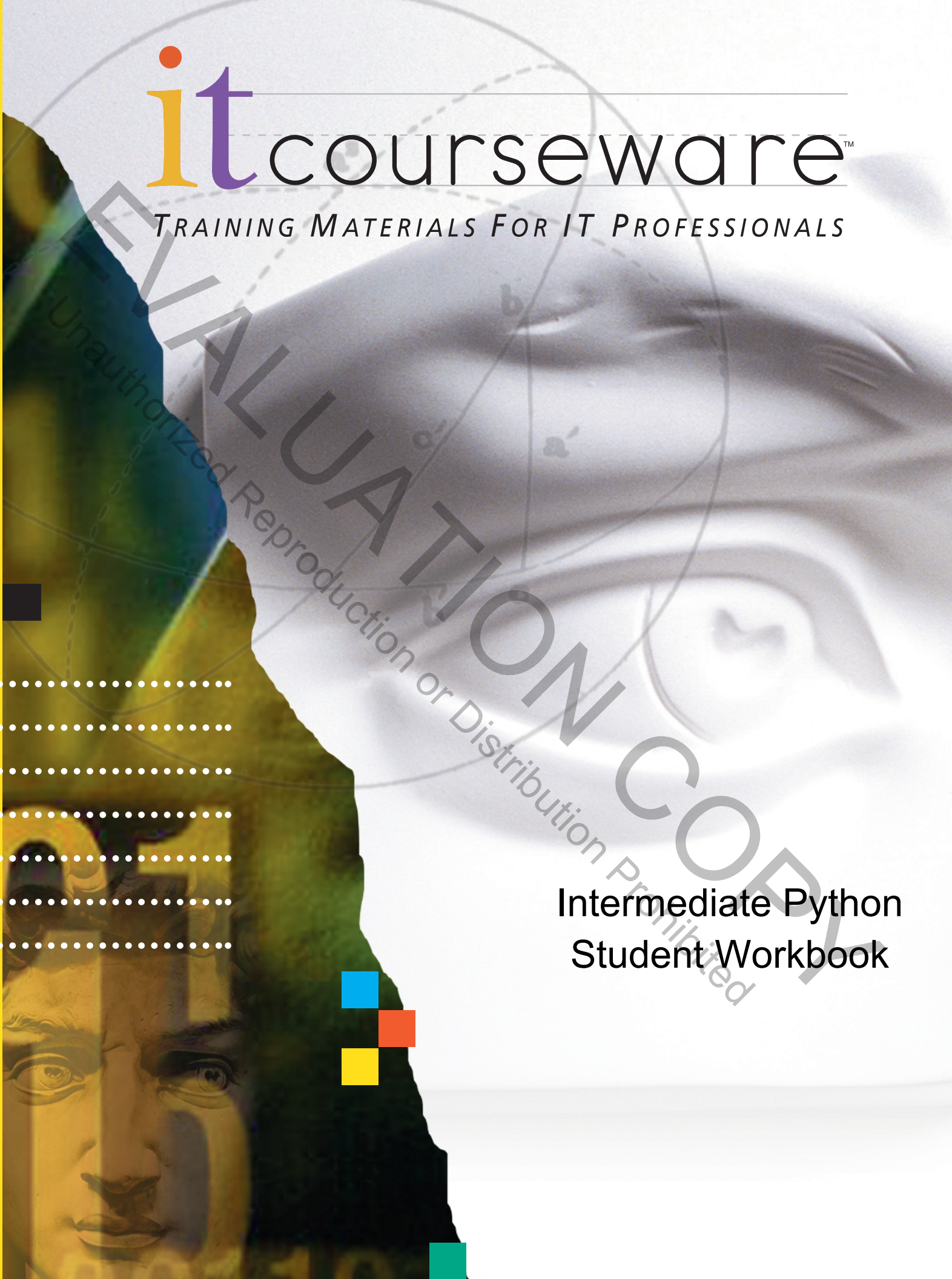

itcourseware™

TRAINING MATERIALS FOR IT PROFESSIONALS

Intermediate Python
Student Workbook

Intermediate Python 3

John Strickler

Version 2.0, March 2020

Table of Contents

About this course **1**

Welcome! 2

Classroom etiquette 3

Course Outline 4

Student files 5

Extracting the student files 6

Examples 7

Lab Exercises 8

Appendices 9

Chapter 1: Python Refresher **11**

Objectives 11

Variables 12

Basic Python Data types 14

Sequence Types 15

Mapping Types 17

Program structure 18

Files and console I/O 20

Conditionals 21

Loops 22

Builtins 25

Functions 26

Modules 27

Packages 28

Chapter 2: OS Services **31**

Objectives 31

The os module 32

Paths, directories and file names 37

Walking directory trees 43

Environment variables 46

Launching external programs 48

Chapter 3: Dates and Times **51**

Objectives 51

Python modules for dates and times . . . 52
Ways to store dates and times . . . 53
Basic dates and times . . . 54
Formatting dates and times . . . 57
Parsing date/time strings . . . 61
Parsing dates the easier way . . . 63
Converting dates and times . . . 66
Time zones . . . 72
Generating calendars . . . 73
Chapter 4: Binary Data . . . 79
Objectives . . . 79
"Binary" (raw, or non-delimited) data . . . 80
Binary vs Text data . . . 81
Using Struct . . . 82
Bitwise operations . . . 88
Chapter 5: Pythonic Programming . . . 95
The Zen of Python . . . 96
Tuples . . . 97
Iterable unpacking . . . 98
Unpacking function arguments . . . 100
The sorted() function . . . 104
Custom sort keys . . . 105
Lambda functions . . . 110
List comprehensions . . . 112
Dictionary comprehensions . . . 114
Set comprehensions . . . 116
Iterables . . . 117
Generator Expressions . . . 119
Generator functions . . . 121
String formatting . . . 123
f-strings . . . 125
Chapter 6: Functions, Modules and Packages . . . 129
Functions . . . 130
Function parameters . . . 133

Default parameters . . . 137
Python Function parameter behavior (from PEP 3102) . . . 139
Name resolution (AKA Scope) . . . 140
The global statement . . . 143
Modules . . . 144
Using import . . . 145
How *import ** can be dangerous . . . 149
Module search path . . . 151
Executing modules as scripts . . . 152
Packages . . . 154
Configuring import with __init__.py . . . 156
Documenting modules and packages . . . 159
Python style . . . 160
Chapter 7: Intermediate Classes . . . 163
What is a class? . . . 164
Defining Classes . . . 165
Object Instances . . . 166
Instance attributes . . . 167
Instance Methods . . . 168
Constructors . . . 170
Getters and setters . . . 171
Properties . . . 172
Class Data . . . 175
Class Methods . . . 177
Inheritance . . . 179
Using super() . . . 180
Multiple Inheritance . . . 185
Abstract base classes . . . 188
Special Methods . . . 191
Static Methods . . . 197
Chapter 8: Metaprogramming . . . 199
Objectives . . . 199
Metaprogramming . . . 200
globals() and locals() . . . 201
The inspect module . . . 204

Working with attributes . . . 207
Adding instance methods . . . 210
Decorators . . . 213
Applying decorators . . . 214
Trivial Decorator . . . 217
Decorator functions . . . 218
Decorator Classes . . . 221
Decorator parameters . . . 225
Creating classes at runtime . . . 228
Monkey Patching . . . 232
Callable classes . . . 235
Do you need a Metaclass? . . . 237
About metaclasses . . . 238
Mechanics of a metaclass . . . 240
Singleton with a metaclass . . . 244
Chapter 9: Developer Tools . . . 249
Objectives . . . 249
Program development . . . 250
Comments . . . 251
pylint . . . 252
Customizing pylint . . . 253
Using pyreverse . . . 254
The Python debugger . . . 256
Starting debug mode . . . 257
Stepping through a program . . . 258
Setting breakpoints . . . 259
Profiling . . . 260
Benchmarking . . . 262
Chapter 10: Unit Tests with pytest . . . 267
Objectives . . . 267
What is a unit test? . . . 268
The pytest module . . . 269
Creating tests . . . 270
Running tests (basics) . . . 271
Special assertions . . . 272

Fixtures . . . 274
User-defined fixtures . . . 275
Builtin fixtures . . . 277
Configuring fixtures . . . 281
Parametrizing tests . . . 284
Marking tests . . . 287
Running tests (advanced) . . . 289
Skipping and failing . . . 291
Mocking data . . . 294
pymock objects . . . 295
Pytest and Unittest . . . 302
Chapter 11: Database Access . . . 305
Objectives . . . 305
The DB API . . . 306
Connecting to a Server . . . 307
Creating a Cursor . . . 310
Executing a Statement . . . 311
Fetching Data . . . 312
SQL Injection . . . 315
Parameterized Statements . . . 317
Dictionary Cursors . . . 325
Metadata . . . 329
Transactions . . . 332
Object-relational Mappers . . . 333
NoSQL . . . 334
Chapter 12: PyQt . . . 341
Objectives . . . 341
What is PyQt? . . . 342
Event Driven Applications . . . 343
External Anatomy of a PyQt Application . . . 345
Internal Anatomy of a PyQt Application . . . 346
Using designer . . . 347
Designer-based application workflow . . . 348
Naming conventions . . . 350
Common Widgets . . . 351

Layouts . . . 354
Selectable Buttons . . . 356
Actions and Events . . . 357
Signal/Slot Editor . . . 361
Editing modes . . . 362
Menu Bar . . . 363
Status Bar . . . 364
Forms and validation . . . 366
Using Predefined Dialogs . . . 369
Tabs . . . 373
Niceties . . . 375
Working with Images . . . 376
Complete Example . . . 379
Chapter 13: Network Programming . . . 383
Objectives . . . 383
Grabbing a web page . . . 384
Consuming Web services . . . 388
HTTP the easy way . . . 391
sending e-mail . . . 398
Email attachments . . . 401
Remote Access . . . 405
Copying files with Paramiko . . . 408
Chapter 14: Multiprogramming . . . 413
Objectives . . . 413
Multiprogramming . . . 414
What Are Threads? . . . 415
The Python Thread Manager . . . 416
The threading Module . . . 417
Threads for the impatient . . . 418
Creating a thread class . . . 420
Variable sharing . . . 423
Using queues . . . 426
Debugging threaded Programs . . . 429
The multiprocessing module . . . 431
Using pools . . . 435

Alternatives to multiprogramming . . . 441
Chapter 15: Effective Scripts . . . 443
Using **glob** . . . 444
Using shlex.split() . . . 446
The subprocess module . . . 447
subprocess convenience functions . . . 448
Capturing stdout and stderr . . . 451
Permissions . . . 454
Using shutil . . . 456
Creating a useful command line script . . . 459
Creating filters . . . 460
Parsing the command line . . . 463
Simple Logging . . . 468
Formatting log entries . . . 470
Logging exception information . . . 473
Logging to other destinations . . . 475
Chapter 16: Serializing Data . . . 479
About XML . . . 480
Normal Approaches to XML . . . 481
Which module to use? . . . 482
Getting Started With ElementTree . . . 483
How ElementTree Works . . . 484
Elements . . . 485
Creating a New XML Document . . . 488
Parsing An XML Document . . . 491
Navigating the XML Document . . . 492
Using XPath . . . 496
About JSON . . . 500
Reading JSON . . . 501
Writing JSON . . . 504
Customizing JSON . . . 507
Reading and writing YAML . . . 511
Reading CSV data . . . 516
Nonstandard CSV . . . 518
Using csv.DictReader . . . 520

Writing CSV Data . . . 522
Pickle . . . 524
If time permits... . . . 529
Chapter 17: Advanced Data Handling . . . 531
Objectives . . . 531
Deep vs shallow copying . . . 532
Default dictionary values . . . 535
Counting with Counter . . . 538
Named Tuples . . . 539
Printing data structures . . . 542
Zipped archives . . . 545
Tar Archives . . . 547
Serializing Data . . . 550
Chapter 18: Type Hinting . . . 555
Objectives . . . 555
Type Hinting . . . 556
Static Analysis Tools . . . 557
Runtime Analysis Tools . . . 558
`typing` Module . . . 560
Input Types . . . 563
Variance . . . 565
`Union` and `Optional` . . . 567
`multimethod` and `functools.singledispatch` . . . 568
Stub Type Hinting . . . 570
Appendix A: Python Bibliography . . . 573
Appendix B: Virtual Environments . . . 579
What are virtual environments? . . . 580
Preparing the virtual environment . . . 581
Creating the environment . . . 581
Activating the environment . . . 582
Deactivating the environment . . . 583
Freezing the environment . . . 584
Duplicating an environment . . . 585
The pipenv/conda/virtualenv/PyCharm swamp . . . 586

Multiple Python versions with pyenv 587
Index **589**

About this course

Welcome!

- We're glad you're here
- Class has hands-on labs for nearly every chapter
- Please make a name tent

Instructor name:

Instructor e-mail:

Have Fun!

Classroom etiquette

- Noisemakers off
- No phone conversations
- Come and go quietly during class.

Please turn off cell phone ringers and other noisemakers.

If you need to have a phone conversation, please leave the classroom.

We're all adults here; feel free to leave the clasroom if you need to use the restroom, make a phone call, etc. You don't have to wait for a lab or break, but please try not to disturb others.

IMPORTANT Please do not bring any exploding penguins to class. They might maim, dismember, or otherwise disturb your fellow students.

Course Outline

Day 1

Chapter 1 Python Refresher
Chapter 2 OS Services
Chapter 3 Dates and Times
Chapter 4 Binary Data

Day 2

Chapter 5 Pythonic Programming
Chapter 6 Functions, Modules, and Packages
Chapter 7 Intermediate Classes
Chapter 8 Metaprogramming

Day 3

Chapter 9 Developer tools
Chapter 10 Unit Testing with PyTest
Chapter 11 Database access
Chapter 12 PyQt

Day 4

Chapter 13 Network Programming
Chapter 14 Multiprogramming
Chapter 15 Scripting for System Administration
Chapter 16 Serializing Data

Time Permitting

Chapter 17 Advanced Data Handling
Chapter 18 Type hinting

NOTE The actual schedule varies with circumstances. The last day may include *ad hoc* topics requested by students

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **py3interm**.

What's in the files?

py3interm contains data and other files needed for the exercises
py3interm/EXAMPLES contains the examples from the course manuals.
py3interm/ANSWERS contains sample answers to the labs.

NOTE The student files do not contain Python itself. It will need to be installed separately. This has probably already been done for you.

Extracting the student files

Windows

Open the file **py3interm.zip**. Extract all files to your desktop. This will create the folder **py3interm**.

Non-Windows (includes Linux, OS X, etc)

Copy or download py3interm.tgz to your home directory. In your home directory, type

```
tar xzvf py3interm.tgz
```

This will create the **py3interm** directory under your home directory.

If your version of Unix is elderly, its tar command may not support the z option. If this is so, use this command line instead:

```
gzip -dc py3interm.tgz | tar xvf -
```

Examples

Nearly all examples from the course manual are provided in EXAMPLES subdirectory. Many of the examples have callouts — numbers that refer to notes just below the code.

It will look like this:

Example

cmd_line_args.py

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

import sys ①

print(sys.argv) ②

name = sys.argv[1]  ③
print("name is", name)
```

① Import the **sys** module

② Print all parameters, including script itself

③ Get the first actual parameter

cmd_line_args.py apple mango 123

```
['/Users/jstrick/curr/courses/python/examples3/cmd_line_args.py', 'apple', 'mango',
'123']
name is apple
```

Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in py3interm/ANSWERS

Appendices

- Appendix A: Python Bibliography
- Appendix B: Virtual Environments

Chapter 1: Python Refresher

Objectives

- Refresh basic (intro-level) Python concepts

Variables

- Declared by assignment
- Dynamic typing

Variables are declared by assigning to them. Python does not require explicit type specifiers, but sets the type implicitly by examining the value that was assigned.

Thus, assigning a literal integer to a variable creates a variable of type int, while assigning quoted text to a variable creates a variable of type string. Once a variable is assigned to, it will cause an error if the variable is used with an operator or function that is inappropriate for the type.

A variable cannot be used before it is assigned to.

Variables must be assigned **some** value. A value of **None** may be assigned if no particular value is needed.

Names may contain only letters, digits, and underscores, and may not start with a digit. The Python convention for variable names is `all_lower_case_words_with_underscores`.

A "variable" is really an object with a name assigned to it. What we think of as the variable is the name. Objects may have more than one name.

```
x = 10
y = x
print(x)
print(y)
```

This creates an object of type **int**, accessible via both the name **x** and the name **y**. Both names refer to the same object.

Example

```
name = 'Fred Flintstone'
count = 0
name = 'Fred Flintstone'
colors = [ 'red', 'purple', 'green' ]
```

Basic Python Data types

- Python has many data types
- Use builtin functions to convert

Python has many data types. There are builtin functions to convert from one type to another. If the source type cannot be converted to the target type, a TypeError is thrown.

Numeric types

- bool
- int
- float
- complex

Sequence types

- str
- bytes
- list
- tuple

Mapping types

- dict
- set
- frozenset

Sequence Types

Strings are text (arrays of Unicode characters)

```
s = "text";
```

Bytes are arrays of bytes

```
b = b"text";
```

Lists are sequences of values

```
my_list = [1
sequence[start:limit:stride]
```

Tuples are *readonly* sequences of values (used as records)

```
my_tuple = 'Mary', 'Poppins', 'London'
```

Python supports four types of sequences – strings, bytes, lists, and tuples. All sequences share a common set of operations, methods, and builtin functions; each type also has operations specific to that type.

A str object is a list of Unicode characters. A **bytes** object is a list of bytes.

All sequences support slicing, which means returning a subset of the sequence using the [**start**:*limit*:*step*] syntax.

Example

```
colors = [ 'red', 'green', 'blue', 'purple', 'pink', 'yellow', 'black' ]
c1 = colors[0]    #  'red'
c2 = colors[1:4]  #  'green', 'blue', 'purple'
c3 = colors[-1]   # 'black'
c4 = colors[:3]   # 'red', 'green', 'blue'
c5 = colors[3:]   # 'purple', 'pink', 'yellow', 'black'
```

Table 1. Slicing syntax

sequence[START:STOP]	START to STOP - 1
sequence[START:]	START to end
sequence[:STOP]	beginning to STOP - 1
sequence[START:STOP:STEP]	START to STOP -1 counting by STEP
sequence[:]	all elements
sequence[::]	all elements
sequence[::STEP]	all elements counting by STEP

NOTE Remember that the starting value of a slice is ***IN***clusive, while the ending value is ***EX***clusive.

Mapping Types

- Dictionaries are mapped sets of values
- Sets are similar to dictionaries but contain only keys
- Syntax

```
d = { }
s = set()
f = frozenset()
```

Python also supports mapping types — dictionaries and sets.

A dictionary (**dict**) is a set of values indexed by an immutable keyword. Dictionaries are used for many tasks, including mapping one set of values to another, and counting occurrences of values. . Prior to version 3.6, dictionaries were unordered, but beginning with 3.6, dictionaries preserve the order in which items are added.

Dictionary keys must be *hashable*, which means in general that they must be immutable. This means that most dictionary keys are strings, but can be numbers, or tuples of immutable types.

A set is an unique collection of values. There are two types — the normal set is dynamic (mutable), and a frozenset is fixed (immutable), like a tuple.

Program structure

- All imports at top
- Variables, functions, and classes must be declared before use
- Main function goes at top
- Main function *called* at bottom

In Python, modules must be imported before their contents may be accessed. Variables, Functions, and classes must be declared before they can be used. Thus most scripts are ordered in this way:

1. import statements
2. global variables
3. main function
4. functions
5. call to main function

You may want to make a template for your Python scripts. Most editors and IDEs support templates or code snippets.

In PyCharm, go to **Settings→Editor→File and Code Templates** to create a new file template.

Example Script Format

script_template.py

```
#!/usr/bin/env python
"""
This is the doc string for the module/script.
"""
import sys

# other imports  (standard library, standard non-library, local)

# constants (AKA global variables -- keep these to a minimum)

# main function
def main(args):
    """
    This is the docstring for the main() function

    :param args: Command line arguments.
    :return: None
    """
    function1()

# other functions
def function1():
    """
    This is the docstring for function1().

    :return: None
    """
    pass

if __name__ == '__main__':
    main(sys.argv[1:])  # Pass command line args (minus script name) to main()
```

TIP copy/paste this script to create new scripts

Files and console I/O

- print()
- open()
- input()

Screen output

To output to the screen, use the **print()** function. print() normally outputs a newline after its arguments, this can be controlled with the **end** parameter.

print() puts spaces between its arguments by default. To use a different separator, set the **sep** parameter to the desired separator, which might be an empty string.

Reading files

To read a file, open it with the open() function as part of a **with** statement.

To read it line by line, iterate through the file with a **for** loop. To read the entire file, use file.read(); to read all the lines into a list, use file.readlines(). To read a specified number of bytes, use file.read(n).

To navigate within a file, use file.seek(offset, whence); to get the current location, use file.tell().

User input

To get input from the user, use input(). It provides a prompt to the user, and returns a string, with the newline already trimmed.

```
file_name = input("What file name? ")
```


Conditionals

- Test a Boolean value
- if-elif-else

The conditional statement in Python, like most languages, is if. There are several variations on how if is used. All depend on testing a value to see whether it is true or false.

The following values are false:

- False
- Empty collections (empty string, empty list, empty dictionary, empty set, etc.)
- Numeric zero (0 or 0.0)

Just about everything else is true. (User-defined objects, and many builtin objects are true. If you create a class, you can control when it it true, and when it is false.)

Python has a shortcut if-else that is something like the A?B:C operator in C, Perl and other curly-brace languages

```
value1 if condition else value2
```

Example

```
if name == 'root:
    print("do not run this utility as root")
elif name == 'guest':
    print("sorry - guests are not allowed to run this utility")
else:
    print("starting processing")

limit = sys.args[1] if len(sys.args) > 1 else 100
```

Loops

- Two kinds of loops
  - **while** waits for condition
  - **for** iterates over a sequence (iterable)

Python has two kinds of loops.

The **while** loop is used for reading data, typically from a database or other data source, or when waiting for user input to end a loop.

The **for** loop is used to iterate through a sequence of data. Because Python uses iterators to simplify access to many kinds of data, the for loop is used in places that would use while in most languages. The most frequent example of this is in reading lines from a file.

while and **for** loops can also have an **else** block, which is always executed unless a **break** statement is executed.

Example

loops_ex.py

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

colors = ['red', 'green', 'blue', 'purple', 'pink', 'yellow', 'black']  ①

for color in colors:  ②
    print(color)
print()

with open('../DATA/mary.txt') as mary_in:  ③
    for line in mary_in:  ④
        print(line, end='')  ⑤
print()

while True:  ⑥
    name = input("What is your name? ")  ⑦
    if name.lower() == 'q':
        break  ⑧
    print("Welcome,", name)
```

① create a list

② loop over list

③ open text file for reading

④ loop over lines in file

⑤ print line with extra newline

⑥ loop "forever"

⑦ read input from keyboard

⑧ exit loop

loops_ex.py

```
red
green
blue
purple
pink
yellow
black

Mary had a little lamb,
Its fleece was white as snow,
And everywhere that Mary went
The lamb was sure to go

What is your name?  Fred
Welcome, Fred
What is your name?  Amir
Welcome, Amir
What is your name?  Jacinto
Welcome, Jacinto
What is your name?  q
```

Builtins

- 73 builtin functions (as of Python 3.7)
- Not called from an object or package
- Can work on many different data types

Python has many *builtin* functions. These provide generic functionality that is not tied to a particular type or package.

They can be applied to many different data types, but not all functions can be applied to all data types.

In some languages they might be called *static methods*.

Functions

- Declared with **def**
- Two reasons
  - Refactor duplicate code
  - Make code modular
- Parameters
  - positional or named
  - required or optional
- Default return value **None**

Functions are critical to any language. They are used to isolate code which is used in more than one place, as well as to organize code into manageable chunks.

Functions may be called with arguments, which are copied into parameters that are part of the function definition.

Functions always return *something*. The **return** statement returns any value (any Python object). If there is no **return** statement, a function returns **None**.

Modules

- No different from scripts
- Refactor duplicate code
- Share code among scripts
- Define PYTHONPATH to add search folders
- If imported
  - __name__ set to "*modulename*"
- If executed directly
  - __name__ set to "__main__"

Modules are used to share code among multiple scripts. They allow you to isolate code in one place. They are also used to organize the code in a project, even if there is no shared code.

There is technically no difference between a "module" and a "script". The only difference is how they are used. A *module* is imported by some other Python file. A *script* is run directly from the Python interpreter. A file can be used as either one.

Modules can be imported with one of the following forms:

```
import amodule
from amodule import afunc1, afunc2
```

To specify the folder a module should be loaded from, define an environment variable PYTHONPATH which contains one or more folders separated by semicolon (Windows) or colon (Unix/Linux/Mac).

A file can know which way it's being used by checking the value of the **__name__** variable. If this variable is set to "__main__", then the script is being run directly. If it is set to the name of the module, then it's being imported.

Packages

- Really just a folder
- Can contain modules or other packages
- Become prefixes for modules
- __init__ is optional, and can contain
  - Doc strings
  - Code common to modules in package
  - Import statements for convenience
  - Code shaed by all modules in package

A package is a folder than contains modules or other packages.

Packags are usually arranged in at least two levels, with the top level being the name of the organization. The next level contains functional or or other divisions, which then contain modules. This hierarchy can be nested as deeply as desired.

Given the following structure, making sure that the *parent* of **mycorp** is in PYTHONPATH:

```
mycorp
├──── eng
│     └──── calc.py
└──── mkt
      └──── sales.py
```

To import the **sales** module, use `from mycorp.mkt import sales`.

To import functions from the **calc** module, use `from mycorp.eng.calc import func1, func2`.

NOTE In Python 2, __init__ was mandatory for packages; in Python 3, it's optional.

Chapter 1 Exercises

Exercise 1-1 (pres_by_state.py, pres_by_state_sorted.py)

Using the file presidents.txt (in the DATA folder), count the number of Presidents who were born in each state. In other words, the output of your script should be a list, sorted by state name, with the state and the number of presidents that were born in that state:

TIP First declare a dictionary to hold the data. Then read the file in one line at a time. Split each line into fields using a colon as the separator. Add/update the element of the dictionary where the key is the state. Add 1 each time the state occurs.

expected output

```
Arkansas                1
California              1
Connecticut             1
Georgia                 1
etc
```

Exercise 1-2 (pres_dates.py, pres_dates_amb.py)

Write an interactive script that asks for a president's last name. For each president whose last name matches, print out their date of birth and date of death. For presidents who are still alive, print three asterisks for the date.

NOTE Dates of death and term end date might be the string "NONE".

For the ambitious

1. Make the name search case-insensitive
2. Change your script to print out matches for partial names – so "jeff" would find "Jefferson", e.g.

Chapter 2: OS Services

Objectives

- Working with the OS
- Running external programs
- Walking through a directory tree
- Working with path names

The os module

- Provides OS-specific services

The **os** module provides many basic services from the operating system. The interface is the same for different operating systems. These services include file and folder utilities, as well as working with dates and times, running external programs, and many others.

Table 2. The os module

Method or Data	Description
path	either posixpath or ntpath
ctermid()	Return name of the controlling terminal
device_encoding()	Return string describing the encoding of the device
dup()	Return a duplicate of a file descriptor.
dup2()	Duplicate file descriptor.
exec...()	Execute file, with different configurations of arguments, environment, etc.
fchdir()	Change to directory of given file descriptor.
fchmod()	Change permissions of file given by file descriptor
fchown()	Change owner/group id of the file given by file descriptor
fdatasync()	force write of file with file descriptor to disk.
fork()	Fork a child process.
forkpty()	Fork a new process with a new pseudo-terminal
fpathconf()	Return the configuration limit name for the file descriptor
fstat()	Return stat result for an open file descriptor.
fstatvfs()	Return stat result for open file descriptor on virtual file system
fsync()	force write of file with filedescriptor to disk.
ftruncate()	Truncate a file to a specified length.
getcwd()	Return unicode string representing current working directory.
getegid()	Return the current process's effective group id.
getenv()	Get specified environment variable or None/Default (returns string)
getenvb()	Get specified environment variable or None/Default (returns bytes)
geteuid()	Return the current process's effective user id.
getgid()	Return the current process's group id.

Method or Data	Description
getgroups()	Return list of supplemental group IDs for the process.
getloadavg()	Return number of processes averaged over 1, 5, and 15 minutes
getlogin()	Return the actual login name.
getpgid()	Call the system call getpgid().
getpgrp()	Return the current process group id.
getpid()	Return the current process id
getppid()	Return the parent's process id.
getresgid()	Return tuple of real, effective, saved group IDs
getresuid()	Return tuple of real, effective, saved user IDs
getsid()	Call the system call getsid().
getuid()	Return the current process's user id.
initgroups()	Initialize the group access list with all groups of which the specified username is a member, plus the specified group id.
isatty()	Return True if file descriptor is an open file descriptor
kill()	Kill a process with a signal.
killpg()	Kill a process group with a signal.
lchown()	Change owner/group of path (don't follow symlinks)
link()	Create a hard link to a file.
listdir()	Return list of all entries in the directory.
lseek()	Set the current position of a file descriptor.
lstat()	Like stat(path), but do not follow symbolic links.
major()	Extracts device major number from a raw device number.
makedev()	Composes a raw device number from major/minor device numbers.
makedirs()	Super-mkdir (like unix mkdir -p)
minor()	Extracts device minor number from a raw device number.

Method or Data	Description
mkdir()	Create a directory.
mkfifo()	Create a FIFO (a POSIX named pipe).
mknod()	Create a filesystem node
nice()	Decrease priority of process by inc and return new priority.
open()	Open a file (for low level IO).
openpty()	Open a pseudo-terminal
pathconf()	Return configuration limit name for file or directory path.
pipe()	Create a pipe.
putenv()	Change or add an environment variable.
read()	Read a file descriptor.
readlink()	Return string representation of symlink target
remove()	Remove a file (same as unlink(path)).
removedirs(name)	Super-rmdir; remove leaf directory and all empty intermediate ones
rename()	Rename a file or directory.
renames()	Super-rename; create directories as necessary
rmdir()	Remove a directory.
setegid()	Set the current process's effective group id.
seteuid()	Set the current process's effective user id.
setgid()	Set the current process's group id.
setgroups()	Set the groups of the current process to list.
setpgid()	Call the system call setpgid().
setpgrp()	Make this process a session leader.
setregid()	Set the current process's real and effective group ids.
setresgid()	Set the current process's real, effective, and saved group ids.
setresuid()	Set the current process's real, effective, and saved user ids.

Method or Data	Description
setreuid()	Set the current process's real and effective user ids.
setsid()	Call the system call setsid().
setuid()	Set the current process's user id.
spawn...()	Execute file with arguments from args in a subprocess.
stat()	Perform a stat system call on the given path.
stat_float_times()	Determine whether os.[lf]stat represents time stamps as float objects.
statvfs()	Perform a statvfs system call on the given path.
strerror()	Translate an error code to a message string.
symlink()	Create a symbolic link
sysconf()	Return an integer-valued system configuration variable.
system()	Execute the command (a string) in a subshell.
tcgetpgrp()	Return the process group associated with the terminal given by a fd.
tcsetpgrp()	Set the process group associated with the terminal given by a fd.
times()	Return tuple of floats indicating process times.
ttyname()	Return the name of the terminal device
umask()	Set the current numeric umask and return the previous umask.
uname()	Return a tuple identifying the current operating system.
unlink()	Remove a file (same as remove(path)).
unsetenv()	Delete an environment variable.
utime()	Set the access and modified time of file
wait...()	Wait for completion of a child process.
walk()	Directory tree generator.
write()	Write a string to a file descriptor.

Paths, directories and file names

- import os.path module
- Many routines for working with file and folder attributes

The os.path module provides many functions for working with file and directory names and paths. These are all about the file and directories *attributes*, not the contents.

Some of the more common methods are

os.path.abspath()
os.path.basename
os.path.dirname()
os.path.getmtime()
os.path.getsize()
os.path.isdir()
os.path.isfile()
os.path.join()
os.path.exists()

Example

paths.py

```
#!/usr/bin/env python
import sys
import os.path

unix_p1 = "bin/spam.txt"  ①
unix_p2 = "/usr/local/bin/ham"  ②

win_p1 = r"spam\ham.doc"  ③
win_p2 = r"\\spam\ham\eggs\toast\jam.doc"  ④

if sys.platform == 'win32':  ⑤
    print("win_p1:", win_p1)
    print("win_p2:", win_p2)
    print("dirname(win_p1):", os.path.dirname(win_p1))  ⑥
    print("dirname(win_p2):", os.path.dirname(win_p2))
    print("basename(win_p1):", os.path.basename(win_p1))  ⑦
    print("basename(win_p2):", os.path.basename(win_p2))
    print("isabs(win_p1):", os.path.isabs(win_p1))  ⑧
    print("isabs(win_p2):", os.path.isabs(win_p2))
else:
    print("unix_p1:", unix_p1)
    print("unix_p2:", unix_p2)
    print("dirname(unix_p1):", os.path.dirname(unix_p1))  ⑥
    print("dirname(unix_p2):", os.path.dirname(unix_p2))
    print("basename(unix_p1):", os.path.basename(unix_p1))  ⑦
    print("basename(unix_p2):", os.path.basename(unix_p2))
    print("isabs(unix_p1):", os.path.isabs(unix_p1))  ⑧
    print("isabs(unix_p2):", os.path.isabs(unix_p2))
    print(
        'format("cp spam.txt {}".format(os.path.expanduser("~"))):',  ⑨
        format("cp spam.txt {}".format(os.path.expanduser("~"))),
    )
    print(
        'format("cd {}".format(os.path.expanduser("~root"))):',  ⑩
        format("cd {}".format(os.path.expanduser("~root"))),
    )
```

① Unix relative path

② Unix absolute path

③ Windows relative path

④ Windows UNC path

⑤ What platform are we on?

⑥ Just the folder name

⑦ Just the file (or folder) name

⑧ Is it an absolute path?

⑨ ~ is current user's home

⑩ ~NAME is NAME's home

paths.py

```
unix_p1: bin/spam.txt
unix_p2: /usr/local/bin/ham
dirname(unix_p1): bin
dirname(unix_p2): /usr/local/bin
basename(unix_p1): spam.txt
basename(unix_p2): ham
isabs(unix_p1): False
isabs(unix_p2): True
format("cp spam.txt {}".format(os.path.expanduser("~"))): cp spam.txt /Users/jstrick
format("cd {}".format(os.path.expanduser("~root"))): cd /var/root
```

Table 3. os.path methods

Method	Description
abspath(path)	Return normalized absolutized version of the pathname path.
basename(path)	Return the base name of pathname path.
commonprefix(list)	Return the longest path prefix (taken character-by-character) that is a prefix of all paths in list. If list is empty, return the empty string ('').
dirname(path)	Return the directory name of pathname path.
exists(path)	Return True if path refers to an existing path. Returns False for broken symbolic links. May be subject to permissions
lexists(path)	Return True if path refers to an existing path. Returns True for broken symbolic links.
expanduser(path)	On Unix, return the argument with an initial component of "~" or "~user" replaced by that user's home directory. Only "~" is supported on Windows.
expandvars(path)	Return the argument with environment variables expanded. Substrings of the form "$name" or "${name}" are replaced by the value of environment variable name. Malformed variable names and references to non-existing variables are left unchanged.
getatime(path)	Return the time of last access of path. (seconds since epoch).
getmtime(path)	Return the time of last modification of path. (seconds since epoch).
getctime(path)	Return the system's ctime. (seconds since epoch).
getsize(path)	Return the size, in bytes, of path. Raise os.error if path does not exist or cannot be accessed.
isabs(path)	Return True if path is an absolute pathname (begins with a slash).
isfile(path)	Return True if path is an existing regular file. This follows symbolic links.
isdir(path)	Return True if path is an existing directory. Follows symbolic links.
islink(path)	Return True if path refers to a directory entry that is a symbolic link. Always False on Windows.

Method	Description
ismount(path)	Return True if pathname path is a mount point (Unix only).
join(path1[, path2[, ...]])	Join one or more path components intelligently.
normcase(path)	Normalize the case of a pathname. On Unix, this returns the path unchanged; on case-insensitive filesystems, it converts the path to lowercase. On Windows, it also converts forward slashes to backward slashes.
normpath(ph)	Normalize a pathname. This collapses redundant separators and up-level references so that A//B, A/./B and A/foo/../B all become A/B.
realpath(path)	Return the canonical path of the specified filename, eliminating any symbolic links encountered in the path.
samefile(path1, path2)	Return True if both pathname arguments refer to the same file or directory (as indicated by device number and i-node number). Raise an exception if a os.stat() call on either pathname fails. Availability: Macintosh, Unix.
sameopenfile(fp1, fp2)	Return True if the file descriptors fp1 and fp2 refer to the same file. Availability: Macintosh, Unix.
samestat(stat1, stat2)	Return True if the stat tuples stat1 and stat2 refer to the same file. These structures may have been returned by fstat(), lstat(), or stat(). Availability: Macintosh, Unix.
split(path)	Split the pathname path into a pair, (head, tail) where tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash.
splitdrive(path)	Split the pathname path into a pair (drive, tail) where drive is either a drive specification or the empty string. On systems which do not use drive specifications, drive will always be the empty string..
splitext(path)	Split the pathname path into a pair (root, ext) such that root + ext == path, and ext is empty or begins with a period and contains at most one period.

Method	Description
splitunc(path)	Split the pathname path into a pair (unc, rest) so that unc is the UNC mount point (such as r'\\host\mount'), if present, and rest the rest of the path (such as r'\path\file.ext'). For paths containing drive letters, unc will always be the empty string. Availability: Windows.
walk(path, visit, arg)	Calls the function visit with arguments (arg, dirname, names) for each directory in the directory tree rooted at path (including path itself, if it is a directory). Note: The newer os.walk() generator supplies similar functionality and can be easier to use. (Like File::Find in Perl)
supports_unicode_filenames()	True if arbitrary Unicode strings can be used as file names (within limitations imposed by the file system), and if os.listdir() returns Unicode strings for a Unicode argument. New in version 2.3.

Walking directory trees

- Use os.walk()
- Returns tuple for each directory
- Tuple contains directory path, subdirectories, and files

The os.walk method provides a way to easily walk a directory tree. It provides an iterator for a directory and all its subdirectories. For each directory, it returns a tuple with three values.

The first element is the full (absolute) path to the directory; the second element is a list of the directory's subdirectories (relative names); the third element is a list of the non-directory entries in the subdirectory (also relative names).

Be sure to use os.path.join() to put together the directory and the file or subdirectory name.

Do not use "dir" as a variable when looping through the iterator, because it will overwrite Python's builtin **dir** function.

Example

walk_ex.py

```
#!/usr/bin/env python
"""print size of every python file whose name starts with "m" """

import os

START_DIR = ".."  # start in root of student files  ①


def main():
    for currdir, subdirs, files in os.walk(START_DIR):  ②
        for file in files:  ③
            if file.endswith('.py') and file.startswith('m'):
                fullpath = os.path.join(currdir, file)  ④
                fsize = os.path.getsize(fullpath)
                print("{:8d} {:s}".format(fsize, fullpath))


if __name__ == '__main__':
    main()
```

① starting location

② walk folder tree

③ loop over file names

④ get file path

walk_ex.py

```
     828 ../custom/pynavy/1.0/f5_week2/EXAMPLES/moreindex.py
     175 ../custom/pynavy/1.0/f5_week2/EXAMPLES/mathop.py
     167 ../custom/pynavy/1.0/f5_week2/EXAMPLES/multi_ex.py
     469 ../custom/pynavy/1.0/f5_week2/EXAMPLES/modtest.py
    1176 ../acc_django_outlines/py3sci3day/EXAMPLES/mammal.py
     228 ../py3scicust.old/1.0/StudentFiles/unix/py3scicust/ANSWERS/media.py
    1139 ../py3scicust.old/1.0/StudentFiles/unix/py3scicust/EXAMPLES/mammal.py
     849 ../py3scicust.old/1.0/StudentFiles/unix/py3scicust/EXAMPLES/moreindex.py
     190
../py3scicust.old/1.0/StudentFiles/unix/py3scicust/EXAMPLES/math_operators.py
     411 ../py3scicust.old/1.0/StudentFiles/unix/py3scicust/EXAMPLES/modtest.py
```

...

Environment variables

- Shell or OS variables
- Same for Windows and non-Windows
- Syntax

```
value = os.environ[varname]
ivalue = os.environ.get(varname)
value = os.getenv(varname)
value = os.getenv(varname,default)
str2 = os.path.expandvars(str1)
```

There are several ways to access environment variables from Python.

The most direct is to use os.environ, which is a dictionary of the current environment. If a non-existent variable name is specified, a KeyError will be raised, so it is safer to use os.environ.get(varname[,default]) than os.environ[varname].

You can also use the os.getenv(varname[,default]) method. It takes the name of an environment variable and returns that variable's value. An optional second argument provides a default value if the variable is not defined.

Another way to use environment variables is to expand a string that contains them, using the expandvars(string) method of the os.path object. This takes a string containing one or more environment variables and returns the strings with environment variables expanded to their values.

If the variables do not exist in the environment, they are left unexpanded.

Example

getenv_ex.py

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

import sys
import os.path

if sys.platform == 'win32':
    user_key = 'USERNAME'
else:
    user_key = 'USER'

count_key = 'COUNT'

os.environ[count_key] = "42"  ①
print("count is", os.environ[count_key], "user is", os.environ[user_key])  ②
print("count is", os.environ.get(count_key), "user is", os.environ.get(user_key))
③
user = os.getenv(user_key)  ④
count = os.getenv(count_key)
print("count is", count, "user is", user)
cmd = "count is ${} user is ${}".format(count_key, user_key)
print("cmd:", cmd)
print(os.path.expandvars(cmd))  ⑤
```

① set environment variable

② os.environ is a dictionary

③ os.environ.get() is safer than os.environ[]

④ os.getenv() is shortcut for os.environ.get()

⑤ expand variables in place; handy for translating shell scripts

getenv_ex.py

```
count is 42 user is jstrick
count is 42 user is jstrick
count is 42 user is jstrick
cmd: count is $COUNT user is $USER
count is 42 user is jstrick
```

Launching external programs

- Different ways to launch programs
  - Just launch (use system())
  - Capture output (use popen())
- import os module
- Use system() or popen() methods

In Python, you can launch an external command using the os module functions **os.system()** and **os.popen()**.

os.system() launches any external command, as though you had typed it at a command prompt. popen() opens a command, returning a file-like object. You can read the output of the command with any of the methods used for a file.

You can open a process for writing as well, by specifying a mode of "w".

TIP For more sophisticated control of processes, see the **subprocess** module.

Example

external_programs.py

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

os.system("hostname")  ①

with os.popen('netstat -an') as netstat_in:  ②
    for entry in netstat_in:  ③
        if 'ESTAB' in entry:  ④
            print(entry, end='')
print()
```

① Just run "hostname"

② Open command line "netstat -an" as a file-like object

③ Iterate over lines in output of "netstat -an"

④ Check to see if line contains "ESTAB"

external_programs.py

```
MacBook-Pro-8.attlocal.net
tcp4       0      0  192.168.1.66.56791     18.214.24.118.443      ESTABLISHED
tcp4       0      0  192.168.1.66.56790     63.251.114.182.443     ESTABLISHED
tcp4       0      0  192.168.1.66.56776     74.121.138.88.443      ESTABLISHED
tcp4       0      0  192.168.1.66.56775     3.210.11.140.443       ESTABLISHED
tcp4       0      0  192.168.1.66.56773     35.174.92.20.443       ESTABLISHED
tcp4       0      0  192.168.1.66.56772     3.216.212.104.443      ESTABLISHED
tcp4       0      0  192.168.1.66.56771     52.4.252.13.443        ESTABLISHED
tcp4       0      0  192.168.1.66.56770     18.210.147.153.443     ESTABLISHED
tcp6       0      0  2600:1700:3901:6.56750 2607:f8b0:4002:8.443   ESTABLISHED
tcp6       0      0  2600:1700:3901:6.56729 2607:f8b0:4002:c.443   ESTABLISHED
tcp6       0      0  2600:1700:3901:6.56728 2607:f8b0:4002:c.443   ESTABLISHED
tcp4       0      0  192.168.1.66.56660     107.178.254.65.443     ESTABLISHED
tcp4       0      0  192.168.1.66.56657     23.221.46.225.443      ESTABLISHED
tcp4       0      0  192.168.1.66.56653     35.190.72.21.443       ESTABLISHED
```

...

Chapter 2 Exercises

Exercise 2-1 (path_files.py)

List each component of your PATH environment variable, together with the number of files it contains. This is the set of files you can execute from the command line without specifying a their path. Output should look something like this (for Windows, the paths will look different, but the idea is the same):

```
/usr/bin          2376
/usr/local/bin      17
/usr/local/sbin      1
/usr/sbin          263
```

TIP Use os to get the pathsep value; then use os.listdir to get the contents of each directory after splitting PATH.

Exercise 2-2 (oldest_file.py)

Write a script that, given a directory on the command line, prints out the oldest file in that directory. If there is more than one file sharing the oldest timestamp, print any one of them.

TIP Use os.path.getmtime()

Exercise 2-3 (all_python_lines.py)

Write a script that finds all the Python files (.py) in the student files (starting at py3interm), and counts the total number of lines in all of them.

7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com