

Understand how to import Modules

Discuss common and useful Python Modules

Specific Objectives

On completion of this module, students will be able to:

Create Modules

Import all symbols from a Module

Import specific symbols from a Module

Access and Change Python's Search Path

Locate the Compiled version of a Module

How to hide symbols in a Module

Section 11–1

Module Overview

Module Overview

- **Often, collections of Classes, Functions and Variables will need to be reused in a number of applications, or in many instances of a running application**

- **Python has a robust application organization system that may be used to collect related Classes, Functions and Variables into files that may be imported by any python program that needs them**

- **Related sets of files may be organized into a logical namespace on disk for Organization, Access and Distribution purposes**

- **Python uses Modules to group together related Classes, Functions and Variables**
  - Modules may be consumed by any Python program
  - Modules are also Python Programs, therefore they may be executed like any other Python Program

- **Packages are directories that contain Collections of Python Modules**

- **As part of Python's "Batteries Included" Philosophy, Python ships with a large standard library of modules organized into different packages**

Section 11–2

Understanding Modules

The Structure of Python Programs

- **Python programs are comprised of a number of Modules**
  - May be library modules
  - May be the script or "main" (`__main__`) module that performs the work of the application
- **Modules are comprised of a number of statements**
- **Statements contain expressions**
- **Expressions are used to create and process objects**

Understanding Modules

- **A Module is a file that contains Python Code**
- **Modules are the highest level organizational unit for code in Python**
  - Package code and/or data for reuse in Python Programs
  - Create self-contained namespaces to reduce variable name collisions
  - Implement shared services or shared data
- **Modules provide a convenient way to organize program components into a self-contained system**
  - Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module
  - A self contained package of variables is known as a namespace
- **A module may contain executable statements as well as function definitions**

Using Modules

- **Modules may be imported by other Python programs, or modules may be directly executed by other Python programs**
- **Modules are imported by using the `import` statement**
- **Within a module, the module's name is available as the value of the global variable `__name__`**
- **Statements in a module are only executed the first time the module name is encountered in an import statement**

The Import Process

- **The Python import process is a 3-step operation that executes at runtime**

- **First, Python must locate the module referenced by the import statement**
  - Python uses the module search path (`sys.path`) plus a combination of known file types (`.py, .pyc`) to locate the module referenced by the import statement

- **Second, Python compiles the module into bytecode if necessary**
  - Python creates bytecode if the bytecode doesn't exist, is older than the timestamp on the `.py` file, or was created by a different Python version
  - Python uses the existing bytecode if it is not older than the current `.py` file and is created by the same version of Python that is calling the module

- **Third, the final step is to execute the module's bytecode**
  - All statements in the module are executed
  - Any assignments made in the module generate the module's attributes
  - Module imports only occur 1 time

Module Search Path

- **`sys.path` is Python's module search path**
- **The sys module contains Python's interpreter-related exports such as standard I/O streams, command line argument access and environmental components**
- **At program startup, Python configures `sys.path` by merging together path statements from the following sources into a mutable list of directory strings**
  - The current directory of the program
  - Any `PYTHONPATH` directories
  - Standard Library directories
  - Contents of any `.pth` (path) files
  - Any `site-packages` of any installed 3rd party extensions
- **Python searches through `sys.path` by concatenating the name of the module you are importing with one of the following extensions (`.py, .pyc, .pyo`) to the paths in `sys.path`**
  - The first instance of a module found in the search path is the instance of the module that is used by Python

Example: sys.path

Example

The following example displays `sys.path` for Python 3.3 on a Windows machine and for Python 3.2 and 2.7 on a Linux machine. The `-c` argument for Python allows for a program to be passed to Python as a string

```
#Python 2.7 on Linux
$ python -c 'import sys; print(sys.path)'
['', '/usr/lib/python27.zip', '/usr/lib/python2.7',
'/usr/lib/python2.7/plat-cygwin', '/usr/lib/python2.7/lib-
tk', '/usr/lib/python2.7/lib-old', '/usr/lib/python2.7/lib-
dynload', '/usr/lib/python2.7/site-packages',
'/usr/lib/python2.7/site-packages/PIL',
'/usr/lib/python2.7/site-packages/gtk-2.0',
/usr/lib/python2.7/site-packages/setuptools-0.6c11-
py2.7.egg-info']

#Python 3.2 on Linux
$ python3 -c 'import sys; print(sys.path)'
['', '/usr/lib/python32.zip', '/usr/lib/python3.2',
'/usr/lib/python3.2/plat-cygwin', '/usr/lib/python3.2/lib-
dynload', '/usr/lib/python3.2/site-
packages','/usr/lib/python3.2/site-packages/setuptools-
0.6c11-py3.2.egg-info']

#Python 3.3 on Windows
C:\HOTTPython>python -c "import sys; print(sys.path)"
['', 'C:\\Python33\\lib\\site-packages\\setuptools-0.9.8-
py3.3.egg', 'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs', 'C:\\Python33\\lib', 'C:\\Pyth
on33', 'C:\\Python33\\lib\\site-packages']
```

bytecode

- **Python source code is compiled into `bytecode`, the internal representation of a Python program in the CPython interpreter**

- **The `bytecode` is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided)**

- **This "intermediate language" is said to run on a virtual machine that executes the machine code corresponding to each bytecode**
  - A Virtual Machine is a computer defined entirely in software
  - Python's virtual machine executes the bytecode emitted by the bytecode compile

- **`bytecodes` are not expected to work between different Python virtual machines**

- **`bytecodes` are not expected to be stable between Python releases**

`bytecode` Storage

- **In Python versions 3.1 and earlier (including 2.x), `bytecode` is stored in files `(.pyc)` in the same directory as the corresponding source files (e.g. `module.pyc`)**
  - `bytecode` files are marked internally with the version of Python that creates them
- **In Python versions 3.2 and later, any generated bytecode is stored in files in a subdirectory called `__pycache__` which is a subdirectory of the directory that contains the original source module**
  - `bytecode` files are also named to include the implementation of Python that created them
    - This allows multiple sets of `bytecode` from multiple implementations of Python to coexist on the same machine
  - calcpay.cpython-33.pyc

Section 11–3

Creating Modules

Creating a Module

- **Creating a module is a fairly simple process**
- **To create a module, use your editor to create a file with a `.py` extension**
  - All files with a `.py` extension are considered to be modules by Python
- **Any names defined in the top level of the module are considered to be the module's attributes**
  - A module's attributes are exported for the modules client to use

Syntax

```
#mytestmod.py
def testfunc():
    pass
```

Syntax

```
#usemytestmod.py
import mytestmod
mytestmod.testfunc()
```

Example: Creating a Module

Example

In this example we are going to create a simple module with an embedded test suite. The module will execute its own code if executed directly. We will then import the module into a Python program that will consume a symbol from the Python module

Example

```
#Module
#mytest.py

def testfunc():
    print("Hello from", __name__)

if __name__ == '__main__':
   testfunc()

#Output
Hello from __main__
```

Example

```
#usemytest.py

import mytest

mytest.testfunc()

#Output
Hello from mytest
```

Importing Modules Using the `import` Statement

- **Modules may be imported by using the `import` statement**
- **The import statement identifies an external file(s) to be loaded**
  - The loaded file(s) become variables in the calling script

Syntax

```
import module [,module]*
import [package.]module [,[ package.]module]*
import [package.]module as name
         [,[ package.]module]* as name
```

- **The import statement imports the module as a whole**
  - Target modules may be Python source code files (`.py`) or compiled modules (`.pyc`)
- **Any assignments in the top level of the module create the module's attributes**
  - Attributes may be accessed using a qualified path statement (`module.attribute`)
- **The optional `as` statement allows an imported module to be referenced with an alias**

Example: Using the import Statement

Example

In this example we are going to store payroll calculator code in a module. We will import the module into a calling script with and without aliasing the module

```
# Module calcpay.py

bh = 40
otm = 1.5

def calc_ot(hours, rate):
    ot_hours = hours - bh
    ot_pay = ot_hours * rate * otm
    gross = bh * rate + ot_pay

    print('The gross pay is $%.2f.' % gross)

def calc_pay(hours, rate):
    gross = hours * rate

    print('The gross pay is $%.2f.' % gross)
```

Example

```
#Calling file pay.py
import calcpay

hours = float(input('Enter the number of hours worked: '))
rate = float(input('Enter the hourly pay rate: '))

    # Calculate and display the gross pay.
if hours > calcpay.bh:
    calcpay.calc_ot(hours, rate)
else:
    calcpay.calc_pay(hours, rate)
```

Example: Importing a Module with an alias

Example

This example uses the module from the previous example and imports the module with an alias

```
#payalias.py
import calcpay as c

hours = float(input('Enter the number of hours worked: '))
rate = float(input('Enter the hourly pay rate: '))

    # Calculate and display the gross pay.
if hours > c.bh:
    c.calc_ot(hours, rate)
else:
    c.calc_pay(hours, rate)
```

Importing Variable Names Using the `from import` Statement

- **The `from import` statement is used to import variable names from a module into the calling script's namespace**
  - This allows you to use the name without qualifying the name with the module
- **As of Python 2.4, imported names may be enclosed in parentheses**
  - This allows your import list to span multiple lines without having to manually span lines with the backslash \ operator

Syntax

```
from [package.]* module import name [,name]*
from [package.]* module import *
from [package.]* module import name as alias
from [package.]* module import (name, namex, .....)
```

- **`from mod import *` imports all names assigned at the top level of a module *except* for those names with a single leading underscore (`_name`) or names not listed in a module's `__all__` attribute**
- **The `as` clause may be used to alias a name**

Example: from mod import *

Example

This example uses the previous module and imports names into the calling script's namespace

```
#payimportsplat
from calcpay import *

hours = float(input('Enter the number of hours worked: '))
rate = float(input('Enter the hourly pay rate: '))
print(50 > 40)
    # Calculate and display the gross pay.
if hours > bh:
    calc_ot(hours, rate)
else:
    calc_pay(hours, rate)
```

Module Namespaces

- **Internally, module namespaces are stored as dictionary objects**
- **Each module has an `__dict__` attribute that provides access to the module's namespace dictionary**
- **Each name assigned in the module file becomes a key in the module's namespace dictionary**
- **The `dir()` function may be used to list off the contents of the module's `__dict__` attribute**

Example

```
dir(calcpay)
```

- **The keys from the `__dict__` attribute may be accessed directly using the dictionary object's `keys()` method**
  - This is useful if you need to search for specific keys

Example

```
list(calcpay.__dict__.keys())
```

```
['__builtins__', '__name__', '__file__', '__package__',
'bh', 'otm', '__loader__', '__cached__', '__doc__',
'calc_pay', '__initializing__', 'calc_ot']
```

Example: Listing Keys in a Namespace

Example

In this example we are going to use generators to list off the contents of the `__dict__` attribute of our module. We are going to filter for attributes created by Python and we are going to filter for attributes created by us. We are going to return the list of keys using a list comprehension and a generator. Because we are using a generator, we do not need to explicitly call the `keys()` method because dictionaries implicitly generate their keys

```
import calcpay

print(list(key for key in calcpay.__dict__.keys() if
key.startswith('__')))

#['__file__', '__builtins__', '__initializing__',
#'__package__', '__name__', '__loader__', '__doc__',
#'__cached__']

print(list(key for key in calcpay.__dict__.keys() if not
key.startswith('__')))

#['calc_ot', 'otm', 'bh', 'calc_pay']

print(list(key for key in calcpay.__dict__ if
key.startswith('__')))

#['__file__', '__builtins__', '__initializing__',
#'__package__', '__name__', '__loader__', '__doc__',
#'__cached__']

print(list(key for key in calcpay.__dict__ if not
key.startswith('__')))

#['calc_ot', 'otm', 'bh', 'calc_pay']
```

Hiding Attributes

- **Recall, that the `from` statement (`from mod import `) imports all names except those not listed in `__all__` and those that have a leading `_`**

- **This means you have the ability to hide (pseudo-privatize) attributes in your modules by using the `from` statement to perform imports and creating objects whose names start with a single leading `_`**

Example

privtest.py
```
def visible(arg):
    print(arg)
def _invisible(arg):
    print(arg)
```

priv.py
```
from privtest import *

print(visible('hello'))

print(_invisible('hello'))
```

```
Message  File Name    Line Position
Traceback
    <module> C:\HOTTPython\ModulesAndPackages\priv.py   5

NameError: name '_invisible' is not defined
```

Changing the Module Search Path

- **One way to change the Python Module search path is to extend it at runtime if necessary**

- **`sys.path` is a list of search paths, which means it may be altered like any other list at runtime**

Syntax

```
sys.path.append(r'path')
```

- **Used to append a path to the end of `sys.path`**

Syntax

```
sys.path.insert(i,r'path')
```

- **Used to insert a new path anywhere in `sys.path`**
  - This is a useful technique to control the `sys.path` search order

- **These changes only last as long as the current session or process is active**

Example: Changing `sys.path`

Example

In this example we are going to execute the file `altersyspath.py` in the `C:\HOTTPython` directory and have the file change `sys.path` so we can import the `calcpay.py` module in the `C:\mods` directory

```
print(__file__)

import sys
print(sys.path)
# C:\HOTTPython\altersyspath.py

#['C:\\HOTTPython', ... 'C:\\Python33',
#'C:\\Python33\\lib\\site-packages']


sys.path.append(r'c:\testmod')
print(sys.path)
#['C:\\HOTTPython', ..., 'C:\\Python33\\lib\\site-
#packages', 'c:\\testmod']


sys.path.insert(0,r'c:\mods')
print(sys.path)
#['c:\\mods', 'C:\\HOTTPython', ...,
#'C:\\Python33\\lib\\site-packages', 'c:\\testmod']


import calcpay
print(calcpay.__file__)
#c:\mods\calcpay.py
```

Section 11–4

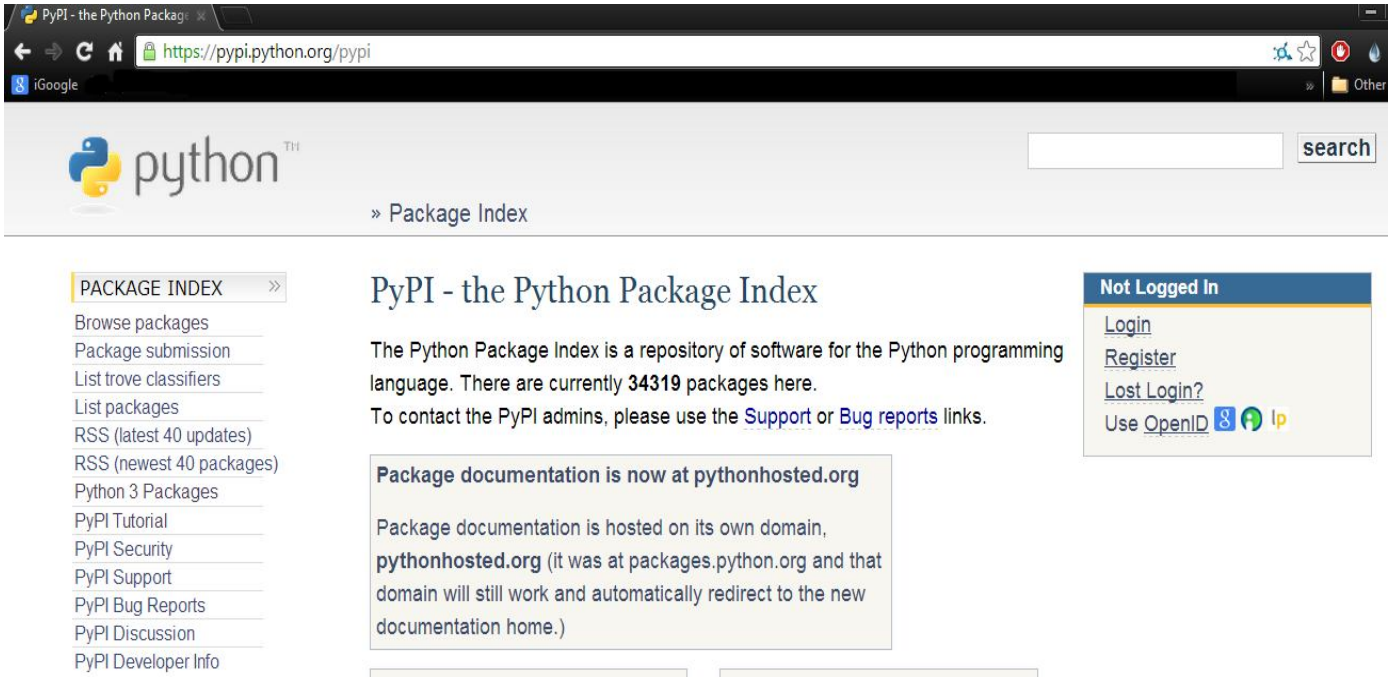
Installing Modules

Installing Modules Overview

- **Python ships with a very robust and mature set of modules in its standard library**
- **It may become necessary over time to add new functionality to your programs or to Python itself through the addition of new modules**
- **Prior to Python 2.0, there was little support for adding 3rd party modules**
- **Starting with Python version 1.6, the Python Standard library started shipping with a standard module to install modules**
  - Python Distribution Utilities (`distutil`)
- **General distribution of Python code is performed using the `Distutils` package from the standard library**
  - `Distutils` can produce source and binary distributions on systems that have Python installed
- **Other distribution tools include Easy Install which is a python module (`easy_install`) bundled with `setuptools` that lets you automatically download, build, install, and manage Python packages**
- **`pip` is another popular tool for installing, uninstalling and managing Python packages**

The Python Package Index (PyPi) aka: The Cheese Shop

- The Python Package Index is the Python Package Repository
- PyPi may be found at `http://pypi.python.org/pypi`
- There are currently over 30,000 packages located at PyPi
- Packages may be downloaded and installed manually using distutils
- Packages may be downloaded and installed automatically using Easy Install



Installing Modules

- **A number of platform-specific modules exist and are installed using the installer of your platform**
- **Red Hat, SuSE, etc users can often locate Red Hat packages that may be installed with the RedHat Package Manger**
- **Debian packages may be installed by using `apt-get` or `yum`**
- **Windows users will often be able to locate Windows Executable files that may be used to install packages**
  - ActivePython users may use the ActiveState ActivePython Package Manager (PyPM) to install Packages
- **Regardless of your platform, it is possible to download the package for your system and install the package using Distutils**

Installing Packages with Distutils

- **Locate and download the Package that you are going to install**

- **Packages that are packaged and distributed according to the standards of using Distutils, have a standard naming syntax**
  - The distribution's name and version number should be the prominent features of the package
    - (e.g. `handcannon-1.6.9.tar.gz` or `windowsmander-4.6.9.zip`)

- **Unpack the distribution to disk**
  - The root level directory should be similar in name to the file that was downloaded

- **The root directory should contain a setup script called `setup.py` and a `README` file**

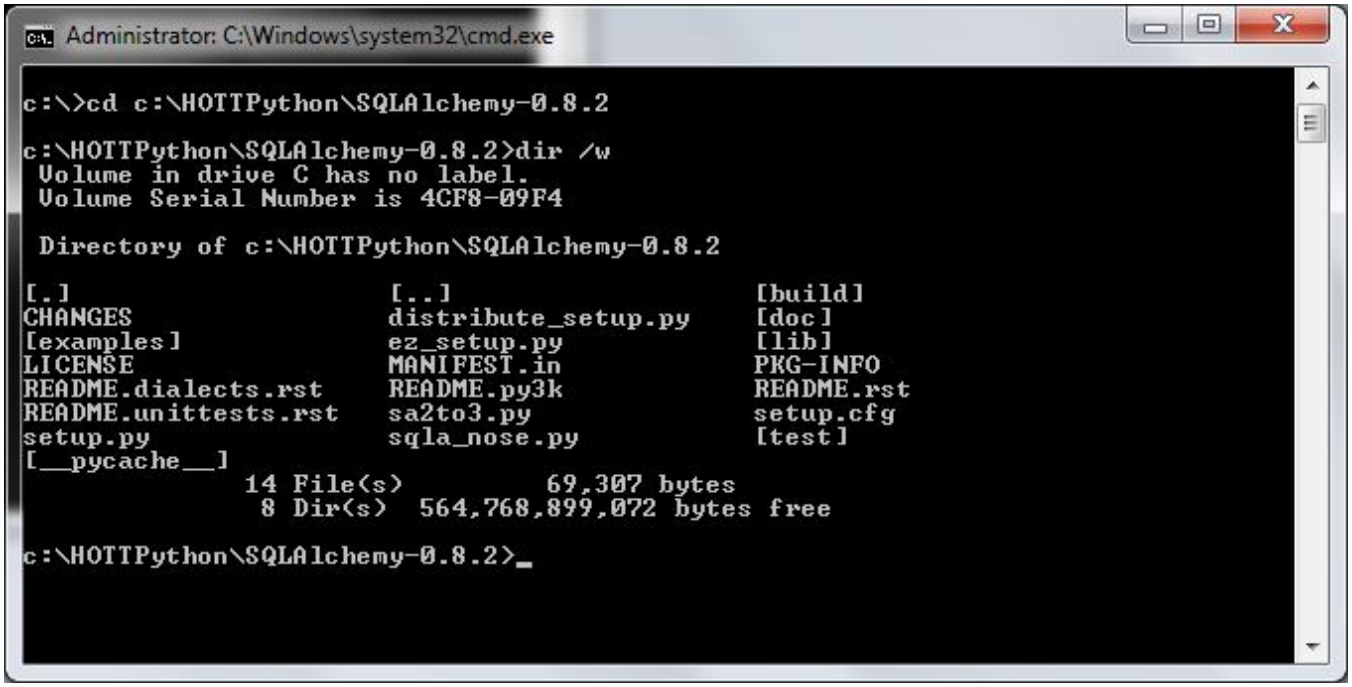
- **Navigate to the root directory of the package you downloaded and run the `setup.py` program**

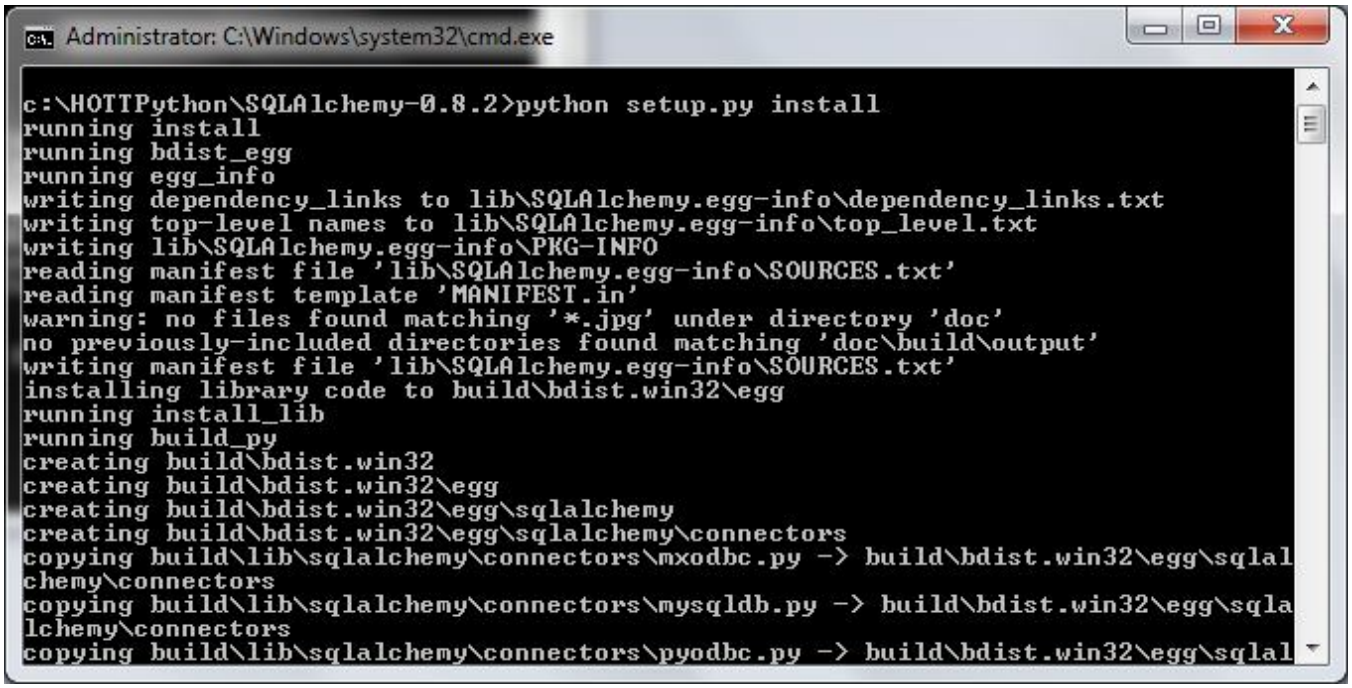
- **The standard syntax for installing a Distutils-based module distribution is:**

Syntax

```
python setup.py install
```

Example: Installing a Module





Module Summary

- **Python Modules may be used to share symbols among multiple Python programs**
- **All symbols or specific symbols may be imported from the Module's name space**
- **`sys.path` is Python's module search path**
- **Modules may be downloaded from the Python Package Index**
- **Distutils is the standard Package Distribution and installation system in Python**

Module 12

Exception Handling

Module Goals and Objectives

Major Goals

Understand Exceptions

Understand how to use Python's Exception Hierarchy

Specific Objectives

On completion of this module, students will be able to:

Use try/except blocks to manually handle exceptions

Control the execution of exception handlers with else and finally

Raise exceptions in response to errors

Create custom Exception classes

Inherit from existing base classes

Handle OS Errors

Section 12–1

Understanding Errors and Exceptions

Error and Exception Overview

- **In Python there are 2 kinds of errors**
  - Syntax Errors
  - Exceptions
- **Syntax errors (aka Parsing Errors) are the most common type of error to occur in Python**
  - Syntax errors are raised when the Python interpreter does not understand how to parse the code
  - Syntax errors are always fatal
  - Python returns an error message when it detects a syntax error
  - Syntax errors are instances of the `SyntaxError` exception object
- **Sometimes errors occur when syntactically correct code is executed**
- **Errors that occur when syntactically correct code is executed are known as Exceptions**
- **Exceptions are a powerful tool to deal with rare or atypical issues in the system**
- **Python has a large number of built-in exceptions which makes it easier for developers to handle exceptions in their programs**

Exceptions

- **Exceptions are events that interrupt the normal processing of a program**
- **When an exception is raised, Python will handle the exception immediately**
- **Exceptions have the ability to dynamically change the execution sequence of a program**
- **Exceptions also return an object that contains information about the exception that occurred**
- **Python allows developers to manually handle exceptions through the use of `try...except` statements**
- **Developers may also manually raise exceptions by using the `raise` statement**
- **All exceptions are instances of classes that are derived from Python `BaseException` class**

Built-in Exceptions

- **Built-in exceptions may be generated by the interpreter or by built-in functions**
  - Most exceptions have an associated value that details the cause of the error
- **Developers may raise any built-in exception**
  - This allows developers to raise exceptions in a manner that is consistent with the way that Python raises errors
  - There is no mechanism in Python that prohibits a developer from raising the wrong type of exception
- **Developers may also define new exceptions**
  - New exceptions may use any existing exception object as a base class for the new exception class
- **All exceptions are instances of a class that derives from a class called `BaseException`**
- **All built-in, non-system-exiting exceptions are derived from a class called `Exception`**
  - `BaseException` is the `Exception` classes base class
  - All user-defined exceptions should be derived from the `Exception` class

Python's Exception Hierarchy

- **Python's Exception Hierarchy may be broken down into three broad categories**
  - Non-error Exceptions
    - Exceptions that define events and change the sequence of execution
  - Run-time Errors
    - Exceptions that occur in the normal course of program execution
    - Indicate typical program problems
  - Internal Errors
    - Exceptions that occur when compiling a Python program or when there are problems with the Python interpreter's internals
    - Typically, recovery isn't possible

Example

The following is the new exception hierarchy for Python 3.3

```
BaseException
  SystemExit
  KeyboardInterrupt
  GeneratorExit
  Exception
       StopIteration
       ArithmeticError
           FloatingPointError
           OverflowError
           ZeroDivisionError
       AssertionError
       AttributeError
       BufferError
       EOFError
       ImportError
       LookupError
           IndexError
```

Python's Exception Hierarchy *cont'd*

```
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
```

Useful Built-in Exceptions

- **While Python has an extensive built in Exception collection, there are a number of common exceptions a developer works with directly or indirectly on a daily bases**

- **`BaseException`**
  - This is the root super class for all built in exceptions
    - Not designed to be directly inherited from by user-defined classes—use `Exception` for that
  - Instance constructor arguments are stored and made available through an args argument
    - All subclasses of `BaseException` inherit `args`
  - If the `str()` function is called on an instance of this class, a representation of the constructors arguments will be returned

- **`Exception`**
  - This is the root super class for all built-in and non-system-exiting exceptions
  - User-defined exceptions should be derived from this class
  - `try` statements that catch this exception will catch all but system exit events (`SystemExit`, `KeyboardInterrupt`, `GeneratorExit`)

Table of Useful Exceptions

Exception	Purpose
`ArithmeticError`	Arithmetic error exceptions: Superclass for `OverflowError`, `ZeroDivisionError`, `FloatingPointError`
`LookupError`	Sequence and mapping index errors: Superclass for `IndexError` and `KeyError`
`EnvironmentError`	Exceptions that occur outside of Python: Superclass for `IOError` and `OSError`. Contains `errno` attribute and `strerror` attribute as part of its `args` collection
`AttributeError`	Raised on attribute reference failure or assignment failure
`FloatingPointError`	Raised on floating-point operation failure
`IOError`	Raised on I/O or file-related operation failure. Merged with `OSError` in Python version 3.3
`OSError`	Exception raised when a system function returns a system-related error including I/O failures. In Python 3.3 `IOError`, `WindowsError` and `VMSError` have been merged into `OSError`
`KeyboardInterrupt`	Raised on user entry of the interrupt key
`NameError`	Raised when local or global unqualified names are not found
`IndentationError`	Raised when improper indentation is found in source code
`SyntaxError`	Raised when Parsers encounter a syntax error. May occur on `import` calls, or calls to `eval()` or `exec()`
`TypeError`	Raised when an operation or function is applied to an inappropriate object

Section 12–2

Handling Exceptions

Handling Exceptions with the `Try` Statement

- **Exceptions are handled using the `try` statement**
- **The `try` statement catches exceptions**
- **Primarily, the `try` statement contains a suite of statements and exception handling clauses**
- **Exception handling clauses identify the type of exception to handle and provide a suite of statements to execute in response to the exception**
  - The suites of code attached to each `except` statement act as handlers for that type of error
- **The `try` statement supports an `else` statement that executes if no exceptions are caught**
- **The `try` statement also supports a `finally` statement that is always executed at the end of a try statement**

`try` Statement Syntax

Syntax

```
try:
    suite
except [exception[as val]]:
    suite
[except [exception[as val]]:
    suite]*
[else:
    suite]
[finally:
    suite]
```

Example

In this example we are going to implement exception handling for a specific Exception class and any other Exceptions that may be raised

```
#!/usr/bin/python
try:
# Ask user how many hours did they work
    hours = int(input('How many hours did you work? '))
# Ask the user for their pay rate
    pay_rate = float(input('Enter your hourly pay rate: '))
# Calculate Cash Money
    gross_pay = hours * pay_rate
# Display the awesome pretax revenues
    print('Gross pay: $', format(gross_pay, ',.2f'),
    sep='')
except ValueError:
    print('ERROR: Hours worked and hourly pay rate must')
    print('be valid integers.')
except:
    print('Something awful happened!')
```

`try` Statement Execution Process

- **When a `try` statement is encountered, Python marks the program's current execution context to return to in case an exception occurs**

- **The `try` statement then executes the suite of code attached to the try header**
  - If an exception occurs and the exception matches one named by an `except` block, Python runs the statements attached to the first except block that matched the raised exception
    - Also, if present, Python assigns the raised exception to a variable named after the `as` statement
  - If an exception occurs that doesn't match any Exception named by an `except` block or there is no generic `except` block, the exception is propagated up to the most recently seen `try` block for handling
    - If the error propagates to the top level of the process without being handled, Python kills the program and prints a default error message
  - If no errors occur in the `try` block's suite of code, Python runs any code found in an `else` statement (if present) then returns control to the next statement
    - An `else` statement may only exist if at least 1 `except` block exists in the `try` statement

`try` Statement Clauses

- **The `except` clauses are focused exception handlers that catch exceptions that occur only in the associated `try` block**

- **Exceptions that are raised are matched to exceptions named in the `except` clauses by `superclass` relationships or by an empty `except` clause that matches all exceptions**

Clause	Meaning
`Except:`	Catch all (or remaining) exception types
`Except exception:`	Catch a specific exception
`Except exception as e`	Catch the listed exception and assign an exception instance
`Except (exception1, exception2)`	Catch any of the listed exceptions
`Except (exception1, exception2) as e`	Catch any listed exception and assign an exception instance
`Except exception , e`	Python 2.x : Catch the listed exception and assign an exception instance
`Except (exception1, exception2) , e`	Python 2.x : Catch any listed exception and assign an exception instance
`Else:`	Code to execute if no exceptions are caught. Requires at least 1 except block to be coded
`Finally:`	Code to execute always when exiting try block

Catching Exceptions

- **Python interrogates the exception list from top to bottom**
  - This gives you the ability to write your exception handlers like an `if/elif` block
- **To catch a specific exception, supply a specific exception class to the except clause**
- **To catch a list of exceptions, supply a list of exceptions to the except clause**
- **To create a "catchall" clause, simply add an empty except clause to a try block**
  - This will catch any exception raised

Syntax

```
try:
    suite
except:
    suite

try:
    suite
except ExceptionClass:
    suite

try:
    suite
except (ExceptionClass1, ExceptionClass2, ExceptionClass3):
    suite
```

Example: Catching Exceptions

Example

In this example, we are going to ask the user to supply some numbers though standard input. We will then handle some specific exceptions that may occur.

```
try:
    num1, num2 = eval(
    input("Enter two numbers, separated by a comma: "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero!")
except SyntaxError:
    print("A comma may be missing in the input")
except KeyboardInterrupt:
    print("User Interrupted Input")
except:
    print("Something wrong in the input")
```

Output

```
Enter two numbers, separated by a comma: 1,1
Result is 1.0

Enter two numbers, separated by a comma: 1 2 3 4
A comma may be missing in the input

Enter two numbers, separated by a comma: User Interrupted
Input

Enter two numbers, separated by a comma: wafer, thin
Something wrong in the input
```

`try/else` statement

- **The `try/else` statement is useful because it provides a way to determine if the flow of the program has proceeded past a `try` statement without producing an error**
  - This behavior is very useful for testing purposes because it makes the processing of the `try` block much more obvious

Syntax

```
try:
    suite
except:
    suite
else:
    suite
```

Example

```
try:
    num1, num2 = eval(
    input("Enter two numbers, separated by a comma: "))
    result = num1 / num2
    print("Result is", result)
except ZeroDivisionError:
    print("Division by zero!")
except SyntaxError:
    print("A comma may be missing in the input")
except KeyboardInterrupt:
    print("User Interrupted Input")
except:
    print("Something wrong in the input")
else:
    print("You've performed division, wisely")
#Output with 1,1 entered into program
Result is 1.0
You've performed division, wisely
```

`try/finally` Statement

- **`finally` statements allow developers to attach a suite of code that is always executed at the end of a `try` statement**
- **If an exception does not occur during the `try` block, Python executes the suite attached to the `finally` block and passes execution onto the next statement**
- **If an exception does occur in the try block, Python returns to the finally block, executes the code in the finally block and then propagates the exception**
- **The try/finally form of the try statement is useful with you want to be completely sure that a suite of code runs, regardless of the exception behavior**
  - Such as `disconnect` statements or `close` statements

Syntax

```
try:
    suite
finally:
    suite

try:
    suite
except:
    suite
else:
    suite
finally:
    suite
```

Example: `try/finally`

Example

```
try:
    num1, num2 = eval(
    input("Enter two numbers, separated by a comma: "))
    result = num1 / num2
    print("Result is", result)

except ZeroDivisionError:
    print("Division by zero!")
except SyntaxError:
    print("A comma may be missing in the input")
except KeyboardInterrupt:
    print("User Interrupted Input")
except:
    print("Something wrong in the input")
else:
    print("You've performed division, wisely")
finally:
    print("I am here to tidy things up")
```

Output

```
Result is 1.0
You've performed division, wisely
I am here to tidy things up
```

Example: Exception Handling Opening Files

Example

In this example we are going to allow the user to supply the path to a file (preferably a comma separated values file) and parse through the contents. An Exception will be raised if the user does not supply a valid file.

```
def main():
# Get the name of a file.
    filename = input('Enter a filename: ')

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

        for line in infile:
            # split the string on the ,'s
            fields= line.split(",")
            for fld in fields:
                    print(fld, end = ' ')
            print()
    # Close the file.
        infile.close()
    except OSError:
        print('An error occurred trying to read')
        print('the file', filename)
# Call the main function.
if __name__ == '__main__':
    main()
```

Output

```
Enter a filename: qvcmia
An error occurred trying to read
the file qvcmia
```

Working with an Exception Object's Attributes

- **When an exception is thrown, an exception object is created in memory**
- **The exception object will typically contain information about the exception that was thrown**
  - This is a useful form of introspection to further deduce the cause of an error
- **The `except` statement may optionally assign the exception object to a name through the use of the as keyword**

syntax

```
except Exception as e
```

- **Exception objects may be passed to the `print` statement to output their default error messages**
- **Exception objects also have access to any arguments passed to them through the `args` attribute**

Example

```
a=Exception(1,2,3)
print(a.args)

# (1, 2, 3)
```

Example: Error Object Instance

Example

In this example we are going to create a payroll calculator and display the default error messages for any ValueError exceptions we encounter

```
def main():
    try:
        # Get the number of hours worked.
        hours = int(input('How many hours did you work? '))
        # Get the hourly pay rate.
        rate = float(input('Enter your hourly pay rate:
         '))
        # Calculate the gross pay.
        gross = hours * rate
        # Display the gross pay.
        print('Gross pay: $', format(gross, ',.2f'),
         sep='')
    except ValueError as err:
        print(err)
# Call the main function.
main()
```

Output

```
Result for sending steve as the number of hours worked
invalid literal for int() with base 10: 'steve'

Result for sending About Tree Fiddy as the hourly rate
could not convert string to float: 'About Tree Fiddy'
```

Raising Exceptions with the `raise` statement

- **The `raise` statement triggers exceptions**
  - The `raise` statement may be used to raise built-in exceptions or user defined exceptions
- **`raise` creates an exception object, and immediately leaves the expected program execution sequence to search the enclosing `try` statements for a matching `except` clause**
- **The effect of a raise statement is to either divert execution in a matching except suite, or to stop the program because no matching except suite was found to handle the exception.**
- **The Exception object created by `raise` may have arguments passed to it, such as an error message**

Syntax

```
raise instance()
raise class()
raise
```

- **Calling raise without arguments re-raises the most recent exception**

Example: Raising Exceptions

Example

This simple example demonstrates how to raise an existing Exception class

```
def main():
    raise ValueError

if __name__ == '__main__':
    main()
```

Output

```
Traceback (most recent call last):
  File "simpleraise.py", line 6, in <module>
    main()
  File "simpleraise.py", line 3, in main
    raise ValueError
ValueError
```

Example: Raising Exceptions *cont'd*

Example

This simple example demonstrates how to pass an argument to an existing Exception class

```
def main():
    raise ValueError("This is the wrong value!")

if __name__ == '__main__':
    main()
```

Output

```
Traceback (most recent call last):
  File "simpleraiseMessage.py", line 6, in <module>
    main()
  File "simpleraiseMessage.py", line 3, in main
    raise ValueError("This is the wrong value!")
```

Defining New Exceptions

- **All Exceptions are classes**
- **To create a new Exception, simply define a new class that inherits from the Exception class**

Syntax

```
class NewException(Exception):
    pass

class NewException(Exception):
    def __init__(self):
        [self.args = (args here)]
        initialization statements here
```

- **To use a custom Exception, simply test for it or raise it like any other Exception**

Syntax

```
raise NewException()

try:
    suite
except NewException:
    suite
```

Example

```
class TestError(Exception): pass
try:
    raise TestError('This is just a test')
except TestError:
    raise
```

Customizing the Constructor for a Custom Exception Object

- **When raising an exception, the optional values supplied with the raise statement are used as the arguments to the exception's class constructor**
  - User-defined exceptions can be written to take one or more exception values
- **To make use of many attributes, define a custom constructor for your error class**

Syntax

```
class CustomError(Exception):
    def __init__(self,errno,msg):
    self.errno = errno
    self.errmsg = msg
```

Example: Custom Exceptions

Example

In this example we are going to create a custom Exception object. The Exceptions will be raised in response to data supplied by the user. The Exception and the function will be defined in a module that is imported by a calling script

```
#The Module userinteraction.py
class UserQuit( Exception ):
    pass

def uexit( prompt, help="" ):
    ok= 0
    while not ok:
        try:
            a=input( prompt + " [y,n,q,?]: " )
        except EOFError:
            raise UserQuit("EOF")
        if a.upper() in [ 'Y', 'N', 'YES', 'NO' ]: ok= 1

        if a.upper() in [ 'Q', 'QUIT' ]:
            raise UserQuit('User Quit the Application')
        if a.upper() in [ '?' ]:
            print(help)

#The Calling Script

import userinteraction as ui
answer= ui.uexit(
help= "Enter Y if finished entering data",
prompt= "All done?")
```

Example: Custom Constructor

Example

This example creates a custom constructor

```
class Overdrawn(Exception):
    def __init__(self, balance, amount):
        self.amount = amount
        self.balance = balance

    def over(self):
        return self.amount - self.balance

try:
    raise Overdrawn(25, 50)
except Overdrawn as e:
    print("I'm sorry, but your withdrawal is "
            "more than your balance by "
            "${}".format(e.over()))
```

Section 12–3

Context Managers and the `with` Statement

Context Managers

- **A context manager is an object that defines the runtime context to be established when executing a `with` statement**

- **The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code**

- **Context managers are normally invoked using the `with` statement but can also be used by directly invoking their methods**
  - Context managers have an `__enter__()` and an `__exit__()` method
    - An example of a context manager that returns itself is a `file` object
    - `File` objects return themselves from `__enter__()` to allow `open()` to be used as the context expression in a `with` statement.

- **Typical uses of context managers include**
  - Saving and restoring various kinds of global state
  - Locking and unlocking resources
  - Closing opened files

The `with` Statement

- **Python 2.6 and 3.0 introduced a new exception-related statement `with`**
  - The `with` statement may be enabled in Python 2.5 by using the following import statement

```
from __future__ import with_statement
```

- **The `with` statement is designed to work with context manager objects**
- **The `with` statement wraps a nested block of code in a context manager, which ensures that block entry and/or block exit actions are run**

Syntax

```
Python 2.6 and 3.0
with expression [as variable]:
    suite

Python 3.1 and higher
with expression [as variable] [,expression [as variable] ]*
:
    suite
```

- **The `expression` argument is assumed to return an object that supports the context management protocol**
  - The object may also return a value that will be assigned to the `variable` argument if the `as` statement is present

Using the `with` Statement

- **Many code sequences follow predictable patterns, with the same actions occurring from start to finish**
  - For instance, files need to be opened before they can be read or written to, and need to be closed when the read or write operation is completed
- **In a predictable sequence of events like that, `with` statements may be used as an alternative to `try / finally` statements for objects that have context managers**
- **`File` objects are context managers, so `with` statements are often used to guarantee that the file is closed once the last operation is processed against the file**

Example: Comparing `try/finally` and `with`

Example

The two blocks of code are functionally equivalent.

```
filename = 'passwords.csv'

print('Try / Finally Statements')
try:
    infile = open(filename)
    data = infile.read()
    print(data)
finally:
    infile.close()

print('with Statement')
with open(filename) as file:
    for line in file:
        print(line, end='')
```

Example: Working with multiple Context Managers

- **Python versions 3.1 and newer allow for the definition of multiple context managers which will work together as if they were defined in multiple, nested `with` statements**

Example

```
filename = 'passwords.csv'

with open(filename) as filein , open('passwordsup.csv',
'w') as fileout:
    for line in filein:
        flds = line.split(',')
        fileout.write(flds[0] + ',' + flds[1].upper() + ','
+ flds[2].upper())
```

Module Summary

- Python has an extensible Exception hierarchy
- `try / except` blocks may be used to conditionally execute code and gracefully handle the exceptions
- `try / finally` blocks allow for blocks of code to be executed whether the code in the try block fails or not
- `try / else` blocks may be used to execute code if the code in the try block does not fail
- Custom Exceptions classes may be defined by inheriting from the Exception class
- The `with` statement may be used with objects that support the Context Manager protocol to guarantee code executes when a suite or expression is entered and exited

Module 13

Using Regular Expressions in Python

Module Goals and Objectives

Major Goals

Understand the purpose of Regular Expressions

Understand how to use the `re` Module

Specific Objectives

On completion of this module, students will be able to:

Write Regular Expressions to search and replace data

Import the re module

Use pattern matching methods

Use pattern substitution methods

Parse files and objects using Regular Expressions

Use backreferences to capture the data that a pattern matched

Section 13–1

Regular Expressions Overview

About Regular Expressions

- **Regular expressions are an extremely powerful tool for manipulating text and data**
- **Regular expressions provide an infrastructure for advanced text pattern matching, extraction, and/or search-and-replace functionality**
- **They are now standard features in a wide range of languages and popular tools, including Python, Perl, Java, .NET languages, PHP, and MySQL**
- **Characters in a regular expression are comprised of two types of characters**
  - Metacharacters which have a special meaning in the Regular Expression Engine
  - Regular Characters which match a literal character
- **A Regular Expression processing engine compares the `re` pattern to a string and returns the results of the comparison**
- **In Python, Regular Expressions are exposed through the `re` module**

The `re` Module

- **The `re` module is the standard Regular Expression matching interface in Python**
  - To use the `re` module it must be imported
- **The `re` module is used to perform Regular Expression pattern matching and pattern replacements in strings**
- **The `re` module supports pattern matching against Unicode and byte-strings**
- **The `re` module supplies module level functions that may be used to perform pattern matching or pattern substitution or pattern manipulation**
- **The `re` module also provides a method to create a Regular Expression Object**
  - The Regular Expression Object inherits many of the module-level functions that may be used for pattern matching or pattern substitution

Section 13–2

Regular Expression Syntax

Regular Expression Syntax

- **Regular Expressions (`re`) are string patterns comprised of metacharacters and literal characters that are used to match Unicode strings or byte-strings**

- **Metacharacters are characters that have a special meaning to a computer program**
  - A metacharacter's literal meaning is discarded in favor of their special meaning
  - If you want to use the literal meaning of a metacharacter, the metacharacter must be escaped with the escape character \

- **Regular Expressions' metacharacter sequences may be broken down into a number of categories**
  - Metacharacters
  - Character Classes
  - Quantifiers
  - Assertions
  - Back References

- **Regular Expression patterns are created by concatenating Regular Expression metacharacters with string literals**
  - The resulting pattern strings are used to compare against text strings

Matching Characters

- **The simplest form of pattern matching involves matching characters or strings**

- **Most characters or strings will match themselves**
  - Special metacharacters will not match themselves
  - If you want to match the literal meaning of a metacharacter, the metacharacter must be escaped

Example

In this example we import the `re` module and use the module-level `search()` method to compare a literal string to a pattern comprised of a literal string. We also create a compiled Regular Expression (`re`) object and compare the literal string to the `re` object. The `search()` method will be discussed in the next section

```
import re

s = 'test'
t = re.compile('test')
if(re.search('test',s)):
    print('test string found')
else:
    print('test string not found')

if(t.search(s)):
    print('test string found')
else:
    print('test string not found')

Output
test string found
test string found
```

Common Metacharacter Sequences

- **The following table lists some of the more common metacharacter sequences**
  - Many of these will be further explained in this section

Sequence	Description
`.`	Matches any character. (Also matches newline character if DOTALL, S flag is specified)
`C`	Any literal character, matches itself
`\`	Escapes special characters and introduces special metacharacter sequences. Because of Python string rules, write as `\\` or as a raw string `r'\\'`
`\\`	Matches a literal `\`. Because of Python string rules, write as `\\\\` in pattern or `r'\\'`
`\int`	Matches the contents of the matched group int. Groups are numbered from 1 to 99 . (\d+) \1 matches 16 16
`P|P`	Alternative pattern: Matches left P or right P
`[...]`	Defines a character class set
`[^...]`	Defines a negative character class set
`(...)`	Match enclosed regex and save as subgroup. Referenced in regex through \int (\1,\2,etc..) or through the MatchObject `group()` method

Example

```
import re
s = '12 12'
p = r'(\d+) \1'
print(re.search(p,s).group()) #12 12
```

Character Classes

- **Character classes are lists of characters that may be found at a single character position**
  - Essentially a character class creates a character based membership set test
- **Character classes are created by placing characters into a set of square brackets `[]`**

Example

```
'h[aiu]t'
```

- **Negative character classes are created by placing a caret operator ^ at the beginning of the character class sequence in the square brackets `[]`**
  - Negative character classes create a list of characters that should not be found at a character position

Example

```
'1[^35]7'
```

- **A range of characters in a character class may be created by using the hyphen operator –**

example

```
'1[2-4]7'
```

Example: Search with Character Classes

Example

```
import re
def search(s):
    if(re.search('h[aiu]t',s)):
        print('test string', s ,  'found')
    else:
        print('test string' , s, 'not found')

testlist = ['hat', 'hit', 'hut', 'that', 'bat', 'cat']
for word in testlist:
    search(word)
```

Output

```
test string hat found
test string hit found
test string hut found
test string that found
test string bat not found
test string cat not found
```

Example: Search with Compiled Character Classes

Example

```
import re
pat = re.compile('ID-1[356]9')
idlist = ['ID-159','ID-169','ID-129','ID-139','ID-168']

def compsearch(p,s):
    """Pass a Compiled re and a Search string"""
    if(p.search(s)):
        print('test string', s ,  'found')
    else:
        print('test string' , s, 'not found')


for id in idlist:
    compsearch(pat, id)
```

Output

```
test string ID-159 found
test string ID-169 found
test string ID-129 not found
test string ID-139 found
test string ID-168 not found
```

Shorthand Escape Sequences for Character Classes

- **The regular expression language supports a number of metacharacters that are exposed as escape sequences that may be used to represent character classes**

Sequence	Description
`\d`	Matches any digit. Equivalent to `[0-9]`
`\D`	Matches any non digit. Equivalent to `[^0-9]`
`\w`	Matches any word character. Equivalent to `[a-zA-Z0-9_]`
`\W`	Matches any non-word character. Equivalent to `[^a-zA-Z0-9_]`
`\s`	Matches any whitespace character. Equivalent to `[ \t\n\r\f\v]`
`\S`	Matches any non-whitespace character. Equivalent to `[ \t\n\r\f\v]`

Example

```
import re
s = '90210'
p = '(\d\d\d\d\d)'
print(re.search(p,s).group()) #90210
```

Pattern Repetition with Quantifiers

- **Often it will be necessary to require a specified number of repetitions in a pattern**
  - Pattern repetition may be facilitated by using the Quantifier Metacharacters

Sequence	Description
`*`	Match preceding element 0 or more times. (match as many as possible)
`+`	Match preceding element 1 or more times. (match as many as possible)
`?`	Match preceding element 0 or 1 times
`{n}`	Match preceding element exactly `n` times
`{m,n}`	Match preceding element from `m` to `n` times
`(*|+|?|{n}{m,n})?`	Same as previous explanations, except `?` indicates to match as few times as possible. Creates a non-greedy match

Example

```
import re
s = '90210'
p = '(\d{5})'
print(re.search(p,s).group()) #90210
```

Specify Boundaries using Assertions

- **Assertions are used to match locations or boundaries in a string, as opposed to matching strings**
  - Assertions are used to specify additional conditions to the pattern

Sequence	Description
^	Matches start of string (or start of every line when MULTILINE,M flag is set)
$	Matches end of string (or end of every line when MULTILINE,M flag is set)
\A	Matches only at the start of a string
\Z	Matches only at the end of a string
\b	Matches empty string at a word boundary
\B	Matches empty string not at a word boundary

Example: Specify Boundaries using Assertions

```
import re
slist = ['90210','8675309','1000000','42'
         ,'432520032744898560000', '10036']

p = re.compile(r'\d{5}')
p2 = re.compile(r'\b\d{5}\b')

def compsearch(p,s):
    """Pass a Compiled re and a Search string"""
    if(p.search(s)):
        print('test string', s ,  'found')
    else:
        print('test string' , s, 'not found')

for num in slist:
    compsearch(p,num)
```

Output

```
test string 90210 found
test string 8675309 found
test string 1000000 found
test string 42 not found
test string 432520032744898560000 found
test string 10036 found
```

```
for num in slist:
    compsearch(p2,num)
```

Output

```
test string 90210 found
test string 8675309 not found
test string 1000000 not found
test string 42 not found
test string 432520032744898560000 not found
test string 10036 found
```

Extended Regular Expression Notation

- **Python also supports a robust extended Regular Expression syntax which includes the ability to name pattern backreferences, facilitate positive and negative lookahead and lookbehinds, and the ability to embed re flags into the pattern itself**

Sequence	Description	Example
`(?iLmsux)`	Embed one or more special "flags" parameters within the re itself (vs. via function/method)	`(?x), (?im)`
`(?:...)`	Signifies a group whose match is not saved	`(?:\w+\.)*`
`(?P<name>...)`	Like a regular group match, only identified with name rather than a numeric ID	`(?P<data>)`
`(?P=name)`	Matches text previously grouped by (?P<name>) in the same string	`(?P=data)`
`(?#...)`	Specifies a comment, all contents within ignored	`(?#comment)`
`(?=...)`	positive lookahead assertion: Matches if `...` comes next without consuming input string	`(?=.com)`
`(?!...)`	Negative lookahead assertion: Matches if `...` doesn't come next without consuming input	`(?!.net)`
`(?<=...)`	Positive lookbehind assertion: Matches if `...` comes prior without consuming input string	`(?<=800-)`
`(?<!...)`	Negative lookbehind assertion: Matches if `...` doesn't come prior without consuming input	`(?<!192\.168\.)`
`(?(id/name)Y|N)`	Conditional match of regex `Y` if group with given id or name exists, else `N`; `|N` is optional	`(?(1)y | x)`

Section 13–3

Using Regular Expressions with the `re` Module

About the `re` Module

- **The `re` module exposes not only a Regular Expression processing engine and the Regular Expression syntax, but also a number of functions that may be used to manipulate or search data with regular expressions**

- **The `re` module exposes module-level functions that may be used to compare arbitrary strings to arbitrary patterns**

- **The `re` module also exposes a method that may be used to create a compiled Regular Expression object**
  - The compiled Regular Expression object inherits a number of the module-level functions and properties

- **Many of the matching methods return a match object**
  - Match objects contain metadata about the sequences matched by the patterns

- **The `re` module must be imported to use**

Syntax

```
import re
```

Regular Expression Modifiers

- **The `compile()`, `match()`, `search()`, `findall()` and `finditer()` functions all take Regular Expression modifiers as part of their arguments**

- **Regular Expression modifiers alter the way the pattern is used in the Regular Expression processing engine**

- **The following table enumerates the Regular Expression modifiers that may be supplied as part of the aforementioned functions' `flag` argument**

Modifier	Description
`A or ASCII or (?a)`	Performs 8-bit, ASCII-only matching (Python 3.x)
`I or IGNORECASE or (?i)`	Performs case-insensitive matching
`L or LOCALE or (?L)`	Makes \w,\W, \b,\B,\s,\S,\d, and \D dependent on the current locale. (Default for Python 3.x is Unicode)
`M or MULTILINE or (?m)`	Changes boundary operators to match each new line, not entire string
`S or DOTALL or (?s)`	Changes meaning of . to add newlines to its match list
`U or UNICODE or (?u)`	Makes \w,\W, \b,\B,\s,\S,\d, and \D dependent on Unicode character properties.(New to Python 2.x)
`X or VERBOSE or (?x)`	Ignores unescaped whitespace and comments in the pattern

About Matching and Searching

- **Python's `re` module supports two different matching operations based on regular expressions**
- **The `match()` method checks for a match only at the beginning of a string**
- **The `search()` operation checks for a match anywhere inside of a string**
  - This behavior is more akin to the way Perl or PHP or any PCRE performs pattern matching
- **Regular Expression objects created using the `compile()` method have a `match()` method and a search `method()`**
- **The `match()` and the `search()` methods may have their behaviors altered by setting Regular Expression modifiers as part of their `flag` arguments**

Creating Compiled Regular Expression Objects

- **Compiled regular expression objects are created using the `re.compile()` function**

- **Compiled regular expression objects are useful when a pattern needs to be used many times**

Syntax

```
re.compile(pattern [,flags])
```

  - Compiles a regular expression pattern string into a regular expression pattern object for later matching

- **The flags argument accepts any top level pattern match modifiers**

  - Multiple modifiers may be set by using the bitwise | operator

Example: Compiled Regular Expression Search with Flags

Example

```
import re

def idsearch(p,s):
    if(p.search(s)):
        print('test string', s ,  'found')
    else:
        print('test string' , s, 'not found')

pat = re.compile('ID-1[356]9')
pati = re.compile('ID-1[356]9', re.I) # Case insensitive

idlist = ['ID-159','Id-169','ID-139','iD-139','ID-169']

for id in idlist:
    idsearch(pat, id)
```

Output

```
test string ID-159 found
test string Id-169 not found
test string ID-139 found
test string iD-139 not found
test string ID-169 found

for id in idlist:
    idsearch(pati, id)
```

Output

```
test string ID-159 found
test string Id-169 found
test string ID-139 found
test string iD-139 found
test string ID-169 found
```

Regular Expression Object Attributes

- **Compiled regular expression objects created by the `re.compile()` function have a number of methods and attributes**

- **The following table enumerates attributes of a regular expression object**

Attribute	Description
`reobj.flags`	The flags argument specified when the object was compiled. Returned as an integer. 0 indicates no flags were specified
`reobj.groupindex`	A dictionary mapping of `{group-name: group-number}` specified by group names defined by `r'(?P<id>)'`
`reobj.pattern`	The pattern string from which the regular expression was compiled

Example

```
import re
def objmetadata(p,s):
    if(p.search(s)):
        print('test string', s ,  'found')
        print('Flag',p.flags,'GroupIndex',
p.groupindex,'Pattern',p.pattern)
    else:
        print('test string' , s, 'not found')
pat = re.compile('ID-1[356]9', re.I)
objmetadata(pat, 'ID-159')
#test string ID-159 found
#Flag 34 GroupIndex {} Pattern ID-1[356]9
```

Regular Expression Object Methods

- **Regular expression object methods are analogous to their module-level counterparts with respect to functionality**

- **Regular expression objects support the following methods and will be discussed with their module-level counterparts in this section**
  - `reobj.findall(string[,pos[,endpos]])`
  - `reobj.finditer (string[,pos[,endpos]])`
  - `reobj.match(string[,pos[,endpos]])`
  - `reobj.search(string[,pos[,endpos]])`
  - `reobj.split(string[,maxsplit=0])`
  - `reobj.sub(repl, string[,count = 0])`
  - `reobj.subn(repl, string[,count = 0])`

Example

```
import re
pat = re.compile('(ID-1[356]9)', re.I)
print(pat.search('ID-159').group())

##ID-159
```

Searching with `re.search()`

Syntax

```
#Module level function

re.search(pattern, string[,flags])
```

Syntax

```
#Compiled object function

reobj = re.compile(pattern [,flags])
reobj.search(string[,pos[,endpos]])
```

- **The `re.search()` and `reobj.search()` functions, search a string for the first match of a pattern**
  - The flags argument for `re.search()` takes the same arguments as the `re.compile()` function
- **`reobj.search()` optionally searches for a match between the `pos` and `endpos` arguments**
  - `pos` and `endpos` are integer values that correlate to positions in the search string
- **Both methods return a `MatchObject` if a match is found**
- **Both methods return `None` if no match was found**

Example: Searching with `re.search()`

Example

```
import re

s = 'I aim to misbehave'
p = 'behave'

pat = re.compile(p , re.I)

if re.search(p, s):
    print('Pattern', p, 'found in string', s)
else:
    print('Pattern', p, 'not found in string', s)
#Pattern behave found in string I aim to misbehave

if pat.search(s):
    print('Pattern', p, 'found in string', s)
else:
    print('Pattern', p, 'not found in string', s)
#Pattern behave found in string I aim to misbehave
```

Searching with `re.match()`

Syntax

```
#Module level function

re.match(pattern, string[,flags])
```

Syntax

```
#Compiled object function

reobj = re.compile(pattern [,flags])
reobj.match(string[,pos[,endpos]])
```

- **The `re.match()` and `reobj.match()` functions search at the beginning of a string for the first match of a pattern**
  - The flags argument for `re.match()` takes the same arguments as the `re.compile()` function
- **`reobj.match()` optionally searches for a match between the `pos` and `endpos` arguments**
  - `pos` and `endpos` are integer values that correlate to positions in the search string
- **Both methods return a `MatchObject` if a match is found**
- **Both methods return `None` if no match was found**

Example: Searching with `re.match()`

Example

```
import re

def mymatch(p,s):
    if re.match(p, s, re.I):
        print('Pattern', p, 'found in string', s)
    else:
     print('Pattern', p, 'not found in string', s)

def mycompmatch(pat, s):
    if pat.match(s):
        print('Pattern', p, 'found in string', s)
    else:
        print('Pattern', p, 'not found in string', s)

s = 'I aim to misbehave'
s2 = 'Behave, Yeah Baby!'
p = 'behave'
pat = re.compile(p , re.I)



mymatch(p,s)
#Pattern behave not found in string I aim to misbehave

mymatch(p,s2)
#Pattern behave found in string Behave, Yeah Baby!

mycompmatch(pat, s)
#Pattern behave not found in string I aim to misbehave

mycompmatch(pat, s2)
#Pattern behave found in string Behave, Yeah Baby!
```

Match Objects

- **Match objects are returned by successful `match()` and `search()` operations**
  - Match objects (`m`)export the following attributes

Attribute	Description
`m.pos, m.endpos`	Values of `pos` and `endpos` passed to a `search()` or a `match()`
`m.re`	`re` object attached to the `search()` or `match()` call that produced the results
`m.string`	String argument that was passed to the `search()` or a `match()`
`m.group([g1,g2,...])`	Returns substrings that were matched by parenthesized groups in the pattern. Accepts 0 or more group numbers Returns the matched substring if 1 argument is passed, returns a tuple of substrings if more than 1 argument is passed
`m.groups()`	Returns a tuple of all groups of the match; groups not participating in the match default to `None`
`m.groupdict()`	Returns a dictionary containing all the named subgroups in the match
`m.start([group]),` `m.end([group])`	Indexes the `start` and the `end` of the substring matched by group or the entire string if there is no group.
`m.span([group])`	Returns the tuple `(start(group),end(group))`
`m.expand(template)`	Returns the string obtained by performing backslash substitution on the template string template as performed by the sub method. Escape sequences are converted to their appropriate characters, and back references are replaced by the corresponding group

Example: Match Objects

Example

```
import re
pat = re.compile('ID-(1[356]9)')
idlist = ['ID-159','ID-169','ID-129','ID-139','ID-168']

def compsearch(p,s):
    """Pass a Compiled re and a Search string"""
    mo = p.search(s)
    if(mo):
        print('test string', s ,  'found')
        print('Complete Match', mo.group(0) , 'Individual
             ID', mo.group(1) )
        print('From string',mo.string,'Character
             Positions', mo.span())
    else:
        print('test string' , s, 'not found')


for id in idlist:
    compsearch(pat, id)
```

Output

```
Test string ID-159 found
Complete Match ID-159 Individual ID 159
From string ID-159 Character Positions (0, 6)
Test string ID-169 found
Complete Match ID-169 Individual ID 169
From string ID-169 Character Positions (0, 6)
Test string ID-129 not found
Test string ID-139 found
Complete Match ID-139 Individual ID 139
From string ID-139 Character Positions (0, 6)
Test string ID-168 not found
```

Example: Match Objects and Named Groups

Example

In this example we are going to use a basic e-mail address pattern and we are going to remember the user, domain, and extension components of the email address in named groups

```
import re


patt =
'(?P<user>\w+)@(?P<domain>(?:\w+\.)?\w+)\.(?P<ext>\w{2,4})'

mo = re.search(patt, 'user@sub.domain.com')

if mo is not None:
    print('User portion of the email address is
    ',mo.group('user'))
    print('Domain portion of the email address is
    ',mo.group('domain'))
    print('Extension portion of the email address is
    ',mo.group('ext'))
```

Output

```
User portion of the email address is  user
Domain portion of the email address is  sub.domain
Extension portion of the email address is  com
```

Substitution with `re.sub()`

Syntax

```
#Module level function
re.sub(pattern, replacement, string [,count = 0])
```

Syntax

```
#Compiled object function

reobj = re.compile(pattern [,flags])
reobj.sub(replacement, string [,count = 0])
```

- **`re.sub()` and `reobj.sub()` replace the leftmost non-overlapping occurrences of the `pattern` in the `string` using the `replacement` argument**
  - The `replacement` argument may be a string or a function
    - If it is a function, then the function is called with a `MatchObject` and should return a replacement string
  - If `replacement` is a string, then pattern back references '`\1,etc...`' or group names '`\g<name>` 'may be used to refer to groups in the pattern
  - `count` is the number of replacements to make
    - By default all occurrences are replaced

Example: Substitution with `re.sub()`

Example

```
import re

phoneimport = '123-867-5309 # Tommy Tutone has got your number'

phone = re.sub(r'#.*$', '',phoneimport)

print('Before', phoneimport, 'After', phone)

#Before 123-867-5309 # Tommy Tutone has got your number
#After 123-867-5309

phonedb = re.sub(r'\D', '' , phone)

print('Before', phone, 'After', phonedb)
#Before 123-867-5309  After 1238675309
```

Example

```
id1 = 'ID=132+47=99'
id2 = 'ID+125=102-93'

newid1 = re.sub('=|\+', '-', id1)
newid2 = re.sub('=|\+', '-', id2)

print(id1, newid1) #ID=132+47=99 ID-132-47-99
print(id2, newid2) #ID+125=102-93 ID-125-102-93
```

Substitution with `re.subn()`

Syntax

```
#Module level function
re.subn(pattern, replacement, string [,count = 0])
```

Syntax

```
#Compiled object function

reobj = re.compile(pattern [,flags])
reobj.subn(replacement, string [,count = 0])
```

- **The `re.subn()` method works the same way as the `re.sub()` function except `re.subn()` returns a tuple**
  - The first item in the tuple is the new string
  - The second item in the tuple is the number of replacements made to the string

Example: Substitution with `re.subn()`

Example

```
import re


id1 = 'ID=132+47=99'
id2 = 'ID+125=102-93'

newid1 = re.subn('=|\+', '-', id1)
newid2 = re.subn('=|\+', '-', id2)

print(id1, ' is now', newid1[0] , 'after making' ,
newid1[1], 'replacements')

#ID=132+47=99  is now ID-132-47-99 after making 3
#replacements

print(id2, ' is now', newid2[0] , 'after making' ,
newid2[1], 'replacements')

#ID+125=102-93  is now ID-125-102-93 after making 2
#replacements
```

Splitting Strings with `re.split()`

Syntax

```
#Module level function
re.split(pattern, string [,maxsplit = 0])
```

Syntax

```
#Compiled object function

reobj = re.compile(pattern [,flags])
reobj.split(string [,maxsplit = 0])
```

- **`re.split()` and `reobj.split()` split strings by the occurrences of pattern**
- **The methods returns a list of strings including the text matched by any groups in the pattern**
- **The `maxsplit` argument is the maximum number of splits to execute**
  - By default all possible splits are executed

Example: Splitting strings with `re.split()`

Example

In this example we are going to parse a csv into a 2 dimensional array. We are then going to parse the 2 dimensional array and perform some text transformations on the contents. We will then write the contents of the 2 dimensional array to another csv. Because that is the way we doose it.

```
import re

ff = open('firefly.txt','r')
ffout = open('fireflyout.txt','w')
serenity = []
for crew in ff:
    cinfo = re.split(',|\t',crew)
    serenity.append(cinfo)
ff.close()

for crew in serenity:
    i = 0
    clen = len(crew)
    for cinfo in crew:
        i+=1
        ffout.write(cinfo.lower())
        if i < clen:
            ffout.write(',')

ffout.close()
```

firefly.txt	fireflyout.txt
Nathan Fillion,Malcolm Reynolds,Mal	nathan fillion,malcolm reynolds,mal
Gina Torres,Zoe Alleyne Washburne,Zoe	gina torres,zoe alleyne washburne,zoe
Alan Tudyk,Hoban Washburne,Wash	alan tudyk,hoban washburne,wash
Morena Baccarin,Inara Serra,Inara	morena baccarin,inara serra,inara
Adam Baldwin,Jayne Cobb,Jayne	adam baldwin,jayne cobb,jayne
Jewel Staite,Kaywinnet Lee Frye,Kaylee	jewel staite,kaywinnet lee frye,kaylee
Sean Maher,Dr. Simon Tam,Simon	sean maher,dr. simon tam,simon
Summer Glau,River Tam, River	summer glau,river tam, river
Ron Glass,Derrial Book,Shepherd	ron glass,derrial book,shepherd

Module Summary

- **Regular expressions are an extremely powerful tool for manipulating text and data**
- **Regular Expressions are exposed in Python through the `re` Module**
- **Python supports an extremely robust Regular Expression syntax**
- **Methods such as `search()` and `match()` may be used to find data based on patterns**
- **Methods such as `sub()` may be used to substitute found data with new data**
- **The `re.split()` method is much more flexible than Python's built-in `split()` method**

Module 14

Database Programming with Python

Module Goals and Objectives

Major Goals

Understand the DB API

Understand common SQL Statements

Understand how to use the SQLite3 API

Specific Objectives

On completion of this module, students will be able to:

Create Disk-based and In-Memory Databases

Use Common Database APIs to interact with Databases

Use DDL statements to Create and Drop Database Objects

Use DML statements to Insert, Update and Delete Records

Create SQLite Databases

Use Cursors to interact with Recordsets

Retrieve records from a database

Issue Parameterized queries to a database

Create and Use Row Objects

Section 14–1

The DB API

The DB API

- **Python supports a common Database API called the DB API**
  - Current version is the Python Database API Specification v2.0 or PEP 249
  - `http://www.python.org/dev/peps/pep-249/`
- **The Python DB API Specification provides a standard interface from Python to databases**
  - Database modules are designed to follow this specification and may add database-specific features
  - Most database interfaces adhere to the DB API Specification
- **The Python DB API was designed to be easy to understand and implement**
- **The DB API was defined to encourage similarity between Python Modules that are used to access databases**
- **The Goals of the API are:**
  - Consistency in the development of database Modules
  - More easily understood modules
  - Portability of code across databases
  - Expanded reach of database connectivity

Database Interfaces

- **Python supports a number of Relational and Non-Relational database interfaces**

- **General Relational Database Interfaces include:**
  - IBM DB2
  - Firebird / Interbase
  - Informix
  - Ingres
  - MySQL
  - Oracle
  - PostgreSQL
  - SAP DB
  - MS SQL Server
  - MS Access
  - SQLite
  - Sybase

Non-Relational Database Interfaces

- **Non-Relational Database interfaces include**
  - Flat File / Fixed Record
    - atop
    - BerkleyDB
    - buzhug
    - Durus
    - KirbyBase
    - MetaKit
    - ZODB
  - XML Databases
    - 4Suite Server
    - Oracle/Sleepycat DB XMl
  - Graph Databases
    - Neo4j

DB API Module Interface

- **At a high level, the database API defines a set of functions and objects for:**
  - Connecting to a database server
  - Executing SQL queries
  - Obtaining results
- **Two primary objects are used to perform these tasks**
  - A Connection object that manages the connection to the database
  - A Cursor object that is used to perform queries
- **Database Interfaces that adhere to the DB API standards will implement the methods and objects described in the API**
- **The DB API requires a constructor to provide access to a database**
  - Access to the database is made available through connection objects

Connection Objects

- **To connect to a database, database modules provide a module level `connect()` function that is used to connect to a database and return a connection object**

Syntax

```
c = connect(parameters)
```

Example

```
import sqlite3
c = sqlite3.connect('test.db')
```

- **The exact parameters for the connect function depend on the database you are connecting to**
  - Typical parameters include:
    - Data source name
    - User name
    - Password
    - Hostname
    - Database name
  - Arguments may be provided as key = value pair arguments

Example

```
c =
connect(dsn="hostname:TESTDB",user="HOTTUser",password="
p@$$w3rd")
```

Connection Object Methods

- **Connection objects are designed to respond to the following methods**

- **`c.close()`**
  - Closes the connection to the database
  - Connection is unusable from this point forward in the program
    - Any operation attempted with this connection will raise an Exception
  - Closing a connection without committing changes will cause an implicit rollback

- **`c.commit()`**
  - Commit any pending transaction to the database
  - Non-transactional databases should implement this method with void functionality

- **`c.rollback()`**
  - Causes the database to roll back to the start of any pending transaction

- **`c.cursor()`**
  - Returns a cursor object

Cursor Objects

- **In order to perform any operations on the database:**
  - First create a connection to a database
  - Then call the `cursor()` method to create a Cursor object.
- **Cursor objects represent a database cursor**
  - Cursor objects have attributes and methods
- **Cursor objects are used execute database commands and to traverse through records from a result set**
- **Cursor objects are also used to manage the context of a fetch operation**

Syntax

```
cur = c.cursor()
```

Example

```
c = sqlite3.connect('test.db');
cur = c.cursor()
```

Cursor Attributes

- **Cursor objects support standard attributes**
  - description
  - rowcount
- **`cur.description()`**
  - Returns a read-only sequence of 7-item sequences
    - One sequence of 7-item sequences is returned for each column in the recordset
  - Each sequence contains information describing a single column

name	type_code
display_size	internal_size
precision	scale
null_ok	

  - Values default to None if no meaningful values are provided
- **`cur.rowcount`**
  - Read-only attribute that specifies the number of rows that the most recent `.execute*()` produced
  - -1 is returned if no `.execute*()` has been performed on the cursor or if the `rowcount` cannot be determined

Cursor Methods

- **Cursor objects are designed to respond to the following methods**

- **`cur.callproc(procname [,params])`**
  - This is an optional method that calls a stored procedure with the procname

- **`cur.close()`**
  - Closes the cursor

- **`cur.execute(operation [,param])`**
  - Prepares and executes a database operation
  - Operation may be a query or a command
  - Params may be provided as a sequence that will be bound to variables in the operation

- **`cur.fetchone()`**
  - Fetches the next row of a query result set
  - Returns None when no more data is available
    - Returns an exception if the previous `.execute*()` did not produce results or no call was issued

- **`cur.fetchall()`**
  - Fetch all rows of a query result

Section 14–2

SQL Primer

Databases and Relational Database Management Systems

- **A database is an organized collection of data**
- **The software used to manage a database are Database Management Systems**
- **There are many types of Databases**
  - Hierarchical
  - Network
  - Relational
  - Object Relational
- **The most prevalent type of Database is the Relational Database**
- **Relational Databases are based off of the Relational Data Model developed by Dr. E F Codd**
  - Dr. Codd discusses the Relational Data Model in his research paper "A Relational Model of Data for Large Shared Data Banks"
- **Codd's research paper also discussed implementing a single language that could be used to define, alter, and administer relational data objects**
  - SQL was one of the first commercial languages based on the research paper

Database Concepts and Terminology

- **Database**
  - An electronic collection of information in discreet, structured objects
- **Relational Database (RDB)**
  - A database organized as a set of formally described tables
- **Relational Database Management System (RDBMS)**
  - An application that stores, modifies, extracts data from an RDB
- **Table**
  - The basic structure of an RDB that stores data in rows (records) and columns (attributes)
- **Element**
  - The data found at the intersection of a row and a column (information)
- **Row**
  - Represents a single structured item in a database
- **Attribute / Field**
  - A column in a table

Structured Query Language (SQL)

- **SQL stands for the Structured Query Language**
  - De-facto standard for interacting with relational databases
- **SQL is a specialized programming language that is used to manage data in Relational Database Management Systems (RDBMS)**
  - SQL is a directive-based programming language based on Relational Algebra and Tuple-based Calculus
    - Declarative languages are languages in which the expected result or operation is given without the specific details about how to accomplish the task
- **Most database management systems implement some form of SQL**
- **In 1986 the American National Standards Institute (ANSI) standardized SQL**
  - Since then SQL has gone through many different revisions
- **Some DBMSs implement vendor-specific versions of SQL**
  - MS SQL Server uses Transact SQL (T-SQL)
  - Oracle DBMSs use Procedural Language SQL (PL/SQL)
  - MySQL uses SQL/Procedural Stored Module (SQL/PSM)

SQL Language Command Categories

- **SQL is the language for interacting with databases**
- **SQL consists of a number of commands with options that allow a developer to carry out operations in a database**
- **The SQL Language is subdivided into different categories to organize the commands**
  - Data Definition Language (DDL)
  - Data Manipulation Language (DML)
  - Data Query Language (DQL)
  - Data Control Language (DCL)
- **The Data Definition Language supports commands to Create or Modify a database structure**
- **The Data Manipulation Language supports commands to manipulate the data stored in a database**
- **The Data Query Language supports commands used for querying or selecting a subset of data from a database**
- **The Data Control Language is used for controlling access to data in the database and the database itself**

Data Definition Language (DDL)

- **The DDL is used to:**
  - Create database objects
  - Modify database objects
  - Delete Database objects
- **The 3 primary SQL commands in the DDL are:**
  - CREATE
    - Used to create database objects
  - ALTER
    - Used to modify database objects
  - DROP
    - Used to delete database objects

Data Manipulation Language (DML)

- **The DML is used to**
  - Add records to tables
  - Modify existing records in tables
  - Delete records from tables
- **The primary SQL commands in the DML are:**
  - INSERT
    - Used to add records
  - UPDATE
    - Used to modify records
  - DELETE
    - Used to remove records

Data Query Language (DQL)

- **The Data Query Language is used to:**
  - Retrieve data from one or more tables or expressions
  - Has no persistent effects on the database
- **The Primary SQL command in the DQL is:**
  - SELECT
    - Retrieves data from one or more tables or expressions
- **The SELECT statement has the following clauses:**
  - FROM
    - Indicates the table(s) from which data is to be retrieved
  - WHERE
    - Restricts the rows returned by the query
  - GROUP BY
    - Used to group rows having common values into a smaller set of rows
    - Used in conjunction with aggregate functions
  - HAVING
    - Used to filter rows resulting from the GROUP BY clause
  - ORDER BY
    - Identifies which columns are used to sort the resulting data, and in which direction they should be sorted

Data Control Language (DCL)

- **The DCL is used to:**
  - Give database users or groups permission to interact with database objects
  - Remove existing permissions on database objects
  - Deny users or groups from interacting with database objects
- **The primary SQL commands in the DCL are:**
  - GRANT
    - Authorizes users to interact with a database object
  - REVOKE
    - Removes the authorization provided by a GRANT operation

SQL Language Syntax Rules

- **SQL Keywords are words that have special meaning in SQL**
  - SQL keywords are not case sensitive
  - Convention is to capitalize keywords to improve their visibility
- **Identifiers are names of databases and database objects**
  - Identifiers should not have spaces in their names
- **Literals are hard-coded values**
  - String literals are placed in single quotes `' '`
    - For example ... `WHERE fname = 'Jenny'`
  - Numeric literals are not placed in quotes
    - For example ... `WHERE id = 8675309`
  - Date literals may be passed as strings
- **Operators are symbols that perform mathematical operations**
  - For example, `+ - * / smooth`
    - Okay, `smooth` isn't actually an operator
- **White space is ignored**
- **SQL Statements should be terminated with a semicolon `;`**

CREATE / DROP Table Statements

- **CREATE table statements are used to create tables to store data**

- **CREATE table statements specify**
  - Name of the table
  - Field names and number of fields
  - Data types for each field
    - May also specify constraints such as integrity rules or business rules

Syntax

```
CREATE TABLE tablename(
  col1 datatype,
  col2 datatype,
  [...coln datatype]);
```

Example

```
CREATE TABLE STUDENT
       (ID INT PRIMARY KEY NOT NULL,
       NAME TEXT NOT NULL);
```

- **Tables may be dropped using the DROP table statement**

Example

```
DROP TABLE tablename;
DROP TABLE STUDENT;
```

INSERT Statement

- **INSERT statements are used to add new rows of data to a table**

- **The syntax for the INSERT statement specifies:**
  - A target table
  - An optional field list
    - If no fields are specified, values are supplied for all fields
  - A values list that matches the data types of the target fields

Syntax

```
INSERT [INTO] tablename
[(col1, col2, .... coln)]
VALUES
(val1, val2, ...valn);
```

Example

```
INSERT INTO student
(id, name)
values
(8675309, 'Jenny');
```

Example

```
INSERT student
values
(1, 'James T. Kirk');
```

`Update` Statement

- **`UPDATE` statements are used to modify existing rows in a table**

- **The syntax for the `UPDATE` statement specifies:**
  - A target table
  - Fields that will be updated
  - New values for the fields
  - `WHERE` clauses may be used to specify which rows will be updated

Syntax

```
UPDATE tablename
SET field = value,
field = value ,.....
[WHERE clause]
```

Example

```
UPDATE product
SET price = price * 1.1
```

Example

```
UPDATE student
SET name = 'Jennay!!!'
WHERE id = 8675309
```

DELETE Statement

- **DELETE statements are used to remove rows from a table**
- **The syntax for the DELETE statement specifies:**
  - A target table
  - A WHERE clause may be used to specify which rows get deleted

Syntax

```
DELETE [FROM] table
[WHERE clause]
```

Example

```
DELETE FROM productcopy
```

Example

```
DELETE FROM student
WHERE id = 8675309
```

SELECT Statement

- **The SELECT statement is used to retrieve records from the database**
  - The most commonly used SQL statement
- **SELECT statements returns rows from a target table based on a specified field list or all fields in a table**
- **The syntax for the SELECT statement specifies:**
  - A target table or tables to return rows from
  - A field list and/or predicate
  - An optional WHERE clause limits the records returned
  - An optional GROUP BY clause to summarize aggregate data into groups
  - An optional HAVING clause to limit the groups returned by the GROUP BY clause
  - An optional ORDER BY clause to sort the returned rows by fields ascending (ASC) or descending (DESC)

SYNTAX

```
SELECT [predicate] col [[as] Alias], col2 [[as] Alias],...
FROM table(s) [ JOIN table ON joinclause]
[WHERE clause]
[GROUP BY clause]
[HAVING clause]
[ORDER BY clause]
```

Example: SELECT Statement

Example

```
SELECT *
FROM student
```

Example

```
SELECT *
FROM student
WHERE id = 8675309
```

Example

```
SELECT name, id
FROM student
```

Example

```
SELECT productname, price , price * 1.1
FROM products
```

Example

```
SELECT id, name
FROM student
ORDER BY name
```

Example

```
SELECT prodname, sum(qty)
FROM orders
GROUP BY prodname
ORDER BY sum(qty) DESC
```

Section 14–3

The SQLite3 Module

The `sqlite3` Module

- **SQLite is a C library that provides lightweight disk-based database access**
  - Does not require a separate server process to execute
  - Allows the use of a nonstandard variant of the SQL programming language to access the database
- **Many applications use SQLite for internal data storage including:**
  - Adobe Air
  - Adobe Lightroom
  - Dropbox
  - Mozilla Firefox
  - Mozilla Thunderbird
  - Intuit Quickbooks
- **SQLite is also useful for prototyping databases that may be ported to Enterprise Database Servers such as Oracle**
- **The `sqlite3` is a DB-API 2.0 interface for SQLite databases and is written by Gerhard Häring**
  - Provides a DB-API 2.0 compliant SQL interface as described in PEP 249

`sqlite3` Module Objects

- **The `sqlite3` module creates and uses 3 major types of objects**
  - Connection Objects
  - Cursor Objects
  - Row Objects
- **Connection objects represent the database**
- **Cursor objects are used to perform SQL commands and represent recordsets**
- **Row objects represent a row of data returned from a Cursor objects**
  - Provide index-based and case-insensitive name-based access to columns
  - Very low memory overhead

Importing The `sqlite3` Module

- **To import the `sqlite3` module use the Python `import` method**

Syntax

```
import sqlite3
import sqlite3 as alias
```

- **The `sqlite3` module supports a number of module level functions and constants**

- **`sqlite3.version`**
  - Returns the version number of the `sqlite3` module as a string

- **`sqlite3.version_info`**
  - Returns the version number of the `sqlite3` module as a tuple of integers

- **`sqlite3.sqlite_version`**
  - Returns the version number of the run-time `SQLite` library as a string

- **`sqlite3.sqlite_version_number`**
  - Returns the version number of the run-time `SQLite` library as a tuple of integers

Example: Importing the sqlite3 module

Example

```
import sqlite3 as s

print('sqlite3 module version', s.version)
print('sqlite3 module version info',s.version_info)
print('sqlite3 version', s.sqlite_version)
print('sqlite3 version info', s.sqlite_version_info)
```

Output

```
sqlite3 module version 2.6.0
sqlite3 module version info (2, 6, 0)
sqlite3 version 3.7.12
sqlite3 version info (3, 7, 12)
```

Using the `sqlite3` Module to Create a Connection Object

- **The `sqlite3.connect()` method may be used to connect to a disk-based database or an in-memory database**

- **If the database path supplied to the `connect()` method does not exist, the `sqlite3` module will create the database**

Syntax

```
c = sqlite3.connect(database [, timeout, detect_types,
isolation_level, check_same_thread, factory,
cached_statements])
```

- **Opens a connection and returns a connection object**
  - The database argument should be the path to an existing database, or to a database you want to create
  - The database argument may also be `":memory:"` which will create an in-memory database

Example

```
import sqlite3 as s

c = s.connect("student.db")
print(c.total_changes)    # 0
c.close()
c = s.connect(":memory:")
print(c.total_changes)    # 0
c.close
```

Setting `connect()` method parameters: `timeout` and `detect_types`

- **SQLite databases are locked until transactions running against them are committed. The `timeout` argument specifies how long the connection should wait for a lock before raising an exception**
  - The default `timeout` parameter is 5 seconds
- **The `detect_types` parameter may be used to support other data types**
  - SQLite natively supports the TEXT, INTEGER, REAL, BLOB and NULL data types
  - Support must be manually added to support other data types
- **The `detect_types` parameter supports the values of:**
  - 0
    - No type detection
  - PARSE_DECLTYPES
    - Parses declared data type for each column. Checks converter dictionary for custom converter function
  - PARSE_COLNAMES
    - Parses column names for [mytype] and checks the converter dictionary for the appropriate converter function
  - Using converter functions is beyond the scope of this course

Controlling Transactions with `connect()` method parameters: `isolation_level`

- **The `isolation_level` parameter sets the current isolation level for the transaction**
  - Isolation levels include
    * `None`, which is the default isolation level and `autocommit` mode
    * `DEFERRED`
    * `IMMEDIATE`
    * `EXCLUSIVE`
  - In `autocommit` mode, all changes to the database are committed as soon as all operations associated with the current database connection complete

- **The "`BEGIN TRANSACTION`" statement is used to take SQLite out of `autocommit` mode**
  - The default transaction behavior is `DEFERRED` and stipulates that no locks are acquired on the database until the database is first accessed
  - The `IMMEDIATE` mode acquires `RESERVED` locks on all databases as soon as a `BEGIN` command is executed, without waiting for the database to be used
    * `RESERVED` locks indicate the database may be written to at some point but is currently in read mode
  - An `EXCLUSIVE` transaction causes `EXCLUSIVE` locks to be acquired on all databases

`connect()` method parameters: `check_same_thread,factory, cached_statements`

- **`sqlite3's` default behavior is to prohibit the usage of a single connection in more than one thread**
  - Intended to facilitate interoperability with older versions of `SQLite` that did not support multithreaded operation under various circumstances
  - Setting the `check_same_thread` parameter to `false` disables this check
- **`sqlite3` uses its `Connection` class to create connections**
  - Custom connection classes may be made by subclassing the `Connection` class
  - Pass the name of your custom `Connection` class to the `factory` parameter to make the `connect()` method use your custom `Connection` class instead of the built-in `Connection` class
- **`sqlite3` uses a statement cache to avoid overhead caused by parsing SQL statements**
  - The default number of statements cached is 100
  - Set the `cached_statements` parameter to an integer value to manually set the number of statements cached for the connection

`sqlite3` Connection Object Attributes

- **Connection objects represent connections to the database and are created using the `.connect()` method**
- **Connection objects support a number of attributes and methods**
- **`c.isolation_level`**
  - Sets or gets the current isolation level
- **`c.in_transaction`**
  - Read-only, True if a transaction is active, False otherwise
    - New in Python 3.2
- **`c.total_changes`**
  - Returns the total number of database rows that have been modified, inserted, or deleted since the connection was opened

Example: Working with Connection Attributes

Example

```
import sqlite3
def main():
        db = sqlite3.connect('test.db')
        db.isolation_level = "EXCLUSIVE"
        print(db.isolation_level)

         db.execute('drop table if exists test')
        db.execute('create table test (t1 text, i1 int)')
        db.execute('insert into test (t1, i1) values (? ,
         ?)' , ('One',1))
        db.execute('insert into test (t1, i1) values (? ,
         ?)' , ('Two',2))
        db.execute('insert into test (t1, i1) values (? ,
         ?)' , ('Three',3))
        db.execute('insert into test (t1, i1) values (? ,
         ?)' , ('Four',4))
        db.commit()

         print(db.total_changes)

        cursor = db.execute('select * from test order by
         t1')

        for row in cursor:
            for field in row:
                print(field, end = ' ')
            print()
if __name__ == '__main__':main()
```

Output

```
EXCLUSIVE
4
Four 4 One 1 Three 3 Two 2
```

Connection Object Methods for Managing Connections and Transactions

- **Along with the `.connect()` method, the `sqlite3` module includes methods to manage the connection and transactions**
  - c.close()
  - c.commit()
  - c.rollback()

- **`c.close()`**
  - Closes the database connection
  - Does not automatically call the `commit()` method
    - `commit()` changes to the database before you `close()` the connection, or else you will lose your changes

- **`c.commit()`**
  - The `commit()` method commits the current transaction in the database
  - Changes to the database are not visible to other transactions unless the `commit()` method is called.

- **`c.rollback()`**
  - This method rolls back any changes to the database since the last `commit()` call

Example: Managing Connections and Transactions

Example

In this example we are going to use try/except/finally error handling to manage our transaction. We will try to connect to the database, create a cursor and insert a record, and commit the transaction. If an exception is thrown we will rollback the transaction, otherwise we will close the connection.

```
import sqlite3 as s
import sys

try:
    c = s.connect('test.db')
    rs = c.cursor()

    c.execute('insert into test (t1, i1) values (? , ?)' ,
     ('Fifth',5))
    c.commit()

except s.Error as e:
    if c:
        c.rollback()

    print('Error {0}'.format(e.args[0]))
    sys.exit(1)
finally:
    if c:
        c.close()
```

Connection Object Methods for Creating Cursors

- **The `sqlite3` module's connection object supports a number of nonstandard methods to create intermediate cursors**
  - These methods create immediate cursors by indirectly calling the `cursor()` method, then call the matching cursor method
  - These methods are useful shortcuts when long term management of a cursor is not necessary such as in the case of executing an `insert`, `update` or `delete` statement

- **`c.execute(sql [,parameters])`**
  - This method creates an intermediate cursor and calls the cursor's `execute()` method with any optional parameters supplied

- **`c.executemany(sql [,parameters])`**
  - This method creates an intermediate cursor then calls the cursor's `executemany()` method with the parameters given

- **`c.executescript(sql_script)`**
  - This method creates an intermediate cursor then calls the cursors `executescript()` method with the parameters given

- **The `execute()`, `executemany()` and `executescript()` will be discussed further during the Cursor discussion**

Example: Use the `execute()` method to Create a Table and Insert Records

Example

In this example we will use the `execute()` method to create a table and insert a record

```
import sqlite3 as s

c = s.connect('student.db')

print("Opened database student.db successfully")

c.execute('''CREATE TABLE STUDENT
       (ID INT PRIMARY KEY     NOT NULL,
       NAME           TEXT    NOT NULL);
       ''')

print("Table Student created successfully")

c.execute("INSERT INTO STUDENT (ID,NAME) VALUES (1, 'Gary
');")
c.commit()
print("Record created successfully")
c.close()
```

Output

```
Opened database student.db successfully
Table Student created successfully
Record created successfully
```

Example: Use the `executescript()` method to Create a Table and Insert Records

Example

In this example we will use the connection objects `executescript()` method to drop a table if the table exists, create a table and insert 3 records into the table.

```
import sqlite3 as s

c = s.connect('student2.db')

c.executescript('''
        DROP TABLE IF EXISTS STUDENT;
        CREATE TABLE STUDENT
       (ID INT PRIMARY KEY     NOT NULL,
        NAME TEXT NOT NULL);
        INSERT INTO STUDENT VALUES(1,'Gary');
        INSERT INTO STUDENT VALUES(2,'Roland');
        INSERT INTO STUDENT VALUES(3,'Dana');

        ''')
c.commit()
print(c.total_changes)
c.close()
```

Output

```
3
```

Connection Object `iterdump()` Method

- **It may be useful to dump the database as `SQL` commands formatted as text**
  - This would allow the developer to have a text-based copy of the database structure
  - Useful for migrating the database, or rebuilding the database, or building the database on another system
- **The `sqlite3` module provides a connection object method called `iterdump()` that dumps the database structure as text formatted `SQL` statements**
  - The output is equivalent to the `sqlite3` shell `.dump` command
- **`c.iterdump()`**
  - Returns an iterator to dump the database in a SQL Text format
  - Very useful to save an in-memory database for later restoration

Example: Dump a database in text format

Example

```
import sqlite3 as s
import os

c = s.connect("test.db")
with open('testdump.sql', 'w') as f :
    for line in c.iterdump():
        f.write('%s\n' % line)
```

Output

```
BEGIN TRANSACTION;
CREATE TABLE test (t1 text, i1 int);
INSERT INTO "test" VALUES('One',1);
INSERT INTO "test" VALUES('Two',2);
INSERT INTO "test" VALUES('Three',3);
INSERT INTO "test" VALUES('Four',4);
COMMIT;
```

About Cursor Objects

- **Per the DB API 2.0 specification, the `sqlite3` module exposes a cursor object that may be created from a connection object**
- **Cursor objects are used to**
  - Execute SQL statements against the database
  - Fetch records from the recordset
  - Return metadata about the recordset
- **The default cursor returns the data in a tuple of tuples**
  - This means you could use one of the cursor object's fetch methods to return records, or you could treat the cursor as an iterator object
  - Cursors may also be used as dictionary objects
    - This allows the use of field names as keys to access values
- **To create a cursor, use the connection object's `.cursor()` method**

Syntax

```
conn.cursor([cursorClass])
```

  - Creates a cursor object from the connection object
  - May accept an optional custom cursor class that extends the `sqlite3.Cursor` class

Cursor Attributes

- **Cursors support a number of attributes that may expose metadata about the recordset or control the behavior of the recordset**

- **`cur.lastrowid`**
  - This is a read-only attribute that provides the `rowid` of the last modified row
    - It is only set if you issue an `INSERT` statement using an `execute()` statement
    - `lastrowid` defaults to `None` for operations other than `INSERT` of when an `executemany()` is used

- **`cur.arraysize`**
  - This read/write attribute specifies how many rows to fetch at a time when using the `.fetchmany()` method
  - Defaults to 1

- **`cur.rowcount`**
  - While the sqlite3 module's Cursor class implements this attribute, support from the underlying SQLite database engine is quirky
    - When using an `executemany()` statement, the number of modifications are summed up into the `rowcount` attribute
    - Per the DB-API specification, the `rowcount` attribute is set to -1 if "no `executexx()` has been performed on the cursor or the `rowcount` of the last operation is not determinable by the interface"

Cursor Methods to Execute SQL Statements

- **Like the connection object, cursor objects support a number of methods to execute `SQL` statements**
  - The connection object methods discussed earlier are shortcuts to these methods

- **`cur.execute(sql[, parameters])`**
  - This method executes a `SQL` statement
  - `SQL` statements may be parameterized
  - Raises a warning if more than one statement is executed

- **`cur.executemany(sql, sequence_of_parameters)`**
  - Executes a `SQL` statement against all parameter sequences or mappings found in the sequence

- **`cur.executescript(sql_script)`**
  - This is a nonstandard method for executing multiple `SQL` statements at once
  - Issues a `COMMIT` statement first, then executes the `SQL` script supplied as a parameter

Using Parameterized Statements with Cursors

- **Occasionally SQL Operations will need to use values from Python variables**
  - Assembling SQL statements using Python's string operations is looked at as an insecure practice and may leave your Database open to SQL injection attacks
- **The DB-API supports the use of parameter substitution in SQL Statements**
  - The default parameter style is to supply a `?` as a placeholder (parameter marker) wherever you need to use a value
  - Provide a tuple of values as the second argument of the `.execute()` method to perform ordinal parameter substitution

Syntax

```
db.execute('insert into table (f1, i2) values (? , ?)' ,
('v1',2))
```

- **The following table lists the different types of parameter markers supported by the DB-API**
  - The sqlite3 module supports the Question mark and Named styles

Parameter Marker Style	Meaning
`qmark`	Question mark style, `field = ?`
`numeric`	Numeric positional style , `field = :1`
`named`	Named Style, `field = :name`
`format`	ANSI C printf format code, `field = %s`
`pyformat`	Python extended format code , `field = %(name)s`

Example: Inserting Records with Parameterized queries

Example

In this example we are going to use the question mark style of parameter markers and the named style of parameter markers. Notice that with the question mark style, we supply the parameters as a tuple, and with the name style we supply the parameters as a dictionary

```
import sqlite3 as s

c = s.connect('student3.db')
cur = c.cursor()
cur.executescript('''
        DROP TABLE IF EXISTS STUDENT;
        CREATE TABLE STUDENT
       (ID INT PRIMARY KEY     NOT NULL,
        NAME TEXT NOT NULL);
        ''')
c.commit()
print(c.total_changes)

cur.execute('INSERT INTO STUDENT VALUES(?,?)', (1, 'Gary'))

c.commit()
print(c.total_changes)

cur.execute('INSERT INTO STUDENT VALUES(:id,:name)',
            {'id':2, 'name':'Roland'})
c.commit()
print(c.total_changes)
c.close()
```

Output

```
0
1
2
```

Example: Using `executemany()` to Insert Multiple Records

Example

In this example we are going to drop and create a table using the `executescript()` method. We are then going to use the `executemany()` method to execute a parameterized insert statement against a multi-dimensional tuple of records. Boom.

```
import sqlite3 as s
students = (
    (1, 'Gary'),
    (2, 'Roland'),
    (3, 'Dana'),
    (4, 'Chris'),
    (5, 'James'),
    (6, 'Khan'),
    (7, 'Spock')
)

c = s.connect('student4.db')
cur = c.cursor()
cur.executescript('''
        DROP TABLE IF EXISTS STUDENT;
        CREATE TABLE STUDENT
       (ID INT PRIMARY KEY     NOT NULL,
        NAME TEXT NOT NULL);
        ''')

cur.executemany('INSERT INTO STUDENT VALUES(?,?)',
students)

c.commit()
print(c.total_changes)
c.close()
```

Output

```
7
```

Example: Updating Records

Example

In this example we are going to use the `execute()` method to update a record. We will use a parameterized update statement.

```
import sqlite3 as s
students = (
    (1, 'James'),
    (2, 'Khan'),
    (3, 'Spock')
)
c = s.connect('student5.db')
cur = c.cursor()
cur.executescript('''
        DROP TABLE IF EXISTS STUDENT;
        CREATE TABLE STUDENT
       (ID INT PRIMARY KEY     NOT NULL,
        NAME TEXT NOT NULL);
        ''')

cur.executemany('INSERT INTO STUDENT VALUES(?,?)',
students)
c.commit()
cur.execute('SELECT * FROM STUDENT WHERE ID = 2')
row = cur.fetchone()
print(row[0],row[1])

cur.execute('UPDATE STUDENT SET NAME = :name where ID = 2',
{'name':'KHAAANNNN!'})
cur.execute('SELECT * FROM STUDENT WHERE ID = 2')
row = cur.fetchone()
print(row[0],row[1])
c.commit()
c.close()
```

Output

```
2 Khan
2 KHAAANNNN!
```

Example: Deleting Records

Example

In this example we will use the cursor object's `execute()` method to delete a record.

```
import sqlite3 as s
away = (
    (1, 'James'),
    (2, 'Bones'),
    (3, 'Spock'),
    (4, 'Uhura'),
    (5, 'Red Shirt')
)
c = s.connect('awayteam.db')
cur = c.cursor()
cur.executescript('''
        DROP TABLE IF EXISTS AWAYTEAM;
        CREATE TABLE AWAYTEAM
       (ID INT PRIMARY KEY     NOT NULL,
        NAME TEXT NOT NULL);
        ''')

cur.executemany('INSERT INTO AWAYTEAM VALUES(?,?)', away)
c.commit()

print('Oh, No ! Trouble Captain! ' )
cur.execute('DELETE FROM AWAYTEAM WHERE ID = 5')

cur.execute('SELECT * FROM AWAYTEAM')
for member in cur:
    for fld in member:
        print(fld, end = ' ')
    print()
```

Fetching Records From The Database

- **Cursor objects support a number of methods to fetch rows from a recordset**
  - `fetchone()`
  - `fetchmany()`
  - `fetchall()`

- **`cur.fetchone()`**
  - Fetches the next row of a query result set and returns a single sequence
  - Returns `None` when no data is available

- **`cur.fetchmany(size = cursor.arraysize)`**
  - Fetches the next set of rows from the query result as a list
  - Returns an empty list when no rows are available
  - The number of rows to retrieve per call is specified by the `size` parameter
    - To avoid performance issues, set the number of records to return by setting the `cursor.arraysize` attribute directly

- **`cur.fetchall()`**
  - Fetches all rows of a query result and returns a list
  - Returns an empty list when no records are available

Example: Using `fetchone()` to Fetch a Single Record

Example

In this example we are going to use the `fetchone()` method to fetch a single record

```
import sqlite3 as s

c = s.connect('awayteam.db')
cur = c.cursor()
cur.execute('SELECT * FROM AWAYTEAM')
row = cur.fetchone()

print(row)
```

Output

```
(1, 'James')
```

Example: Using `fetchmany()` to Fetch a Specified Number of Records

Example

In this example we will set the cursor.arraysize attribute to specify how many records to retrieve at a time using the `fetchmany()` method

```
import sqlite3 as s

c = s.connect('awayteam.db')
cur = c.cursor()
cur.arraysize = 3
cur.execute('SELECT * FROM AWAYTEAM')
rows = cur.fetchmany()

for row in rows:
    for fld in row:
        print(fld, end = ' ')
    print()
```

Output

```
1 James
2 Bones
3 Spock
```

Example: Using `fetchall()` to Fetch All Records

Example

In this example we will use `fetchall()` to return a tuple of all the records in the recordset. We will then iterate through the tuples using multiple for loops

```
import sqlite3 as s

c = s.connect('awayteam.db')
cur = c.cursor()
cur.execute('SELECT * FROM AWAYTEAM')
rows = cur.fetchall()


for row in rows:
    for fld in row:
        print(fld, end = ' ')
    print()

for row in rows:
    print(row)
```

Output

```
1 James
2 Bones
3 Spock
4 Uhura
5 Red Shirt

(1, 'James')
(2, 'Bones')
(3, 'Spock')
(4, 'Uhura')
(5, 'Red Shirt')
```

`Row` Objects

- **Cursor objects may return `Row` objects from record sets**
  - Row objects serve as an optimized `row_factory` for Connection objects
    - The connection object `row_factory` property allows developers to implement advanced ways to return results, such as returning an object that allows access by column names
    - It is suggested to set the `conn.row_factory` attribute to the `sqlite3.Row` type
- **Row objects support the following features**
  - Try to mimic tuples
  - Access through index and column name mappings
  - Iteration
  - Length testing
  - Equality testing
  - A `keys()` method which returns a tuple of column names
- **`Row` objects are created by assigning the connection object's `row_factory` attribute to the `sqlite3.Row` type**

Syntax

```
conn.row_factory = sqlite3.Row
```

Example: Working with a Row Object

Example

In this example we will set the connection's row_factory attribute to use the sqlite3.Row type. We will then use the `fetchone()` method to retrieve a single row object. We will then work with that row object.

```
import sqlite3 as s

c = s.connect('awayteam.db')
c.row_factory = s.Row
cur = c.cursor()
cur.execute('SELECT * FROM AWAYTEAM')
row = cur.fetchone()
print(type(row))
print(tuple(row))
print(len(row))
print(row[0],row[1])
print(row.keys())
print(row['id'], row['name'])

for fld in row:
    print(fld , end = ' ' )
```

Output

```
<class 'sqlite3.Row'>
(1, 'James')
2
1 James
['ID', 'NAME']
1 James
1 James
```

Module Summary

- **Database APIs written to the DB-API 2.0 specification provide consistent interfaces for database access**
- **The sqlite3 module is an API to access SQLite databases and is built based on the DB-API 2.0 specification**
- **Common database API objects include:**
  - Connection Objects
  - Cursor Objects
- **Connection Objects represent the database connection**
- **Cursor Objects are used to perform operations against the database**
- **The Structured Query Language (SQL) is used to interact with a Database**
- **Common SQL Statements include SELECT, INSERT, UPDATE, and DELETE**

Module 15
(Optional)

Graphical User Interface Programming with `tkinter`

Module Goals and Objectives

Major Goals

Understand how to use tkinter to create Graphic User Interfaces

Understand how to Handle Events

Specific Objectives

On completion of this module, students will be able to:

Create Graphic User Interfaces

Bind controls to commands

Handle events

Use geometry managers to layout controls

Capture input from users

Use input captured from users

Use common tkinter Widgets

Section 15–1

GUI Programming in Python

GUI Programming Overview

- **While shell-based applications may be fine for System Administrators and Developers to use, most users are accustomed to using some form of a Graphical Interface to interact with applications**

- **Because GUIs give users event driven access to programs, GUIs are a critical part of modern systems**

- **Python has proven to be an effective tool in the GUI programming domain**
  - Developing GUI applications takes much less time in Python in comparison to other programming languages
  - Because of Python's rapid development time, refactoring existing GUIs or prototyping GUIs also takes much less time in comparison to other programming languages

- **While the core Python programming language does not support GUI programming, it does ship with the `tkinter` module, which is an interface to `TCL/TK`, as part of its standard library**
  - There are also many Python ports of cross-platform and platform-specific GUI programming Libraries

Goals of GUI Programming

- **In GUI Programming there is a standard process for building a User Interface**
  - Specify how the GUI is going to look
    - What controls will the user see
    - How will the controls be laid out
    - What thematic elements will be used
  - Specify what tasks will be accomplished by the GUI
  - Bind any user input controls to any functions or methods that have been created to perform the tasks
  - Create event handling code that will wait for or sense any End User stimulus to execute the tasks

Introducing `tkinter`

- **`tkinter` is an open source GUI library**
  - Is the continuing de facto standard for portable GUI development in Python
- **`tkinter` is a thin object-oriented layer that sits on top of Tcl/Tk**
- **Python GUI Programs that are developed using `tkinter` run portably on Windows, X Windows (UNIX and Linux), and Macintosh OS X**
  - `Tkinter` GUIs display a native look-and-feel on each of these platforms
- **`tkinter` makes it easy to build simple and portable GUIs quickly**
  - It can be easily augmented with Python code, as well as with larger extension packages
- **`tkinter` is mature, robust, widely used, and well documented**
  - `tkinter` includes roughly 25 basic widget types, plus various dialogs and other tools

`tkinter` and `TK`

- **The underlying `Tk` library used by `tkinter` is a standard in the open source world**
  - Tk is also used by the Perl, Ruby, PHP, Common Lisp, and Tcl scripting languages
- **The Python binding to `Tk` is enhanced by Python's simple object model**
  - In Python, `Tk` widgets become customizable and embeddable objects
- **`tkinter` takes the form of a module package in Python 3.X**
  - The `tkinter` package ships with nested modules that group some of its tools by functionality
- **In Python 2.X the `tkinter` module is known as `Tkinter` and has a different organizational structure**

`tkinter` and the Python Standard Library

- **`tkinter` is a relatively lightweight toolkit, and works well with Python**

- **Python's `tkinter` module ships with Python as a standard library module**
  - `tkinter` is the only GUI toolkit that is part of Python
  - All other GUI libraries used by Python are third-party extensions

- **`tkinter` is the basis of Python's standard `IDLE` integrated development environment GUI**

- **Python 2.7 and Python 3.1 incorporate the "themed Tk" ("`ttk`") functionality of `Tk 8.5`**
  - This allows `Tk` widgets to be easily themed to look like the native desktop environment in which the application is running
  –

Other GUI Libraries: wxPython and PyQT

- **wxPython**
  - A Python interface for the open source wxWidgets (formerly known as wxWindows) library
  - It is a portable GUI class framework originally written to be used from the C++ programming language
  - The wxPython system is an extension module that wraps wxWidgets classes
  - The wxPython library builds sophisticated interfaces
  - GUIs coded in Python with wxPython are portable to Windows, UNIX-like platforms, and Mac OS X
  - Considered the second most popular Python module behind tkinter

- **PyQt**
  - A Python interface to the Qt toolkit
  - PyQt is a full featured GUI library and runs portably today on Windows, Mac OS X, and UNIX and Linux
  - Qt is generally more complex, yet more feature rich, than tkinter; it contains hundreds of classes and thousands of functions and methods.
  - The PyQt and PyKDE extension packages provide access to KDE development libraries
  - The third most widely used GUI toolkit for Python today

Other GUI Libraries: PyGTK and Dabo

- **PyGTK**
  - A Python interface to GTK, a portable GUI library originally used as the core of the Gnome window system on Linux
  - The gnome-python and PyGTK extension packages export Gnome and GTK toolkit calls
  - PyGTK runs portably on Windows and POSIX systems such as Linux and Mac OS X
- **Dabo**
  - Dabo is a 3-tier, cross-platform application development framework, written in Python atop the wxPython GUI toolkit
  - The Dabo framework enables you to easily create powerful desktop applications
  - The 3-tier pattern separates code into 3 logical layers: UI/presentation, business logic, and data access
  - Dabo has native support for many Databases including MS SQL Server, MySQL, SQLite and PostgresSQL
  - Its open design is intended to eventually support a variety of databases and multiple user interfaces (wxPython, tkinter, and even HTML over HTTP)

The Case for tkinter

- **`tkinter`'s accessibility, portability, availability, documentation, and extensions have made it the most widely used Python GUI solution for many years running**
- **tkinter is generally regarded as a lightweight toolkit**
- **It is also one of the simplest GUI solutions for Python available**
  - Simple `tkinter` GUIs can be created in a few lines of code
    - Features may be gradually and easily added to create more robust GUIs
- **A Python GUI built with tkinter will run without source code changes on all major windowing platforms today with a native look-and-feel on each of these platforms the GUI runs on**
- **`tkinter` is a standard module in the Python library, shipped with the interpreter**
  - Most Python installation packages come with `tkinter` support bundled into the package
- **`tkinter` is also generally better supported than alternative GUI packages because the underlying Tk library is also used by Tcl and other programming languages such as Perl and Ruby**

`tkinter` Extensions: PMW and Tix

- **Python developers also have access to Python extensions designed to work with or augment tkinter**

- **PMW**
  - Pmw (Python Mega Widgets) is a toolkit for building high-level compound widgets in Python using the tkinter module
  - It consists of a set of base classes and a library of flexible and extensible megawidgets built on this foundation
    * These megawidgets include notebooks, combo boxes, selection widgets, paned widgets, scrolled widgets and dialog windows

- **Tix**
  - The Tix (Tk Interface Extension) module provides an additional rich set of widgets to tkinter
  - Tix is a collection of more than 40 advanced widgets, originally written for Tcl/Tk
  - Tix is available for use in Python/tkinter programs as a Python standard library module, called `tkinter.tix`

`tkinter` Extensions: TTK and PIL

- **ttk**
  - Tk themed widgets, ttk, is a relatively new widget set which attempts to separate the code implementing a widget's behavior from that implementing its appearance
  - The ttk module provides access to the Tk themed widget set, introduced in Tk 8.5
  - Widget classes handle state and callback invocation, whereas widget appearance is managed separately by themes
  - ttk comes with advanced widget types, some of which are not present in standard tkinter itself including Combobox, Notebook, Progressbar, Separator, Sizegrip and Treeview
  - The best use for the ttk module is to import it after tkinter in order to use its replacement widgets and configure style objects possibly shared by multiple widgets, instead of configuring widgets themselves
  - ttk was incorporated into Python's standard library in Python3.1, as module `tkinter.ttk`

- **PIL**
  - The Python Imaging Library (PIL) is an open source extension package that adds image processing tools to Python
  - It provides tools for image thumbnails, transforms, and conversions, and it extends the basic tkinter image object to add support for displaying many image file types
    * JPEG, TIFF, PNG, etc.

Verifying `tkinter` Is Installed: the `_tkinter` Extension

- **If you're using IDLE, you already have Tkinter installed**
  - On Windows and Mac OS X, Tkinter comes as part of the Python distribution
- **From an interactive shell , make sure you have access to the extension module (written in C) that tkinter uses**

Syntax

```
import _tkinter
```

  - If you get no errors then move on to the next step
  - If you do get errors, check the documentation for installing tkinter on your platform
  - You may have to install `Tcl` and `Tk` (when using RPM, install the `-devel` RPMs as well) and/or edit the `setup.py` script to point to the right locations where `Tcl/Tk` is installed
    * If you install `Tcl/Tk` in the default locations, simply rerunning "make" should build the `_tkinter` extension

Verifying `tkinter` Is Installed: the `tkinter` Module

- **Next, import the `tkinter` module**

Syntax

```
import Tkinter #Python 2.x

import tkinter #Python 3.x
```

- **If this check fails with "No module named Tkinter", your Python configuration needs to be changed to include the directory that contains `Tkinter.py` in its default module search path**
  - You may have forgotten to define `TKPATH` in the Modules/Setup file

- **A temporary workaround would be to find the directory that contains `Tkinter.py` and add it to your `PYTHONPATH` environment variable**

Verifying `tkinter` Is Installed: Does `tkinter` work?

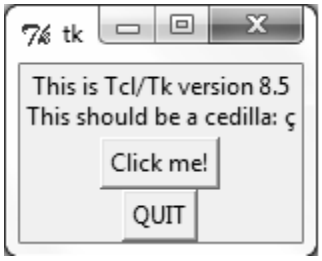
- **If you have access to the `_tkinter` extension and the `tkinter` module is available, the last step is to check that `tkinter` is working properly by using the `_test()` method**

- **The `_test()` method opens a pop-up window with a message and a quit button**
  - If the `_test()` method works then you are all set

Syntax

```
Tkinter._test() #Python 2.x
tkinter._test() #Python 3.x
```

–



- **Optionally loading the tkinter module from the shell should execute the `_test()` method**

Syntax

```
>python -m tkinter
```

Section 15–2

`tkinter` Concepts

Importing `tkinter`

- **To import tkinter to explicitly reference all symbols from the package, use the import module form on the import statement**

Syntax

```
import tkinter #Python 3.x
import Tkinter #Python 2.x
```

- **To import all symbols from tkinter into the calling scripts namespace use the `from module import *` form of import**

Syntax

```
from tkinter import * #Python 3.x
from Tkinter import * #Python 2.x
```

- **To import specific symbols from tkinter use the `from module import namex` form of import**

Syntax

```
from tkinter import sym1 , sym2, ... symn #Python 3.x
from Tkinter import sym1 , sym2, ... symn #Python 2.x
```

Example: Importing tkinter

Example

```
import tkinter
class MyGUI:
    def __init__(self):
        self.win = tkinter.Tk()
        self.label = tkinter.Label(self.win,
                                   text='Hello World!')
        self.label.grid()
        tkinter.mainloop()
my_gui = MyGUI()
```

```
from tkinter import *
class Gui:
    def __init__(self):
        self.win = Tk()
        self.label = Label(self.win,text='Hello World!')
        self.label.grid()
        mainloop()
my_gui = Gui()
```

```
from tkinter import Tk, Label, mainloop
class Gui:
    def __init__(self):
        self.win = Tk()
        self.label = Label(self.win,text='Hello World!')
        self.label.grid()
        mainloop()
my_gui = Gui()
```

Widgets

- **One of the fundamental concepts in tkinter is the widget**
  - Widget is a portmanteau of the words Window and Gadget
- **Widgets are data structures that have an visible onscreen representation**
- **Widgets are the building blocks of tkinter GUI applications**
- **tkinter is a collection of widget definitions and commands used to operate on the widgets**
- **In Python, each different type of widget is represented by a different Python Class**
  - Each widget is a subclass of the tkinter `Widget` abstract class
- **Widgets share a number of common attributes which makes configuring widgets consistent**
- **All widgets also have a shared "Universal" set of methods that are attached to them**
- **Widgets are created by instantiating an object with a Widget Class**

Core Widgets

- **The following is a table of the core widget classes in tkinter**

Widget Class	Description
`Label`	Simple message area
`Button`	Simple pushbutton widget
`Frame`	Container for attaching and arranging other widget objects
`Toplevel, Tk`	Top level windows managed by the window manager
`Message`	Multiline text display field
`Entry`	Simple single line text entry field
`Checkbutton`	Two-state button widget, used for multiple choice selections
`Radiobutton`	Two-state button widget, used for single choice selections
`Scale`	A slider widget with scalable positions
`PhotoImage`	Image object for placing full-color images on other widgets
`BitmapImage`	Image object for placing bitmap images on other widgets
`Menu`	Options associated with a Menubutton or top-level window
`Menubutton`	Button that opens a Menu of selectable options/submenus
`Scrollbar`	Bar for scrolling other widgets
`Listbox`	List of selection names
`Text`	Multiline text browse/edit widget, supports fonts
`Canvas`	Graphics drawing area
`OptioinMenu`	Composite: pull-down selection list
`PanedWindow`	A multipane window interface
`LabelFrame`	A labeled frame widget
`Spinbox`	A multiple selection widget

Configuring Widgets

- **Each widget has attributes attached to them that may be used to configure the behavior of each widget**

- **tkinter supports 3 different ways to set the options of a widget**

- **When a widget is created, it may be configured using keyword arguments**

Syntax

```
x = Widget(parent, key = 'value' [,...])
```

- **After a widget is created, it may be treated like a dictionary object, or it may be changed by using the `config()` method**

Syntax

```
x['Key'] = 'value
```

Syntax

```
x.config(key = 'value')
```

Application Windows

- **In tkinter there are a number of Widgets that are used as container objects such as Windows and Frames**
- **Root Window / Top-Level Windows**
  - These are windows that exist independently on your screen
  - They are decorated with the standard frame and controls for your system's desktop manager
- **Root windows are created when a call is made to the `Tk()` constructor**

Syntax

```
import tkinter
class Gui:
    def __init__(self):
        self.win = tkinter.Tk()
```

- **Top level windows are made with a call to the `Toplevel()` widget**

Syntax

```
import tkinter
class Gui:
    def __init__(self):
        self.win = tkinter.Tk()
        self.win.title('Root')

        self.top = tkinter.Toplevel()
        self.top.title('Top')
```

Example: Application Windows

Example

```
import tkinter
class Gui:
    def __init__(self):
        self.win = tkinter.Tk()
        self.win.title('Root')

        self.top = tkinter.Toplevel()
        self.top.title('Top')

        self.labelwin = tkinter.Label(self.win,text='Root')
        self.labeltop = tkinter.Label(self.top,text='Top')
        self.labelwin.grid()
        self.labeltop.grid()

        tkinter.mainloop()
my_gui = Gui()
```

Geometry Managers

- **Geometry management in tkinter is the process of specifying widget placement and size inside the GUI**
- **Geometry managers specify how widgets will be arranged inside of the GUI**
- **tkinter has 3 different geometry managers**
  - place
  - pack
  - grid
  - The text and canvas widgets do have a limited capability to manage the geometry of child widgets
- **Geometry management in tkinter is based off of a parent/child relationship between widgets**
  - Parent widgets (typically a root window, Toplevel window or a Frame) contain a series of child widgets
  - Geometry managers specify how the child widgets will be displayed and laid out within the scope of the patent
    - I.E. unless otherwise specified, placement of widgets is relative

Geometry Managers: `pack()`

- **The packer geometry manager is the simplest of the geometry managers to use**
- **The packer allows for widgets to be placed on the screen relative to their parent widget**
- **Each child widget is positioned within the parent widget from the edges of the parent to the center**
- **Packer is one of the most commonly used geometry managers in tkinter**
- **The packer allows for the rapid design of GUI layouts**
- **The packer excels when widgets are going to be laid out relatively side by side or on top of each other**

Syntax

```
widget.pack(**kwargs)
```

Geometry Mangers: `pack()` attributes

- **The packer has a number of attributes that may be specified as arguments of the pack method or as arguments of one of the packer `configure()` methods**

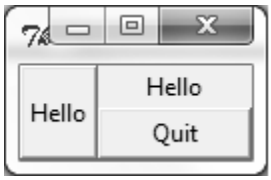
Attribute	Values	Explanation
`expand`	`YES , NO , 1 , 0`	Specifies whether the widget should expand to fill the available space (at packing time and on window resize)
`fill`	`NONE, X, Y, BOTH`	Specifies how the child widget should be resized to fill the available space (if the child widget is smaller than the available space)
`side`	`TOP (default), BOTTOM, RIGHT, LEFT`	Specifies which side of the parent should be used to position the child
`padx, pady`	`Integer`	Specifies the padding space between two adjacent widgets
`ipadx, ipady`	`Integer`	Specifies any internal padding in a widget
`anchor`	`N, NW, W, SW, S, SE, E , SW, NS, EW CENTER (default)`	Instructs the packer to position the widget at the desired position within its allocated space, if the space allocated for the widget is larger than the space needed to display the widget

Example: Geometry Manager `pack()`

Example

In this simple example we demonstrate how to use some common attributes of the packer

```
from tkinter import *


win = Tk()
Button(win, text='Hello').pack(side=LEFT, fill=Y)
Label(win, text='Hello').pack(side=TOP)
Button(win, text='Quit', command=win.quit).pack(side=RIGHT,
expand=YES,fill=X)

mainloop()
```



Geometry Managers: `grid()`

- **The Grid uses familiar tabular layout principles to position widgets in the UI**
- **The Grid geometry manager places the widgets in a 2-dimensional table, which consists of a number of rows and columns**
- **The position of a widget is defined by a row and a column number**
  - Widgets with the same column number and different row numbers will be above or below each other
  - Widgets with the same row number but different column numbers will be next to each other on the same row
- **Widgets are placed in the grid by specifying the row and the column number**
- **The size of the grid doesn't have to be defined, because the manager automatically determines the best dimensions for the widgets used**
- **The Grid in many cases is the best choice for general GUI development**

Syntax

```
widget.grid(**kwargs)
```

Geometry Managers: `grid()` attributes

- **The grid has a number of attributes that may be specified as arguments of the grid method or as arguments of one of the grid `configure()` methods**

Attribute	Values	Explanation
`row`	`Integer`	Integer value specifying which row to use (Starts with 0)
`column`	`Integer`	Integer value specifying which column to use (Starts with 0)
`rowspan`	`Integer`	Integer value specifying how many rows a cell will span
`columnspan`	`Integer`	Integer value specifying how many columns a cell will span
`padx, pady`	`Integer`	Specifies the padding space between two adjacent widgets
`ipadx, ipady`	`Integer`	Specifies any internal padding in a widget
`sticky`	`N, NW, W, SW, S, SE, E , SW, NS, EW, NSEW`	Instructs the grid to position the widget at the desired position within its allocated space, if the space allocated for the widget is larger than the space needed to display the widget

Example: Geometry Manager `grid()`

Example

In this simple example we demonstrate how to use some common attributes of the grid. (Insert Tron joke here)

```
from tkinter import *

mprd = {'Something Completely Different' : 1971,
        'Holy Grail': 1975,
        'Life of Brian': 1979,
        'Live at the Hollywood Bowl': 1982,
        'The Meaning of Life': 1983
        }

r = 0
for m in mprd:
    Label(text=mprd[m], relief=RIDGE, width =
              5).grid(row=r,column=0)
    Label(text = m, relief=SUNKEN, width =
              30).grid(row=r,column=1)
    r = r + 1

mainloop()
```

Geometry Managers: `place()`

- **The place geometry manager is the most flexible of all the geometry managers**

- **The placer allows you to explicitly set the position of child widgets within the scope of the parent**
  - Positioning may be performed in either relative terms or in absolute terms by specifying x and y coordinates

Syntax

```
widget.place(**kwargs)
```

Geometry Managers: `place()` attributes

- **The placer has a number of attributes that may be specified as arguments of the place method or as arguments of one of the place `configure()` methods**

Attribute	Values	Explanation
`bordermode`	`INSIDE, OUTSIDE`	Specifies if the outside border should be taken into consideration when placing the widget
`x`	`Int`	X coordinate relative to parent widget to position the `place` widget. Default is 0.
`y`	`Int`	y coordinate relative to parent widget to position the `place` widget. Default is 0.
`rely`	`Float (between 0.0 and 1.0)`	locate anchor of this widget between 0.0 and 1.0 relative to height of parent (1.0 is bottom edge)
`relx`	`Float (between 0.0 and 1.0)`	locate anchor of this widget between 0.0 and 1.0 relative to height of parent (1.0 is right edge)
`width`	`Integer`	Width of widget in pixels
`height`	`Integer`	Height of widget in pixels
`anchor`	`N, NW, W, SW, S, SE, E , SW, NS, EW, NSEW, CENTER`	Instructs the grid to position the widget at the desired position within its allocated space, if the space allocated for the widget is larger than the space needed to display the widget

Example: Geometry Managers `place()`

Example

In this simple example we demonstrate how to use some common attributes of the `place()` geometry manager

```
from tkinter import *
from tkinter.messagebox import *

def helloCallBack():
    showinfo( "Hello Python", "Hello World")

top = Tk()

b = Button(top, width=12, height=12,
               text ="Hello", command = helloCallBack)
b.place(bordermode = OUTSIDE, relx=1, x=-2, y=2, anchor=NE)

top.mainloop()
```

Section 15–3

Event Handling in tkinter

Event Handling in tkinter

- **In Tk, there is an event loop which receives events from the operating system**

- **tkinter applications spend most of the time running in an event loop that is generated by the `mainloop()` method**

Syntax

```
mainloop()
w.mainloop()
```

  - This method must be called—generally after all the static widgets are created—to start processing events
  - To leave the event loop, use the `quit()` method
  - You can also call this method inside an event handler to resume the main loop

Syntax

```
quit()
w.quit()
```

  - This method exits the main loop

- **Typically, tkinter will handle management of the event loop on your behalf**
  - tkinter will monitor widgets and dispatch any built-in events accordingly

Connect Application Logic with Widgets

- **Events occur when users interact with widgets**
- **tkinter makes it very easy to bind application logic to the events that are attached to widgets**
- **There are 2 methods that can be used to bind events to application logic**
  - Command Callbacks
  - Event Binding
- **Command Callbacks may be used with widgets that have a `command` attribute**
- **Event Binding may be used to bind events to widgets that do not have a `command` attribute, or to bind application logic to specific events**

Command Callbacks

- **Often you will want your application to handle particular events**
  - Such as a user pushing a button, or selecting an item from a menu
- **Most widgets provide the ability to set a option to bind a callback to a widget**
  - This option is typically the "`command`" attribute of a widget, but more complex widgets may provide more specialized forms of the `command` attribute
- **Callbacks in tkinter tend to be simpler than other languages, as callbacks are just normal code that gets evaluated**
  - May be function or method calls
  - May be a lambda expression

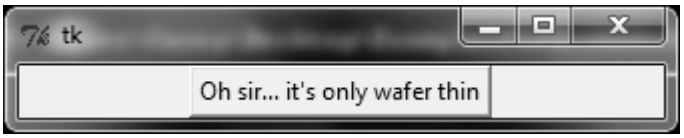
Syntax

```
widget(command  =  code)
widget.config(command = code)
```

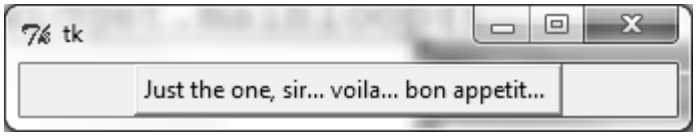
Example: Command Callbacks

Example

```
import sys
from tkinter import *
widget = Button(None, text='But it\'s wafer thin!!',
command=sys.exit)
widget.pack()
widget.mainloop()
```




```
import sys
from tkinter import *
def quit(): # a custom callback handler
    print(' Thank you, sir, and now the check. ')
widget = Button(None, text='Just the one, sir... voila...
bon appetit... ', command=quit)
widget.pack()
widget.mainloop()
```




```
C:\HOTTPython\GUI>python examplecustomcommand.py
 Thank you, sir, and now the check.
```

Binding Events with `bind()`

- **For widgets or events that do not have a `command` callback associated with them, specific methods may be used to capture any event and then associate that event to any callback**

- **The `bind()` method is a more general tkinter event binding mechanism that can be used to register callback handlers for lower-level interface events**

- **Unlike `command` callbacks, `bind()` callbacks receive an event object argument (an instance of the tkinter Event class) that gives context about the event**

Syntax

```
w.bind(eventsequence=None, func=None, add=None)
```

  - This method is used to attach an event binding to a widget
  - The `sequence` argument describes the event
  - The `func` argument is the function callback that is to be bound
  - Typically, old callback bindings are replaced by new callback bindings, however old bindings may be preserved with new bindings by passing `add='+'`

Example: Binding Events with `bind()`

Example

This is not The Greatest Example of Binding in the World, no. This is just a tribute.

```
import sys
from tkinter import *

def msg(event):
    print('Press twice to exit')

def bail(event): # on double-left click
    print('System powering down in 3...2...1...')
    sys.exit()

w = Button(None, text='Big Damn Heroes')
w.pack()
w.bind('<Button-1>', msg) # bind left mouse clicks
w.bind('<Double-1>', bail) # bind 2X-left clicks
w.mainloop()
```

Binding Levels

- **Event handlers may be bound at different levels in your application**
- **To bind event handlers to an instance use the `bind()` method as discussed**
- **To bind events to the top level objects, bind events using `bind()` to the `Toplevel` object or to the `root` object**
- **To bind events to an entire widget class, use the `bind_class()` method**

Syntax

```
w.bind_class(className, eventsequence=None, func=None,
add=None)
self.myCanvas.bind('<Button-2>', self.drawLine)
```

- **To bind events to an entire application use the `bind_all()` method**

Syntax

```
w.bind_all(eventsequence=None, func=None, add=None)
self.bind_all("<F1>", help)
```

Event Sequences

- **Python uses event sequences to allow the user to define specific and general events that they want to bind to event handlers**
- **Event sequences are the first argument of the bind methods**
- **Event sequences are given as strings that have the following syntax**

Syntax

```
<[modifier-]...type[-detail]>
```

  - The entire sequence is contained within < >
  - The `type` denotes the type of event
  - `modifier` is optional and modifies the type argument to specify combinations of type
  - `detail` items are describe what keys are being monitored or which mouse buttons are being interacted with

Event Types

- **The following is a list of commonly used event types**

Name	Description
`Activate`	Occurs when there are changes in the state option of a widget such as being changed from inactive to active
`Button`	The user pressed a mouse button. The `detail` component specifies which mouse button
`Configure`	The user changed the size of a widget. (e.g. Dragged the corner of a window)
`Destroy`	A widget is being destroyed
`Enter`	A user has moved a mouse pointer into a visible portion of a widget
`FocusIn`	A widget has received input focus
`FocusOut`	The input focus has moved out of a widget
`KeyPress, Key`	The user pressed a key on the keyboard. The `detail` component specifies which key was pressed. Shorthand for `KeyPress` is `Key`
`KeyRelease`	User released a key
`Motion`	The user moved the mouse pointer in the widget
`MouseWheel`	User moved a mouse wheel up or down. May not work on Linux. `Button-4` may be used for wheel up on Linux, `Button-5` may be used for wheel down on Linux

Event Modifiers and Details

- **The following is a list of commonly used event modifiers**

Name	Description
Alt	True when the user is holding the alt key
Any	Generalizes an event type. (e.g. <Any-KeyPress>)
Control	True when the user is holding the control key down
Double	Specifies 2 events happening rapidly. (e.g. <Double-Button-1> describes 2 left clicks in rapid succession)
Shift	True when the user is holding the shift key

- **There are also shorthand forms of events**
  - `<1>` may be used for `<Button-1>`
  - `x` may be used for `<KeyPress-x>`
    - <> operators may be left off for most single characters

- **The following is a list of common event details**

Name	Description
1	Left mouse key
2	Right mouse key
3	Middle mouse key
Down	User pressed the down arrow
Up	User pressed the up arrow

Common Event Combinations

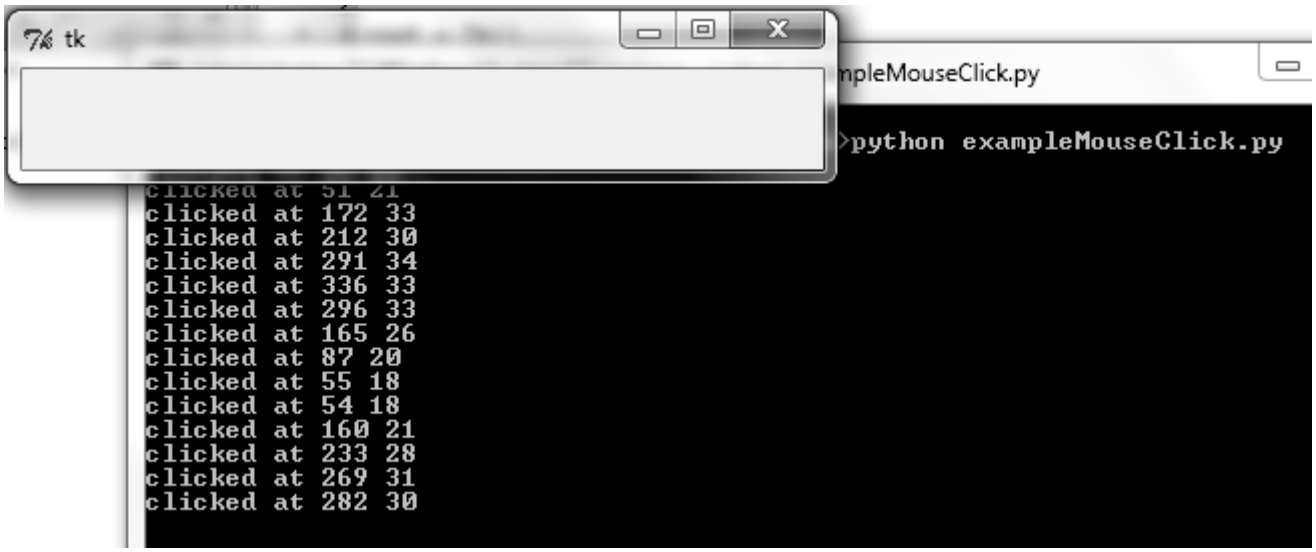
Name	Description
<Button-1>	Left mouse button pushed over widget. Current position of pointer relative to widget plus x and y coordinates are passed to callback object's event object
<Button-2>	Middle mouse button pushed over widget. Current position of pointer relative to widget plus x and y coordinates are passed to callback object's event object
<Button-3>	Right mouse button pushed over widget. Current position of pointer relative to widget plus x and y coordinates are passed to callback object's event object
<B1-Motion>	Mouse is being moved with Left button being held. B2 for middle button and B3 for right button
<ButtonRelease-1>	Left Button was released
<Double-Button-1>	Left Button was double clicked
<Enter>	Mouse pointer entered a widget
<Return>	Enter key was pressed on the keyboard
<Key>	User pressed any key
q	User pressed q
Alt-f	User pressed the f key while holding alt

Example: Capturing Mouse Events

Example

In this example we are going to bind an event handler to left mouse clicks. The x and the y coordinates of the clicks will be reported back to the console.

```
from tkinter import *

root = Tk()

def checkmouse(event):
    print("clicked at", event.x, event.y)

frame = Frame(root, width=400, height=50)
frame.bind("<Button-1>", checkmouse)
frame.pack()

root.mainloop()
```



Example: OO Capture Mouse Events

Example

In this example, we are going to capture mouse events. This script will be written OOP style.

```
from tkinter import *
from tkinter.messagebox import *

class Gui():

    def __init__(self, parent):

        self.frame = Frame(parent)
        self.frame.pack()

        self.myButton = Button(self.frame,
        text="Click for an Important Message from the
        Commander", command=self.msg)
        self.myButton.pack(side=LEFT)

        self.quitButton = Button(self.frame,
        text="Exit", command=self.frame.quit)
        self.quitButton.pack(side=LEFT)

    def msg(self):

        showinfo("Ovaltine!",
        "Be sure and drink your Ovaltine!")

parent = Tk()
app = Gui(parent)
parent.mainloop()
```

Section 15–4

Using Common tkinter Widgets

The Frame Widget

- **A frame is a container widget that displays just as a simple rectangle**

- **Frame widgets are a valuable tool in making your application modular**
  - Each frame has its own layout, so the layout of widgets within each frame works independently

- **Developers can group a set of related widgets into a compound widget by placing them into a frame**
  - Containers may be nested inside other containers to build hierarchical displays

Syntax

```
w = Frame(parent = None, **options)
```

- Creates a Frame widget
- `parent` is the parent widget
- `**options` are Frame configuration options

```
w.config(**options)
```

- Modifies one or more of the widget options
- If no options are given, returns a dictionary of all current widget options

Common Frame Options

- **This table enumerates some of the more common Frame widget options**

Option	Meaning
`background,bg`	The background color to use in this frame. This defaults to the application background color
`padding,pd`	Extra padding to be left on the inside of the frame; single argument is same padding all around, two arguments is different horizontal and vertical, and four is different for left, top, right and bottom
`borderwidth`	Width of a border around the widget, used in conjunction with relief
`relief`	Sets the display the frame's border: `flat, raised, sunken, solid, ridge, groove`
`width,height`	Specify an explicit size for the frame to request; if the frame contains other widgets, normally this is omitted, and the frame requests the size needed to accommodate the other widgets
`padx`	Adds horizontal padding to the inside of the Frame
`Pady`	Adds vertical padding to the inside of the Frame
`Takefocus`	By default, Frames do not participate in the focus traversal. Set this option to 1 if you want the Frame to participate in the focus traversal

Example

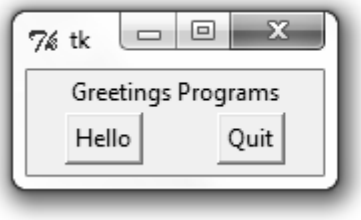
```
import tkinter

class Gui:
    def _ _init_ _(self):
    self.main = tkinter.Tk()
    self.top = tkinter.Frame(self.main)
    self.bottom = tkinter.Frame(self.main)
```

Example: Using Frame Widgets

Example

```
from tkinter import *
import sys
def chosen():
    print('The Master Control Program has chosen you to
serve your system on the Game Grid')

win = Frame()
win.pack()

Label(win, text='Greetings Programs').pack(side=TOP)
Button(win, text='Hello', command=chosen).pack(side=LEFT)
Button(win, text='Quit', command=sys.exit).pack(side=RIGHT)
win.mainloop()
```

```
The Master Control Program has chosen you to serve your system on the Game Grid
```



The Label Widget

- **A label is a widget that displays static text or static images**
- **Labels are used for such things as identifying controls or other parts of the user interface, providing textual feedback or results**

Syntax

```
w = Label(parent = None, **options)
```

- Creates a Label widget
- `parent` is the parent widget
- `**options` are Label configuration options

```
w.config(**options)
```

- Modifies one or more of the widget options
- If no options are given, returns a dictionary of all current widget options

Example

```
import tkinter
class Gui:
def __init__(self):
self.main = tkinter.Tk()
self.l1 = tkinter.Label(self.main, text='Hello World!')

from Tkinter import *
parent = Tk()
w = Label(parent, text="Hello, world!")
```

Common Label Options

- **This table enumerates some of the more common Label widget options**

Option	Meaning
`anchor`	This option controls where the text is positioned if the widget has more space than the text needs. Use one of N, NE, E, SE, S, SW, W, NW, or CENTER
`bg, background`	The background color of the label area
`bitmap`	Set this option equal to a bitmap or image object and the label will display that graphic
`font`	If you are displaying text in this label, specifies font for text rendering
`fg, foreground`	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap
`height`	Height of the label in lines. If this option is not set, the label will be sized to fit its contents.
`justify`	Defines how to align multiple lines of text. Use LEFT, RIGHT, or CENTER. Note that to position the text inside the widget, use the anchor option. Default is CENTER
`padx`	Extra space added to the left and right of the text within the widget. Default is 1
`pady`	Extra space added above and below the text within the widget. Default is 1.
`relief`	Specifies the appearance of a decorative border around the labelBorder decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE
`state`	Label state. This option controls how the label is rendered. The default is NORMAL. Other possible values are ACTIVE and DISABLED
`takefocus`	If true, the widget accepts input focus. The default is false
`text`	The text to display in the label. The text can contain newlines. If the bitmap or image options are used, this option is ignored. (text/Text)
`textvariable`	Associates a Tkinter variable (usually a StringVar) with the label. If the variable is changed, the label text is updated
`underline`	Used with the text option to indicate that a character should be underlined (e.g. for keyboard shortcuts). Default is -1 (no underline)
`width`	Width of the label in characters . If this option is not set, the label will be sized to fit its contents
`wraplength`	You can limit the number of characters in each line. The default value, 0, means that lines will be broken only at newlines

Example: OO Label Widget

Example

```
import tkinter

class Gui:
    def __init__(self):

        self.main = tkinter.Tk()

        self.l1 = tkinter.Label(self.main,
                  text='Hello World!')
        self.l2 = tkinter.Label(self.main,
                  text='This is a text label.')
        self.l2.config(padx = 10, pady = 10,
                  font = 'Helvetica')
        self.l1.pack()
        self.l2.pack()

        tkinter.mainloop()

my_gui = Gui()
```

The PhotoImage Widget

- **The PhotoImage class is used to display images (either grayscale or true color images) in labels, buttons, canvases, and text widgets.**
  - Used whenever an icon or an image needs to be displayed in a Tkinter application

- **The PhotoImage class can read GIF and PGM/PPM images from files:**

```
photo = PhotoImage(file="icon.gif")
photo = PhotoImage(file="icon.pgm")
```

  - The PhotoImage can also read base64-encoded GIF files from strings

Syntax

```
w = PhotoImage(parent = None, **options)
```

  - Creates a PhotoImage widget
  - `parent` is the parent widget
  - `**options` are PhotoImage configuration options

```
w.config(**options)
```

  - Modifies one or more of the widget options
  - If no options are given, returns a dictionary of all current widget options

Example: Using an Image as a Label

Example

```
from tkinter import *

parent = Tk()
photo = PhotoImage(file="icon.gif")
e = Label(parent, text = "Epic Smiley")
i = Label(parent, image=photo)
i.photo = photo
e.pack()
i.pack()

parent.mainloop()
```

The Button Widget

- **The Button widget is a standard Tkinter widget used to implement various kinds of buttons**
- **A button, unlike a frame or label, is very much designed for the user to interact with**
- **When the button is pressed, Tkinter automatically calls that function or method**
- **Buttons can contain text or images, and you can associate a Python function or method with each button**

Syntax

```
w = Button (parent = None, **options)
```

  - Creates a Button widget
  - `parent` is the parent widget
  - `**options` are Button configuration options

```
w.config(**options)
```

  - Modifies one or more of the widget options
  - If no options are given, returns a dictionary of all current widget options

Common Button Options

- **This table enumerates some of the more common Button widget options**

Option	Meaning
`anchor`	Where the text is positioned on the button. Use one of N, NE, E, SE, S, SW, W, NW, or CENTER
`bd, borderwidth`	Width of the border around the outside of the button. Default is two pixels
`bg, background`	Set background color
`bitmap`	Name of one of the standard bitmaps to display on the button
`command`	Function or method to be called when the button is clicked
`fg, foreground`	Foreground (text) color
`font`	Text font to be used for the button's label
`height`	Height of the button in text lines or pixels
`image`	Image to be displayed on the button (instead of text).
`justify`	Defines how to align multiple lines of text. Use LEFT, RIGHT, or CENTER. Note that to position the text inside the widget, use the anchor option. Default is CENTER
`padx`	Extra space added to the left and right of the text within the widget. Default is 1
`pady`	Extra space added above and below the text within the widget. Default is 1
`relief`	Specifies the appearance of a decorative border around the labelBorder decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE
`state`	Set this option to NORMAL, ACTIVE or DISABLED. Default is NORMAL
`takefocus`	Normally, keyboard focus does visit buttons
`text`	Text displayed on the button. Use internal newlines to display multiple text lines
`textvariable`	An instance of `StringVar()` that is associated with the text on this button. If the variable is changed, the new value will be displayed on the button
`underline`	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined
`width`	Width of the button in letters (if displaying text) or pixels (if displaying an image)

Example: OO Button Widget

Example

```
import tkinter
import tkinter.messagebox

class Gui:
    def __init__(self):

        self.main = tkinter.Tk()
        self.button = tkinter.Button(self.main,
         text='Click Me!',
        command=self.msg)

        self.button.pack()

        tkinter.mainloop()

    def msg(self):
        tkinter.messagebox.showinfo('Carolanne!!',
         'This button has been clicked')

my_gui = Gui()
```

Example 2: OO Button Widget

Example

This example extends the previous example which calls the `destroy()` method on the root object

```
import tkinter
import tkinter.messagebox

class Gui:
    def __init__(self):

        self.main = tkinter.Tk()
        self.button = tkinter.Button(self.main,
                      text='Click Me!',
                      command=self.msg)
        self.bquit = tkinter.Button(self.main,text='Quit',
                      command=self.main.destroy)

        self.button.pack()
        self.bquit.pack()

        tkinter.mainloop()

    def msg(self):
        tkinter.messagebox.showinfo('Carolanne!!',
         'This button has been clicked')

my_gui = Gui()
```

The Entry Widget

- **An entry widget presents the user with a single line input field that may be used to input string values**

- **Entry also supports methods to :**
  - Retrieve the value that was entered
  - Delete the value that was entered
  - Perform text selections

Syntax

```
e = Entry(parent = None, **options)
```

- Creates a Entry widget
- `parent` is the parent widget
- `**options` are entry configuration options

```
e.config(**options)
```

- Modifies one or more of the widget options
- If no options are given, returns a dictionary of all current widget options

```
e.method(**options)
```

- Use an entry widget method with optional options

Common Entry Widget Options

- **The following table enumerates some of the more common Entry widget options**

Option	Meaning
`bg, background`	The background color inside the entry area. Default is a light gray
`bd, borderwidth`	The width of the border around the entry area
`disabledbackground`	The background color to be displayed when the widget is in the disabled state
`disabledforeground`	The foreground color to be displayed when the widget is in the disabled state
`exportselection`	By default, if you select text within an Entry widget, it is automatically exported to the clipboard. To avoid this exportation, use exportselection=0
`fg, foreground`	The color used to render the text. Default is black
`font`	The font used for text entered in the widget by the user
`justify`	Defines how to align multiple lines of text. Use LEFT, RIGHT, or CENTER
`readonlybackground`	The background color to be displayed when the widget's state option is 'readonly'
`relief`	Specifies the appearance of a decorative border around the labelBorder decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE
`selectbackground`	The background color to use displaying selected text
`selectborderwidth`	The width of the border to use around selected text. The default is one pixel.
`selectforeground`	The foreground (text) color of selected text.
`show`	Normally, the characters that the user types appear in the entry. To make a "password" entry that displays characters as an asterisk, set show=`'*'`
`state`	The entry state: NORMAL, DISABLED, or "readonly" (same as DISABLED, but contents can still be selected and copied). Default is NORMAL. If you set this to DISABLED or "readonly", calls to insert and delete are ignored
`takefocus`	By default, the focus will tab through entry widgets. Set this option to 0 to take the widget out of the tab sequence
`textvariable`	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the StringVar class. This is an instance of a Control Variable. You can retrieve the text using `e.get()`, or set it using `e.set()`

Common Entry Widget Methods

- **The following table enumerates some of the common methods of the Entry widget**

Method	Purpose
`e.delete(first, last=None)`	Deletes characters from the widget, starting with the one at index `first`, up to but not including the character at position `last`. If the second argument is omitted, only the single character at position `first` is deleted
`e.get()`	Returns the entry's current text as a string
`e.icursor(index)`	Set the insertion cursor just before the character at the given index. This also sets the INSERT index
`e.index(index)`	Shift the contents of the entry so that the character at the given index is the leftmost visible character. Has no effect if the text fits entirely within the entry. Use `insert(INSERT, text)` to insert text at the cursor, `insert(END, text)` to append text to the widget
`e.insert(index, s)`	Inserts string `s` before the character at the given index
`e.select_clear()`	Clears the selection. If there isn't currently a selection, has no effect
`e.select_from(index)`	Sets the ANCHOR index position to the character selected by index, and selects that character
`e.select_present()`	If there is a selection, returns true, else returns false.
`e.select_range(start, end)`	Sets the selection under program control. Selects the text starting at the start index, up to but not including the character at the end index. The start position must be before the end position. To select all the text in an entry widget e, use `e.select_range(0, END)`
`e.select_to(index)`	Selects all the text from the ANCHOR position up to but not including the character at the given index

  - There are several ways to specify an index in an Entry widget:
    - Indexes start from 0
    - The constant END refers to the position after the existing text
    - The constant INSERT refers to the current position of the insertion cursor
    - The constant ANCHOR refers to the first character of the selection, if there is a selection

Common Entry Widget Patterns

Example

These code scraps represent common Entry Widget code patterns that are the basis of most of your Entry Widget Interactions

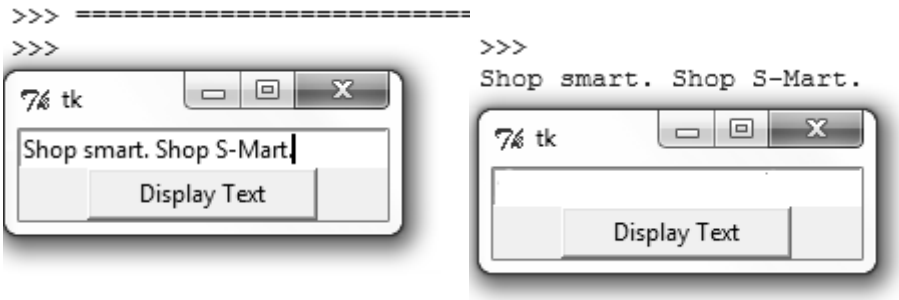
```
from tkinter import *

parent = Tk()
e = Entry(parent)

#Retrieve the value of an Entry widget
e.get()

#Delete the value of an Entry Widget
e.delete(0, END)

#Insert a value into an Entry Widget
e.insert(0, "a default value")

#Set Focus on an Entry Widget
e.focus_set()

#Set the value of an Entry Widget with a Control Variable
v = StringVar()
v.set("Default")
e = Entry(parent, textvariable=v)
e.pack()
parent.mainloop()
```

Example: Retrieve Data From An Entry Widget

Example

This simple example allows the user to press a button that prints the contents of the Entry widget to stdout

```
from tkinter import *

parent = Tk()
def display():
    print(e.get())
    e.delete(0,END)

e = Entry(parent, width = 30)
b = Button(parent, text="Display Text", width=15,
command=display)

e.pack()
b.pack()

e.focus_set()

mainloop()
```



Control Variables

- **A Tkinter control variable is a special object that acts like a regular Python variable in that it is a container for a value, such as a number or string**
  - Some widgets can be connected directly to control variables by using special options
- **Control Variables have the ability to hold values returned from widgets and they also have the ability to assign values to widgets**
  - These Tkinter control variables are used like regular Python variables to keep values.
- **It is not possible to hand over a regular Python variable to a widget through a variable or textvariable option; Control Variables must be used instead of Python Variables**

Syntax

```
x = StringVar()   # String; default value ""
x = IntVar()      # Integer; default value 0
x = DoubleVar()   # Float; default value 0.0
x = BooleanVar()  # Boolean, returns 0 for False and 1 for True
```

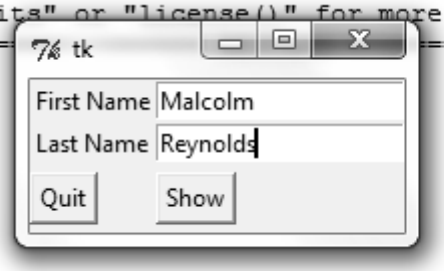
  - All control variables have these two methods:
    - `.get()` Returns the current value of the variable.
    - `.set(value)` Changes the current value of the variable. If any widget options are linked to this variable, those widgets will be updated when the main loop next idles

Example: Control Variables

Example

```
from tkinter import *

def displayfields():
   print("First Name: {}\nLast Name: {}".format(e1.get(),
         e2.get()))

parent = Tk()
Label(parent, text="First Name").grid(row=0)
Label(parent, text="Last Name").grid(row=1)

e1 = Entry(parent)
e2 = Entry(parent)

e1.grid(row=0, column=1)
e2.grid(row=1, column=1)

Button(parent, text='Quit',
command=parent.destroy).grid(row=3, column=0,
                      sticky=W, pady=4)
Button(parent, text='Show',
command=displayfields).grid(row=3, column=1,
                      sticky=W, pady=4)

mainloop( )
```

```
Type "copyright", "credits" or "license()" for more information.
>>> ==========
>>>
First Name: Malcolm
Last Name: Reynolds
```



Example: Kilometer Converter

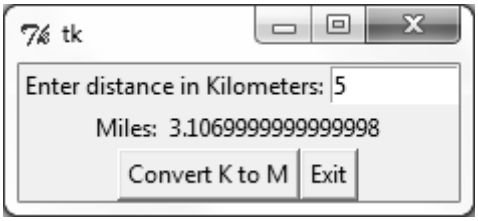
Example

This example ties together all of the concepts discussed in this module

```
import tkinter

class MileConverter:
    def __init__(self):

        self.main = tkinter.Tk()

        self.top = tkinter.Frame()
        self.mid = tkinter.Frame()
        self.bottom = tkinter.Frame()


        self.prompt = tkinter.Label(self.top,
             text='Enter distance in Kilometers:')
        self.kilo = tkinter.Entry(self.top, width=10)

        self.prompt.pack(side='left')
        self.kilo.pack(side='left')


        self.descr = tkinter.Label(self.mid, text='Miles:')

        self.value = tkinter.StringVar()

        self.miles = tkinter.Label(self.mid,
             textvariable=self.value)

        self.descr.pack(side='left')
        self.miles.pack(side='left')

        self.calc = tkinter.Button(self.bottom,
             text='Convert K to M', command=self.convert)
        self.quit = tkinter.Button(self.bottom,
             text='Exit', command=self.main.destroy)
```

Example: Kilometer Converter *cont'd*

```
        self.calc.pack(side='left')
        self.quit.pack(side='left')

        self.top.pack()
        self.mid.pack()
        self.bottom.pack()

        tkinter.mainloop()

    def convert(self):

        kilo = float(self.kilo.get())

        miles = kilo * 0.6214

        self.value.set(miles)


m = MileConverter()
```



Summary

- **Python supports a number of modules that may be used to create Graphic User Interfaces**
- **Python ships with the tkinter GUI module which is an interface to tk**
- **The primary object in tkinter is the Widget**
- **Geometry Managers are used to layout objects in tkinter**
- **tkinter supports a robust event handling system**
- **Control Variables may be used to retrieve values from widgets or assign values to widget**

Appendix A

Working the IDLE IDE

Python's IDLE IDE

- **IDLE is the built-in development environment for Python**
- **IDLE stands for "Integrated DeveLopment Environment"**
  - May be named for a member of the comedy group Monty Python's Flying Circus
- **IDLE is built completely in Python**
  - Uses the Tkinter GUI toolkit
- **Cross-Platform IDE**
  - Works on
    * Windows
    * Unix
    * Mac
- **Ships as a standard library module with Python**
  - The source code for IDLE is also included as part of the Standard Library
- **IDLE is extensible**
  - A number of extension modules for IDLE may be downloaded to extend IDLE's functionality

IDLE Features

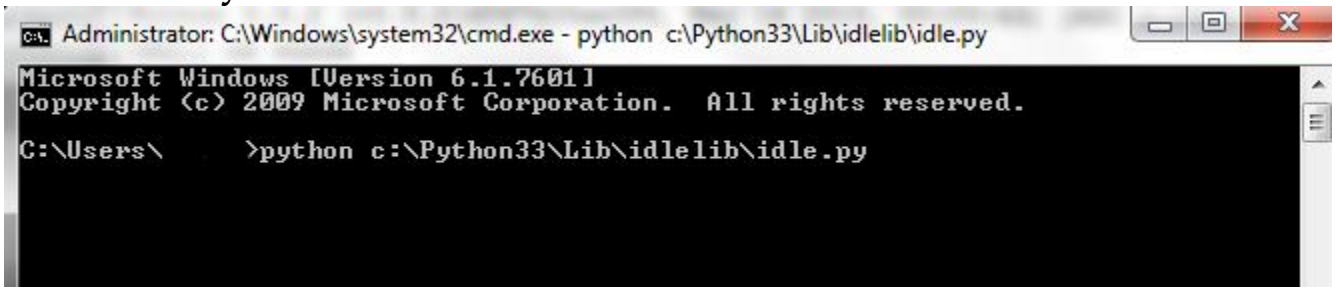
- **Multi-Window Code editor**
  - Syntax Highlighting
  - Auto Completion
  - Smart Indenting
- **Interactive Code Editing**
- **Integrated Debugger**
  - Ability to step in and out of code
  - Break Points
  - Call stack visibility
- **A good first IDE to use to learn to write Python**
  - Novice developers excel at learning Python with the IDLE IDE
  - Experienced developers benefit from the ability to write and execute small sets of code without having to write a full program

Benefits of using IDLE

- **IDLE provides developers with a number of Benefits**
  - Color Coded Syntax
  - Syntax Highlighting
  - Buffered Input
  - Automatic Indentation of code
  - Code/Word Completion
  - Write and Execute Code immediately
  - Replay code for testing
- **Familiar menus with keyboard shortcuts provide accessibility to IDLE's options**

Running IDLE

- **Running IDLE on Windows**
  - Navigate to Start > [All Programs/Program Files] > Python 3.x > IDLE (Python GUI)
- **Running IDLE on Mac OS X**
  - If you are using Mac OS X 10.8, 10.7 or 10.6, use IDLE or Tkinter from a 64-bit/32-bit Python installer
  - Navigate to the Python 3.x subfolder in the Applications folder and run IDLE
- **Running IDLE on Linux or Unix**
  - Bring up a shell and type `idle3.x` where x is the version of Python you have installed
  - If Python was installed through a package manager, you may have an IDLE entry in the Programming submenu
- **IDLE may also be run from its source directory**
  - `idle.py`
  - Typically installed to the Python directory, `idlelib` subdirectory



IDLE Command Line Options

Syntax

idle.py [-c command] [-d] [-e] [-s] [-t title] [arg] ...

-c command	Run Command. sys.argv[0] is set to -c, sys.argv[1:...] reflects the rest of the arguments passed
-d	Enable debugger
-e	Edit mode; arguments are files to be edited. sys.argv[] reflects the arguments passed to IDLE
-s	Run $IDLESTARTUP or $PYTHONSTARTUP first. Use IDLE Environmental variables at startup.
-t title	set title of shell window

Executing Code Interactively in IDLE

- **IDLE can be used to run code interactively or execute Python programs**

- **To work with IDLE interactively simply type python code at the IDLE prompt**
  - The >>> prompt is the interactive IDLE prompt

Example

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43)
[MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more
information.
>>> print("Hello, World!")
Hello, World!
>>>
```

- **The interactive editor gives Python programmers the ability to write code and immediately see the result of the code**

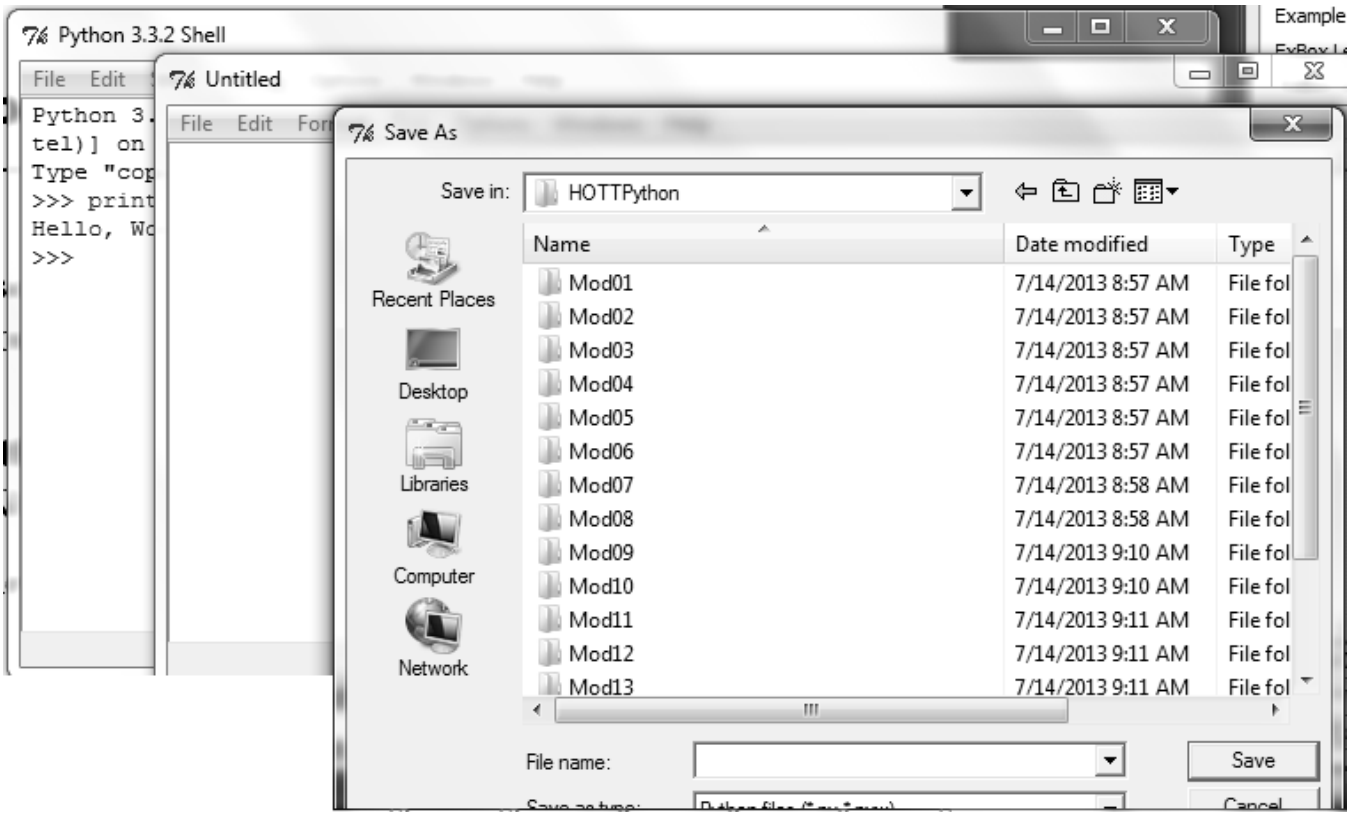
Writing Python Programs in IDLE

- **The IDLE IDE also gives programmers the ability to write and execute Python programs**

- **To start a new Python program click the `File` menu > `New Window (CTL +N)`**

  – This will open a new window to write a program

- **Save the program to a location on disk by clicking the `File` menu > `Save as` > choose a location to save your program**



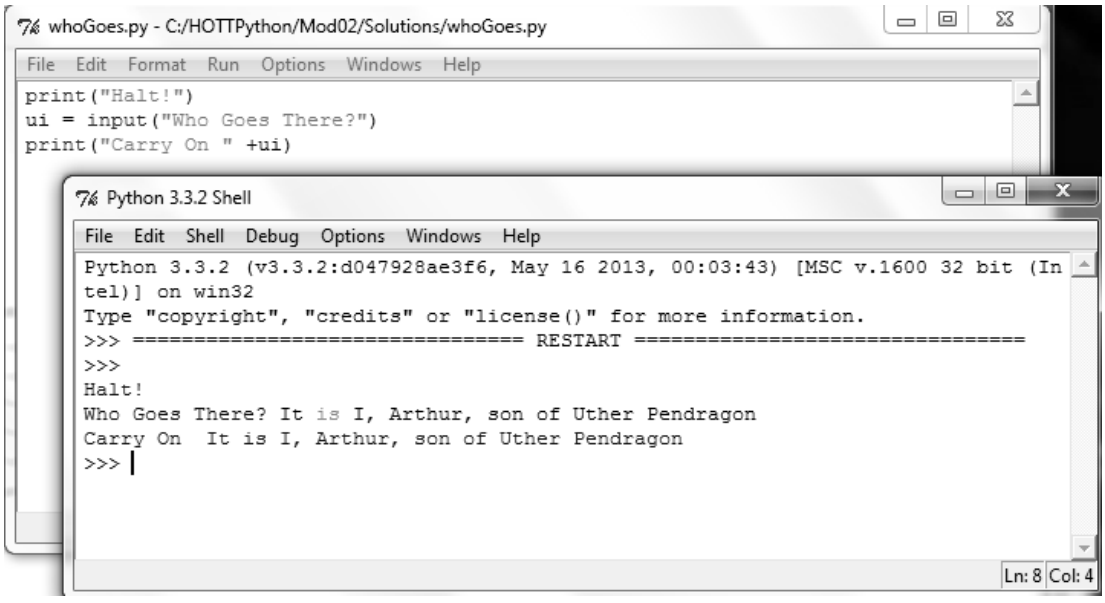
Executing Python Programs in IDLE

- **You may execute programs in IDLE by opening the files through the IDLE interface or by calling the files through the shell**
  - To run a program from the IDLE interface
  - Open the file by clicking File > Open | Open Module > then navigate to the location of your `.py` file

- **Execute the program by clicking** Run > Run Module **or press** F5 **on the keyboard**

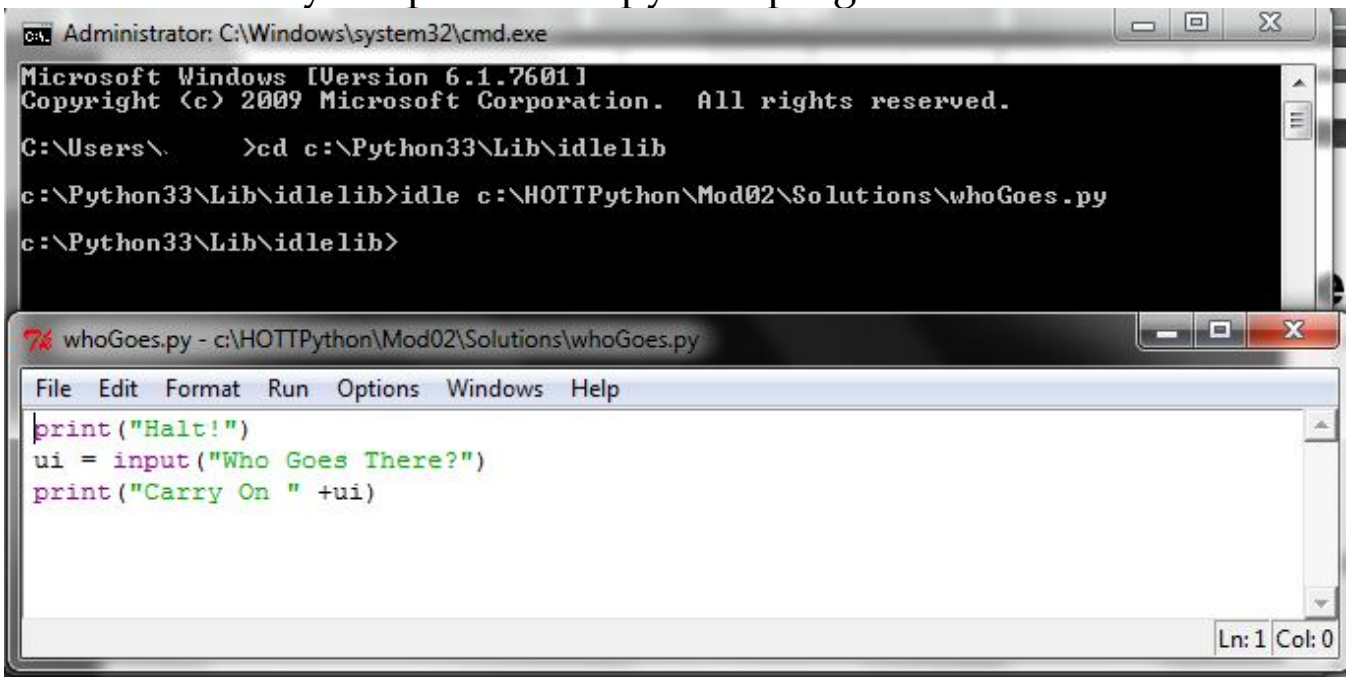
- **Alternatively you may check your syntax by clicking** Run > Check Module **or press** ALT + x **on the key board**

- **You may navigate between your source code and the interactive shell by using the Windows menu**
  - Windows > Source File

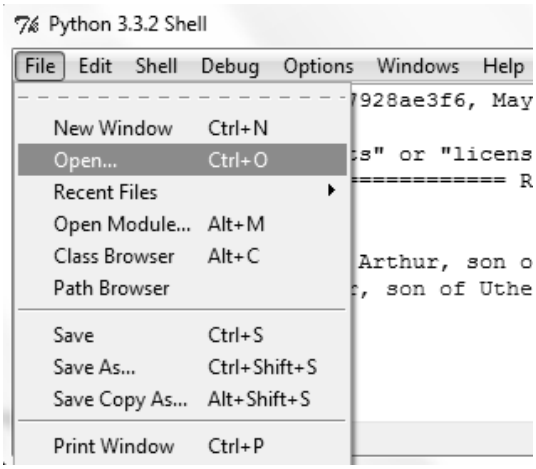


Load a program into IDLE from the shell

- **From the Shell**
  - Type the path to IDLE (or IDLE if IDLE is in the OS search path) followed by the path to the python program



- **From IDLE**
  - Click the File Menu > Open
  - Click the File Menu > Recent Files



Syntax Coloring in IDLE

- **IDLE uses syntax-directed colorization for the code typed in both the main window and all text edit windows**

- **Python Syntax Colors**

Keywords	**Orange**
Strings	**Green**
Comments	**Red**
Definitions	**Blue**
Miscellaneous Words	**Black**

- **Shell Colors**

Console Output	**Brown**
stdout	**Blue**
stderr	**Dark Green**
stdin	**Black**

Buffered Sessions in IDLE

- **Because your session is buffered, you may search through any code you have written**
  - You can replay any code you have previously written and executed
  - Place your cursor on the interesting line and press enter (hard return)
    * IDLE will copy the code on that line to the bottom of the screen
    * You may edit the code before executing or simply execute the code again by pressing enter
- **Commands can be toggled though by using keyboard shortcuts**
  - Alt+p toggles the previous command
  - Alt+n toggles the next command
  - Both commands copy code to the bottom of the interactive editor where you may edit and execute or just execute the commands again
- **Python keywords or User Defined values may be auto completed by using the `TAB` key**
  - If there is only one option for autocomplete, then Python autocompletes the token
  - Otherwise, after a predefined delay (2 seconds) or by pressing a `'.'` IDLE will open an AutoCompleteWindow (ACW)

Interrupting IDLE

- **You can press CTL+C on your keyboard to send an interrupt signal to IDLE**
  - This is useful if your code gets hung
  - Pressing CTL+C will interrupt IDLE and return you to a prompt
- **You may close IDLE by exiting through the file menu FILE > Exit**
- **You may also close IDLE by typing the `exit()` command in the interactive editor**
- **You may also close IDLE in Linux/Unix Operating Systems by pressing CTL+Z or pressing CTL+D in Windows Operating systems**

Changing IDLE Options

- **A number of IDLE's options may be changed by accessing the IDLE Preferences window through the Options menu**
  - Options > IDLE Preferences
- **The IDLE Preferences windows is subdivided into 4 tabbed groups**
  - Fonts/Tabs
  - Highlighting
  - Keys
  - General
- **Python's default indentation width of 4 spaces is configured from the Fonts/Tabs tab**
- **A list of keyboard shortcuts may be accessed or created in the Keys tab**
- **The Highlighting tab allows a developer to change the default syntax highlighting style**
- **The General tab allows the customization of a number of preferences including:**
  - Opening the Edit window or Shell window at startup
  - Autosave Preferences

Getting Help with IDLE

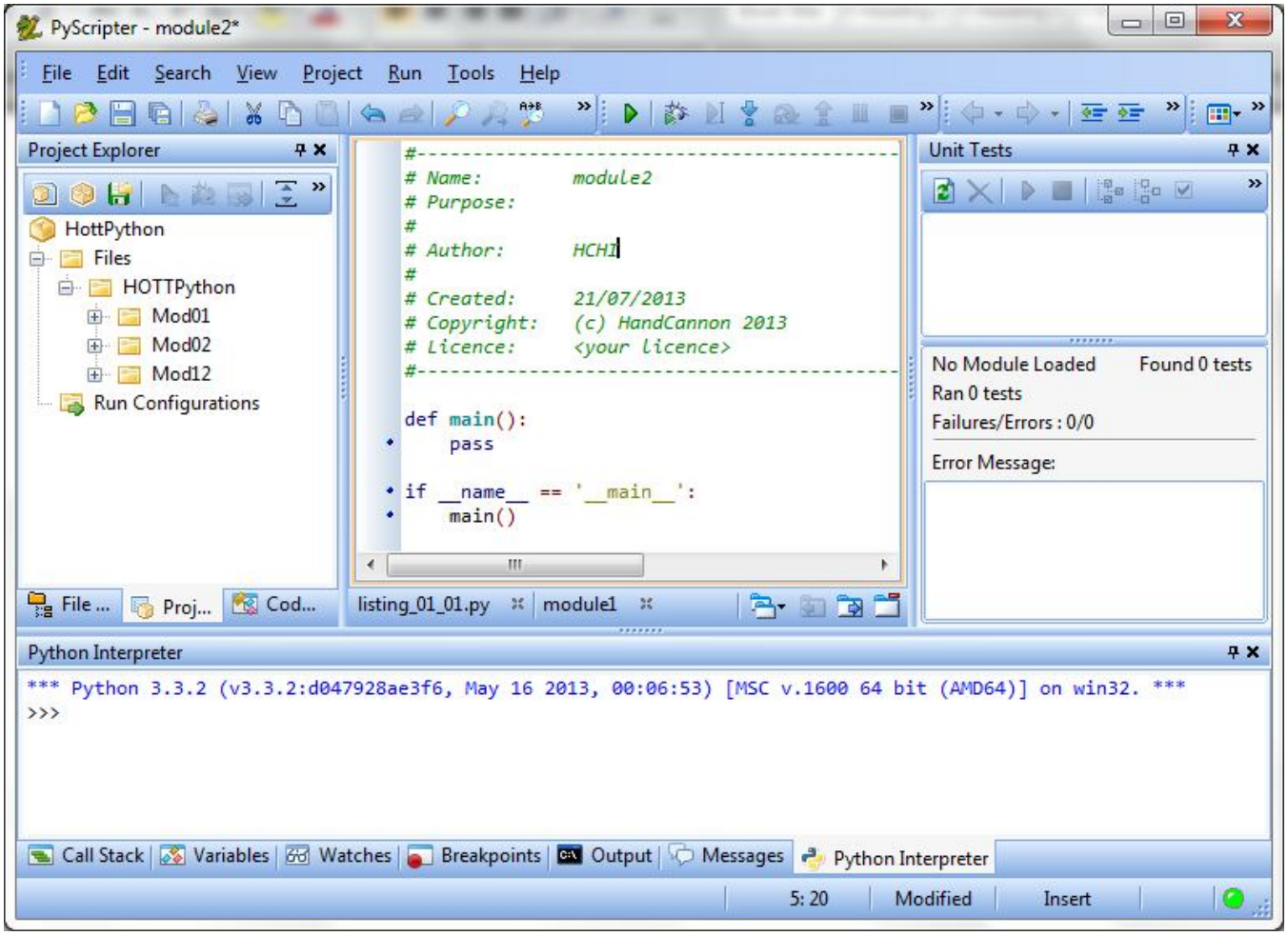
- **To access the IDLE help system**
  - Click the Help Menu > IDLE Help
- **You may also access the Python documentation through IDLE**
  - Click the Help Menu > Python Docs
  - Click F1 on the keyboard

PyScripter

- **PyScripter is a free, light-weight, Python IDE**
  - Built in Delphi
  - Extensible through the use of Python scripts
  - Currently only available on Windows platforms

- **PyScripter has a number of features beneficial for developers**
  - An uncluttered, easy to navigate interface
  - Integrates with the Python Interpreter
  - Integrated Remote Python Debugger
  - Unicode Support
  - Syntax Highlighting
    * Code Completion
    * Call Tips
    * Code Explorer
  - Command History
  - Watch Window
  - Code Explorer
  - Internal File Explorer
  - Ability to create and Manage Projects

PyScripter User Interface



PyScripter Windows: Interpreter, File Explorer and Code Explorer

- **Python Interactive Interpreter**
  - Command History
  - Code Completion
  - Call Tips
  - Acts as the standard output window of active scripts
- **The File Explorer Window**
  - Provides a powerful file explorer similar to the Windows Explorer
  - Provides filtering capabilities and a favorites list
- **The Code Explorer Window**
  - Displays a structured (tree) view of the source code
  - Shows relationships between code
  - Selecting code in the Code Explorer selects the code in the editor
  - Can locate the definition code for call statements
  - Can locate all references to an identifier

PyScripter Windows: Interpreter, File Explorer and Code Explorer

–

- **The Project Explorer Window**
  - Hierarchically groups together collections of related files into folders and subfolders
  - Folder and subfolder organization may be logical. IE a workspace of files
  - Can create and maintain Project Configurations
  - Commands may be executed from the Project Explorer Window
- **The Messages Windows**
  - Logs and displays messages to the developer
  - Syntax and Runtime Errors
  - Python trace back and warning information
  - File location errors
  - Maintains history of 10 viewable messages logs
- **The Output Window**
  - Shows the captured output from external tools
  - Such as
    - Command Prompt , PyLint Style Checker, External Python Interpreter, Profiler

PyScripter Windows: Find-in-Files, To-do list and Unit Tests

- **Find-in-Files Window**
  - Displays the results of a Find-in-Files Search
  - Supports multi-file search and replace
  - Uses a folding display to expand and collapse results
  - Supports Regular Expressions for Find and Replace

- **The To-do List Window**
  - The To Do List window allows you to create a task list of source items you need to work on
  - To add items to the To Do List, add shell style comments to your code

```
* #ToDo1 Rework this code to function on Windows 8
```

- **The Unit Tests Window**
  - The Unit Tests Window provides an advanced GUI for running tests based on the standard Python module unittest
  - Tests can be run, stopped, cleared, refreshed, selected and deselects
  - Failed tests can be identified and selected

Debugger Windows: Call Stack and Variables

- **The Call Stack Window**
  - The call stack window displays the Python interpreter call stack while debugging
  - Shows the function name and the corresponding source code position for each stack frame
  - You may jump to a given code position by double-clicking on a stack frame line

- **The Variables Window**
  - During debugging and while the interpreter is stopped at a breakpoint , the Variables window displays the local and global variables for the selected stack frame in the Call Stack window
  - The left pane shows a hierarchical view with the value of each variable
  - Any Python object with a dictionary interface (classes, objects, dictionaries etc.) can be expanded so that key-value pairs are inspected
  - Variables that have been changed or are new while stepping through code are color coded
  - Changed variables are displayed with red color and new variables with blue color
  - The left hand pane of the Variables window displays the type, value and documentation of the selected variable

Debugger Windows: Watches and Breakpoints

- **The Watches Window**
  - This provides typical "Watch Expression" functionality found in most debuggers. You can set watches for arbitrary Python expressions
  - These expressions get re-evaluated as you step through the code or when you stop at breakpoints
- **The Breakpoints Window**
  - This window shows the breakpoints in all open Python scripts and modules
  - Double-clicking on a specific breakpoint takes you to the given code position
  - You can enable/disable a breakpoint by checking/unchecking the check-box at the start of the corresponding row
  - You can also apply a condition by specifying a Python expression using the context menu

Running Scripts

- **The Run menu in PyScripter provides a number of different commands to execute your Python code**

- **Debug using the internal integrated Python debugger**
  - Set any breakpoints you need
    - Run > Toggle Break Point or Shortcut key F5
  - Click Run > Debug
  - Shortcut key F9

- **Run using the Internal Python Interpreter**
  - Click Run > Run
  - Shortcut key CTL + F9

- **Specifying Command Line parameters**
  - To specify command line parameters for your program click Run > Command Line Parameters
  - A dialog box is displayed where you may specify the command line parameters you want to use