Python Programming

Iverson

Trainer's Introduction

- Name: Asmaliza Ahzan @ Emma
- Email: asmaliza@iverson.com.my
- Senior Technical Consultant with Iverson Associates since 2012
- Been in training industry since 2008, graduated with MEng in Computer Systems Engineering from University of Queensland, Australia
- Domain would be programming languages, application development, data analytics/science, machine learning and artificial intelligence.

Now it's your turn...

- Name
- Job Title/Role/Designation
- Experiencing with data analytics/science tools, process, projects etc.
- Expectations from this course
- Misc.

Some Logistics

- Class hours: 9am – 5pm
- Monday – Friday
- 1 hour lunch break
- Morning and afternoon breaks

Agenda – Day 1

- About Python
- Getting Started
- The Quick Python Overview
- The absolute basics
- Lists, tuples and sets

Agenda – Day 2

- Strings
- Dictionaries
- Control Flow
- Functions
- Modules and Scoping Rules

Agenda – Day 3

- Python Programs
- Using the Filesystem
- Reading and Writing Files
- Exceptions

Agenda – Day 4

- Classes and Object-Oriented Programming
- Regular Expressions
- Data Types as Objects
- Packages
- Using Python Libraries

Agenda – Day 5

- Basic File Wrangling
- Processing Data Files
- Data Over the Network
- Saving Data
- Exploring Data

Training Objectives

- Master the fundamentals of writing Python scripts
- Learn core Python scripting elements such as variables and flow control structures
- Discover how to work with lists and sequence data
- Write Python functions to facilitate code reuse
- Use Python to read and write files
- Make their code robust by handling errors and exceptions properly
- Work with the Python standard library
- Explore Python's object-oriented features
- Search text using regular expressions

About Python

Chapter 1

This chapter cover

- Why use python?
- What python does well?
- What python does not do as well?
- Why learn python 3?

Why Use Python?

- Easy to learn and use
- Mature and supportive Python community
- Hundreds of Python libraries and frameworks
- Versatility, efficiency, reliability, and speed
- Big Data, Machine Language and Cloud Computing
- First-choice Language
- The flexibility of Python language
- Use of Python in academics
- Automation

Background

- Python is a modern programming language developed by Guido van Rossum in the 1990s (and named after a famous comedic troupe).
- Although Python isn't perfect for every application, its strengths make it a good choice for many situations.

What Python Does Well

- Python is easy to use
- Python is expressive
- Python is readable
- Python is complete – “batteries included”
- Python is cross-platform
- Python is free

What Python Doesn't Do Well

- Python isn't the fastest language
- Python doesn't have the most libraries
- Python doesn't check variable types at compile time
- Python doesn't have much mobile support
- Python doesn't use multiple processors well

Why Learn Python 3?

- Python has been around for a number of years and has evolved over that time.
- Python 3, originally whimsically dubbed Python 3000, is notable because it's the first version of Python in the history of the language to break backward compatibility.
- What this means is that code written for earlier versions of Python probably won't run on Python 3 without some changes.
- Why learn Python 3? Because it's the best Python so far.

Summary

- Python is a modern, high-level language with dynamic typing and simple, consistent syntax and semantics.
- Python is multiplatform, highly modular, and suited for both rapid development and large-scale programming.
- It's reasonably fast and can be easily extended with C or C++ modules for higher speeds.
- Python has built-in advanced features such as persistent object storage, advanced hash tables, expandable class syntax, and universal comparison functions.
- Python includes a wide range of libraries such as numeric processing, image manipulation, user interfaces, and web scripting.
- It's supported by a dynamic Python community.

Getting Started

Chapter 2

This chapter cover

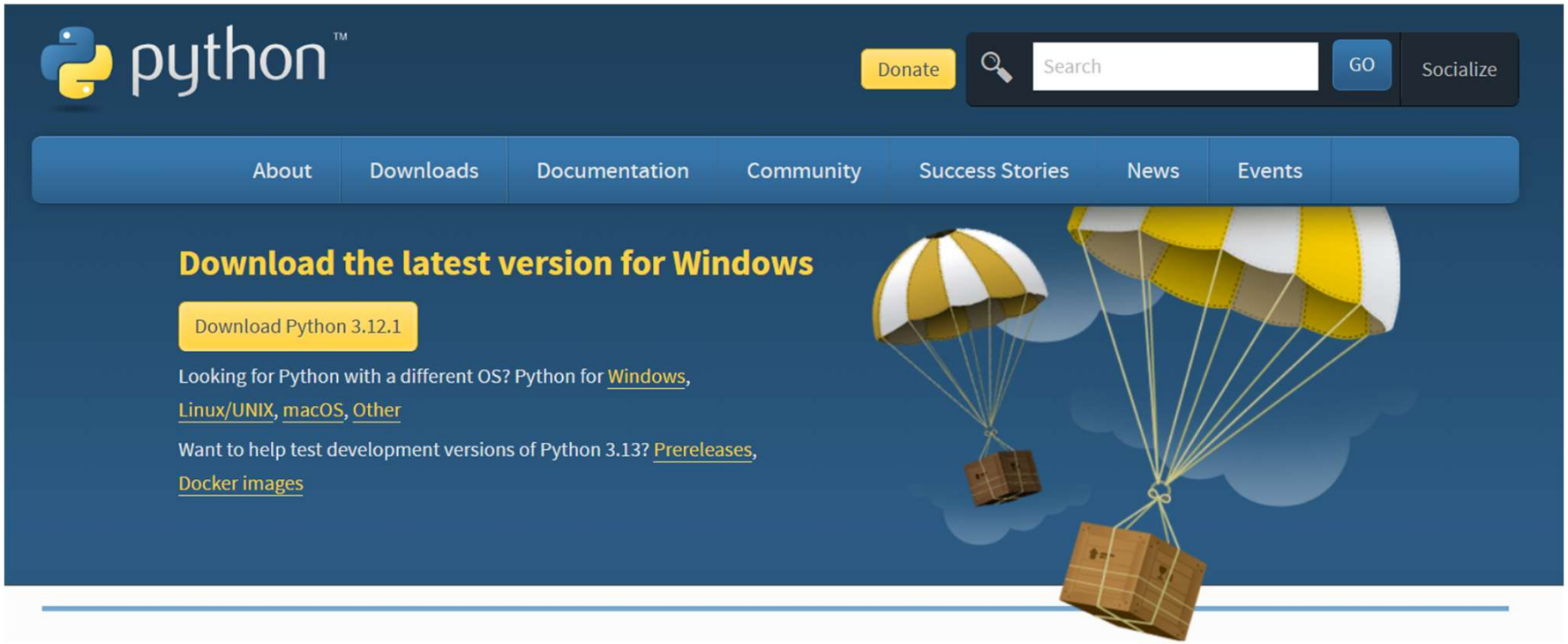
- Installing Python
- Using IDLE and the basic interactive mode
- Writing a simple program
- Using Visual Studio Code
- Using Python shell

Setup the Environment

Activity

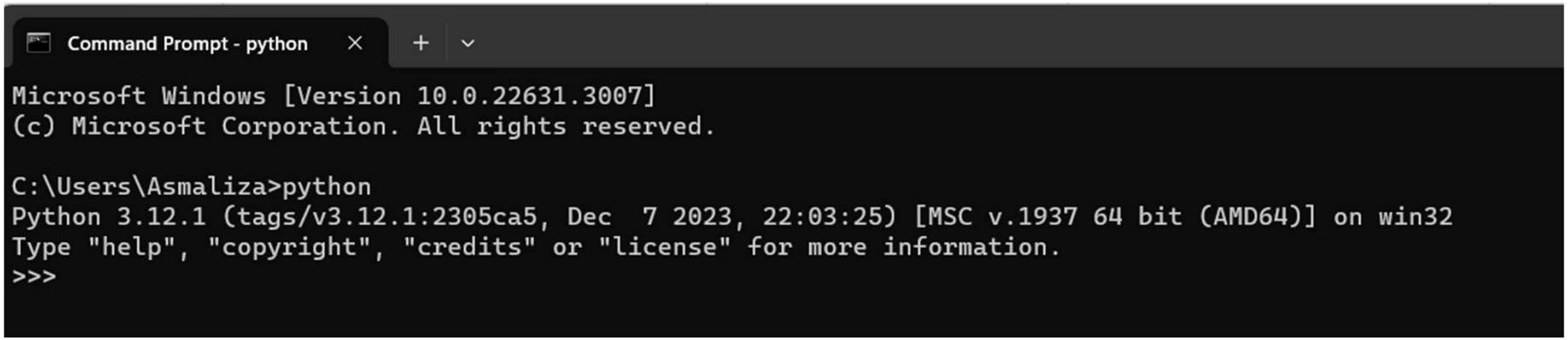
Installing Python

- Installer can be downloaded from https://www.python.org/downloads/



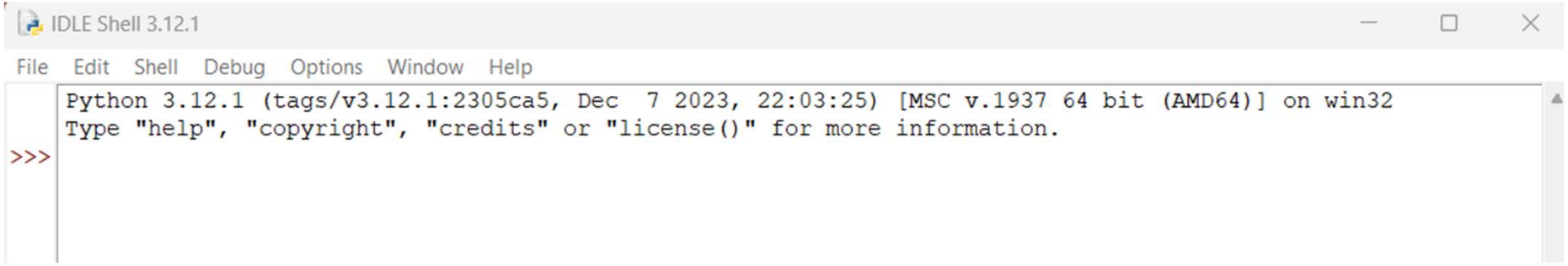
Basic Interactive Mode

- Launch from terminal/command prompt.



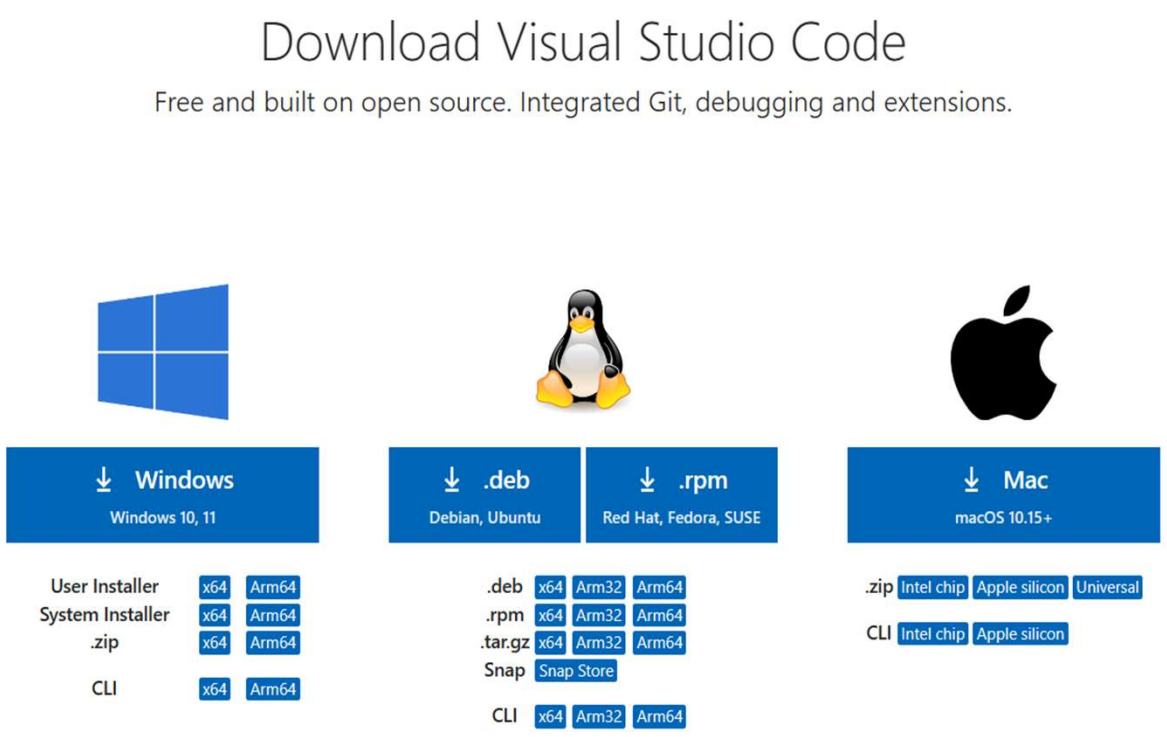
IDLE

- IDLE is the built-in development environment for Python.



Visual Studio Code

- Download the installer from https://code.visualstudio.com/
- Once installed, add the Python plugin.



Simple Program

- In VS Code, create new file hello.py
- Type below codes;

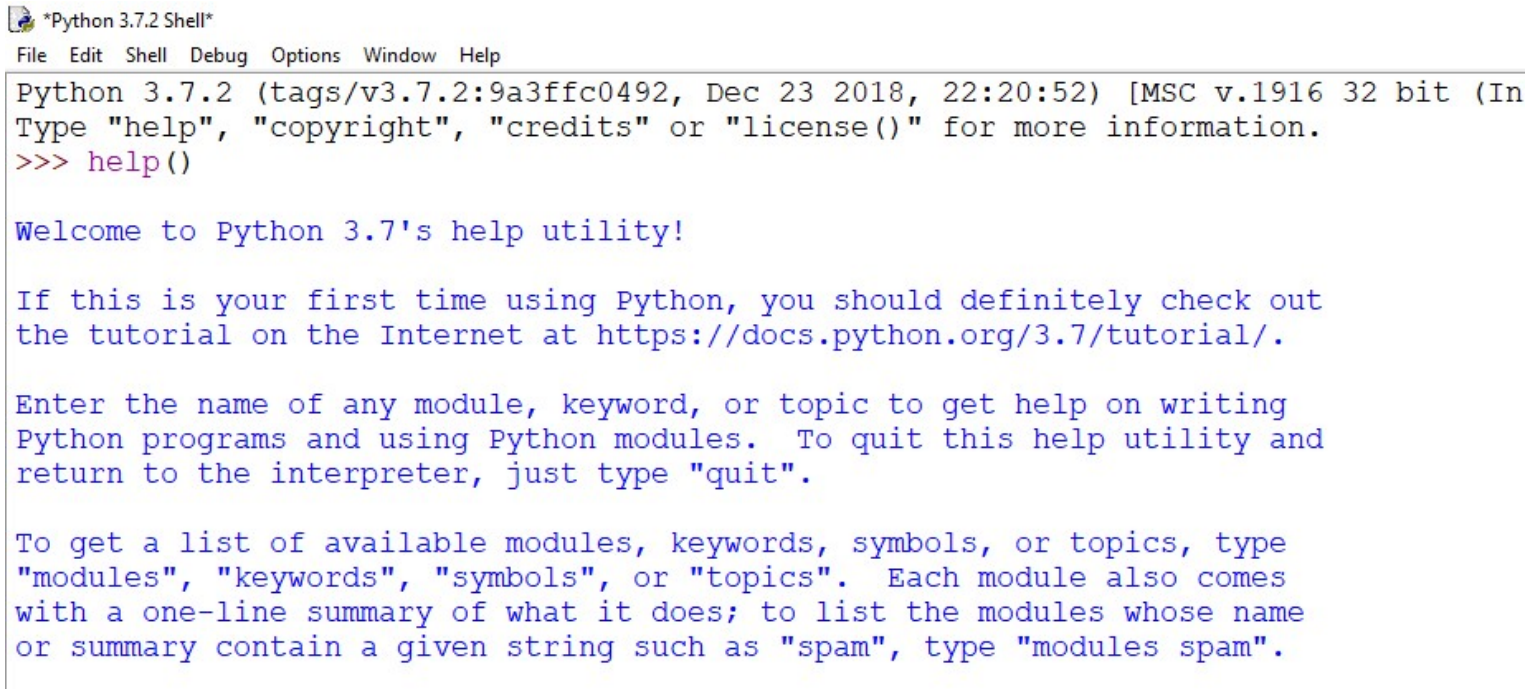
```
print("Hello World")
```

- To execute the codes, click on the Play button (right-top) 

PROBLEMS TERMINAL OUTPUT DEBUG CONSOLE

```
:/pythoncodes/basic/hello.py
Hello World
```

Using python shell



Python help

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

Python help - Functions

```
help> FUNCTIONS
Functions
*********

Function objects are created by function definitions.  The only
operation on a function object is to call it: "func(argument-lis

There are really two flavors of function objects: built-in funct
and user-defined functions.  Both support the same operation (to
the function), but the implementation is different, hence the
different object types.

See Function definitions for more information.

Related help topics: def, TYPES
```

Summary

- Installing Python 3 on Windows systems is as simple as downloading the latest installer from www.python.org and running it. Installation on Linux, UNIX, and Mac systems will vary.
- Refer to installation instructions on the Python website and use your system's software package installer where possible.
- Another installation option is to install the Anaconda (or miniconda) distribution from https://www.anaconda.com/download/.
- After you've installed Python, you can use either the basic interactive shell (and later, your favorite editor) or the IDLE integrated development environment.

The Quick Python Overview

Chapter 3

This chapter covers

- Surveying Python
- Using built-in data types
- Controlling program flow
- Creating modules
- Using Object-Oriented programming

Python Synopsis

- Python has several built-in data types, such as integers, floats, complex numbers, strings, lists, tuples, dictionaries, and file objects.
- These data types can be manipulated using language operators, built-in functions, library functions, or a data type's own methods.
- Programmers can also define their own classes and instantiate their own class instances.
- These class instances can be manipulated by programmer-defined methods, as well as the language operators and built-in functions for which the programmer has defined the appropriate special method attributes.

Python Synopsis

- Python provides conditional and iterative control flow through an if-elif-else construct along with while and for loops. It allows function definition with flexible argument-passing options.
- Exceptions (errors) can be raised by using the raise statement, and they can be caught and handled by using the try-except-else-finally construct.
- Variables (or identifiers) don't have to be declared and can refer to any built-in data type, user-defined object, function, or module.

Built-in Data Types

- Numbers
- Lists
- Tuples
- Strings
- Dictionaries
- Sets
- File Objects

Numbers

- Python's four number types are integers, floats, complex numbers, and Booleans:
- Integers—1, –3, 42, 355, 888888888888888, –7777777777 (integers aren't limited in size except by available memory)
- Floats—3.0, 31e12, –6e-4
- Complex numbers—3 + 2j, –4- 2j, 4.2 + 6.3j
- Booleans—True, False

```
 9  # define a variable age and assign value 17
10  age = 17
11
12  # print age to output
13  print(age)
```

Numbers

```
1   # define a variable age and assign value 17
2   age = 17
3
4   # print age to output
5   print(age)
6
7   # numbers
8   print(5 + 2 - 3 * 2)
9   print(5 / 2)    # floating 2.5
10  print(5 / 2.0)  # also floating 2.5
11  print(5 // 2)   # integer result 2
12  print(3000000000)   # a large number
13  print(3000000000 * 3)
14  print(3000000000 * 3.0)
15  print(2.0e-8)   # scientific
```

String

- String processing is one of Python's strengths.
- Strings can be delimited by single (' '), double (" "), triple single (''' '''), or triple double (""" """) quotations and can contain tab (\t) and newline (\n) characters.
- Strings are also immutable.

```
1  # strings
2  str1 = "A string in double quotes can contain 'single quote' characters."
3  str2 = 'A string in single quotes can contain "double quote" characters.'
4  str3 = '''\tA string which starts with a tab; ends with a newline character.\n'''
5  str4 = """This is a triple double quoted string, the only kind that can
6  contain real newlines."""
7
8  print(str1)
9  print(str2)
10 print(str3)
11 print(str4)
```

String

- Strings have several methods to work with their contents, and the re library module also contains functions for working with strings.

```
13  # define string s
14  x = "live and let \t \tlive"
15  print(x)
16
17  # split
18  y = x.split()
19  print(y)
20
21  # replace
22  z = x.replace(" let \t \tlive", "enjoy life")
23  print(z)
24
25  import re
26  regexpr = re.compile(r"[\t ]+")
27  w = regexpr.sub(" ", x)
28  print(w)
```

String

- The print function outputs strings. Other Python data types can be easily converted to strings and formatted.

```
30  # formatted output
31  e = 2.718
32  x = [1, "two", 3, 4.0, ["a", "b"], (5, 6)]
33  print("The constant e is:", e, "and the list x is:", x)
34
35  print("the value of %s is: %.2f" % ("e", e))
```

Lists

- A list can contain a mixture of other types as its elements, including strings, tuples, lists, dictionaries, functions, file objects, and any type of number.
- A list can be indexed from its front or back. You can also refer to a subsegment, or slice, of a list by using slice notation.

```
14  # lists and slicing
15  x = ["first", "second", "third", "fourth"]
16
17  # display all values in list
18  print(x)
19
20  # display the first value in list
21  print(x[0])
22
23  #slicing
24  print(x[1:-1])
25  print(x[0:3])
26  print(x[-2:-1])
27  print(x[:3])
28  print(x[-2:])
```

Tuples

- Tuples are similar to lists but are immutable - that is, they can't be modified after they've been created.
- A list can be converted to a tuple by using the built-in function tuple and vice-versa using the built-in function list.

```
1  # A list can be converted to a tuple by using the built-in function tuple
2  x = [1, 2, 3, 4]
3  print(x)
4  print(tuple(x))
5
6  # Conversely, a tuple can be converted to a list by using the built-in function list
7  x = (1, 2, 3, 4)
8  print(x)
9  print(list(x))
```

Dictionaries

- Python's built-in dictionary data type provides associative array functionality implemented by using hash tables.
- The built-in len() function returns the number of key-value pairs in a dictionary.
- The del statement can be used to delete a key-value pair.
- As is the case for lists, several dictionary methods (clear, copy, get, items, keys, update, and values) are available
- Keys must be of an immutable type B, including numbers, strings, and tuples.
- Values can be any kind of object, including mutable types such as lists and dictionaries.

Dictionaries

```
1  # dictionaries
2  x = {1: "one", 2: "two"}
3  print(x)
4
5  # add key-value pair
6  x["first"] = "one"
7  print(x)
8
9  # get all keys
10 keys = list(x.keys())
11 print(keys)
12
13 # get value
14 print(x[1])
15
16 # optional user-defined value if key-value pair not found
17 print(x.get(1, "not available"))
18 print(x.get(4, "not available"))
```

Sets

- A set in Python is an unordered collection of objects, used in situations where membership and uniqueness in the set are the main things you need to know about that object.
- Sets behave as collections of dictionary keys without any associated values.

```
1  # sets
2  x = set([1, 2, 3, 1, 3, 5])
3  print(x)
4
5  # in operator
6  print(1 in x)
7  print(4 in x)
```

File Objects

A file is accessed through a Python file object.

```
# open file for reading and writing
f = open("myfile", "w")

# write into file
f.write("First line with necessary newline character\n"
f.write("Second line to write to the file\n")
f.close()

# open file for reading only
f = open("myfile", "r")

line1 = f.readline()
line2 = f.readline()
f.close()

print(line1, line2)
```

Control Flow Structures

- Python has a full range of structures to control code execution and program flow, including common branching and looping structures.
- Python has several ways of expressing Boolean values; the Boolean constant False, 0, the Python nil value None, and empty values (for example, the empty list [] or empty string "") are all taken as False.
- The Boolean constant True and everything else is considered True.
- The comparison operators (<, <=, ==, >, >=, !=, is, is not, in, not in) and the logical operators (and, not, or), which all return True or False.

The if-elif-else Statement

- The block of code after the first True condition (of an if or an elif) is executed.
- If none of the conditions is True, the block of code after the else is executed.
- The elif and else clauses are optional B, and there can be any number of elif clauses.
- No explicit delimiters, such as brackets or braces, are necessary.
- All these statements must be at the same level of indentation.

```
1   x = 5
2
3   if x < 5:
4       y = -1
5       z = 5
6   elif x > 5:
7       y = 1
8       z = 11
9   else:
10      y = 0
11      z = 10
12
13  print(x, y, z)
```

The while Loop

- The while loop is executed as long as the condition (which here is x > y) is True.

```
1   u, v, x, y = 0, 0, 100, 30
2
3   while x > y:
4       u = u + y
5       x = x - y
6       if x < y + 2:
7           v = v + x
8           x = 0
9       else:
10          v = v + y + 2
11          x = x - y - 2
12  print(u, v)
```

The for Loop

- The for loop is simple but powerful because it's possible to iterate over any iterable type, such as a list or tuple.
- Unlike in many languages, Python's for loop iterates over each of the items in a sequence (for example, a list or tuple), making it more of a foreach loop.

```
1  item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
2
3  for x in item_list:
4      if not isinstance(x, int):
5          continue
6      if not x % 7:
7          print("found an integer divisible by seven: %d" % x)
8          break
```

Function Definition

- Functions are defined by using the def statement.
- The return statement is what a function uses to return a value.
- This value can be of any type. If no return statement is encountered, Python's None value is returned.
- Function arguments can be entered either by position or by name (keyword).s

```
1  # syntax
2  # def name(param1, param2,...):
3  #   body
4
5  # function that will return the factorial
6  def fact(n):
7      """ Return the factorial of the given number """
8      r = 1
9      while n > 0:
10         r = r * n
11         n = n -1
12     return r
13
14 print(fact(4))
```

Exceptions

- Exceptions (errors) can be caught and handled by using the try-except-else-finally compound statement.
- This statement can also catch and handle exceptions you define and raise yourself.
- Any exception that isn't caught causes the program to exit.

```
1  class EmptyFileError(Exception):
2      pass
3  filenames = ["myfile1", "nonExistent", "emptyFile", "myfile2"]
4  for file in filenames:
5      try:
6          f = open(file, 'r')
7          line = f.readline()
8          if line == "":
9              f.close()
10             raise EmptyFileError("%s: is empty" % file)
11     except IOError as error:
12         print("%s: could not be opened: %s" % (file, error.strerror))
13     except EmptyFileError as error:
14         print(error)
15     else:
16         print("%s: %s" % (file, f.readline()))
17     finally:
18         print("Done processing", file)
```

Context handling using the with keyword

- A more streamlined way of encapsulating the try-except-finally pattern is to use the with keyword and a context manager.
- One benefit of context managers is that they may (and usually do) have default cleanup actions defined, which always execute whether an exception occurs.

```
1  filename = "myfile.txt"
2  with open(filename, "r") as f:
3      for line in f:
4          print(f)
```

Module Creation

- It's easy to create your own modules, which can be imported and used in the same way as Python's built-in library modules.
- The example in this listing is a simple module with one function that prompts the user to enter a filename and determines the number of times that words occur in this file.

```
"""wo module. Contains function: words_occur()"""
# interface functions
def words_occur():
    """words_occur() - count the occurrences of words in a file."""
    # Prompt user for the name of the file to use.
    file_name = input("Enter the name of the file: ")
    # Open the file, read it and store its words in a list.
    f = open(file_name, 'r')
    word_list = f.read().split()
    f.close()
    # Count the number of occurrences of each word in the file.
    occurs_dict = {}
    for word in word_list:
        # increment the occurrences count for this word
        occurs_dict[word] = occurs_dict.get(word, 0) + 1
    # Print out the results.
    print("File %s has %d words (%d are unique)" \
    % (file_name, len(word_list), len(occurs_dict)))
    print(occurs_dict)

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

Object-Oriented Programming

- Python provides full support for OOP.
- Listing is an example that might be the start of a simple shapes module for a drawing program.
- Classes are defined by using the class keyword.
- The instance initializer method (constructor) for a class is always called __init__
- Methods, like functions, are defined by using the def keyword.

```
"""sh module. Contains classes Shape, Square and Circle"""
class Shape:
    """Shape class: has method move"""
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, deltaX, deltaY):
        self.x = self.x + deltaX
        self.y = self.y + deltaY
class Square(Shape):
    """Square Class:inherits from Shape"""
    def __init__(self, side=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.side = side
class Circle(Shape):
    """Circle Class: inherits from Shape and has method are
    pi = 3.14159
    def __init__(self, r=1, x=0, y=0):
        Shape.__init__(self, x, y)
        self.radius = r
    def area(self):
        """Circle area method: returns the area of the circ
        return self.radius * self.radius * self.pi
    def __str__(self):
        return "Circle of radius %s at coordinates (%d, %d)
                % (self.radius, self.x, self.y)
```

Summary

- This chapter is a rapid and very high-level overview of Python; the following chapters provide more detail. This chapter ends the book's overview of Python.
- You may find it valuable to return to this chapter and work through the appropriate examples as a review after you read about the features covered in subsequent chapters.
- If this chapter was mostly a review for you, or if you'd like to learn more about only a few features, feel free to jump around, using the index or table of contents.
- You should have a solid understanding of the Python features in this chapter before skipping ahead to part 4.

The Absolute Basics

Chapter 4

This chapter covers

- Indenting and block structuring
- Differentiating comments
- Assigning variables
- Evaluating expressions
- Using common data types
- Getting user input
- Using correct Pythonic style

Indentation and block structuring

- Python differs from most other programming languages because it uses whitespace and indentation to determine block structure.

```
1  # This is Python code. (Yea!)
2  n = 9
3  r = 1
4  while n > 0:
5      r = r * n
6      n = n - 1
```

Advantages of indentation

- It's impossible to have missing or extra braces. You never need to hunt through your code for the brace near the bottom that matches the one a few lines from the top.
- The visual structure of the code reflects its real structure, which makes it easy to grasp the skeleton of code just by looking at it.
- Python coding styles are mostly uniform. In other words, you're unlikely to go crazy from dealing with someone's idea of aesthetically pleasing code. Everyone's code will look pretty much like yours.

Differentiating comments

- For the most part, anything following a # symbol in a Python file is a comment and is disregarded by the language.
- The obvious exception is a # in a string, which is just a character of that string.

```
 9   # Assign 5 to x
10   x = 5
11   x = 3 # Now x is 3
12   x = "# This is not a comment"
```

Variables and assignments

- In Python, unlike in many other computer languages, neither a variable type declaration nor an end-of-line delimiter is necessary.
- The line is ended by the end of the line.
- Variables are created automatically when they're first assigned.
- Python variables can be set to any object, whereas in C and many other languages, variables can store only the type of value they're declared as.
- The following is perfectly legal Python code

```
14   x = "Hello"
15   print(x)
16
17   x = 5
18   print(x)
```

Expressions

- Arithmetic and similar expressions.
- Standard rules of arithmetic precedence apply. If you'd left out the parentheses in the last line, the code would've been calculated as x + (y / 2).

```
1   x = 3
2   y = 5
3   z = (x + y) / 2
```

Try This

- In your editor, create some variables.
- What happens when you try to put spaces, dashes, or other nonalphanumeric characters in the variable name?
- Play around with a few complex expressions, such as x = 2 + 4 * 5 – 6 / 3. Use parentheses to group the numbers in different ways and see how the result changes compared with the original ungrouped expression.

Strings

- Python, like most other programming languages, indicates strings using double quotes.

```
39  x = "Hello, World"
40
41  x = "\tThis string starts with a \"tab\"."
42  x = "This string contains a single backslash(\\)."
43
44  x = "Hello, World"
45  x = 'Hello, World'
46
47  x = "Don't need a backslash"
48  x = 'Can\'t get by without a backslash'
49  x = "Backslash your \" character!"
```

Numbers

- Python offers four kinds of numbers: integers, floats, complex numbers, and Booleans.

```
4   # print age to output
5   print(age)
6
7   # numbers
8   print(5 + 2 - 3 * 2)
9   print(5 / 2)    # floating 2.5
10  print(5 / 2.0)  # also floating 2.5
11  print(5 // 2)   # integer result 2
12  print(3000000000)   # a large number
```

Built-in numeric functions

- abs
- div
- mod
- float
- hex
- int
- max
- min
- oct
- pow
- round

Advanced numeric functions

- from math import *

acos	asin	atan	ceil	cos
cosh	e	exp	fabs	floor
fmod	frexp	hypot	ldexp	log
log10	mod	pi	pow	sin
sinh	sqrt	tan	tanh	

Try This

- In your editor, create some string and number variables (integers, floats, and complex numbers).
- Experiment a bit with what happens when you do operations with them, including across types.
- Can you multiply a string by an integer, for example, or can you multiply it by a float or complex number?
- Also load the math module and try a few of the functions; then load the cmath module and do the same. What happens if you try to use one of those functions on an integer or float after loading the cmath module? How might you get the math module functions back?

The None Value

- Python has a special basic data type that defines a single special data object called None.
- As the name suggests, None is used to represent an empty value.
- None is often useful in day-to-day Python programming as a placeholder to indicate a point in a data structure where meaningful data will eventually be found, even though that data hasn't yet been calculated.

Getting input from the user

- Use the input() function to get input from the user.

```
1   # getting input from user
2   name = input("Name? ")
3   print(name)
4
5   age = int(input("Age? "))
6   print(age)
```

Try This

- Experiment with the input() function to get string and integer input. Using code similar to the previous code, what is the effect of not using int() around the call to input()for integer input?
- Can you modify that code to accept a float - say, 28.5?
- What happens if you deliberately enter the wrong type of value? Examples include a float in which an integer is expected and a string in which a number is expected - and vice versa.

Built-in operators

- Python provides various built-in operators, from the standard (+, *, and so on) to the more esoteric, such as operators for performing bit shifting, bitwise logical functions, and so forth.
- Most of these operators are no more unique to Python than to any other language.

Basic Python style

Situation	Suggestion	Example
Module/package names	Short, all lowercase, underscores only if needed	imp, sys
Function names	All lowercase, underscores_for_readability	Function names
Variable names	All lowercase, underscores_for_readability	my_var
Indentation	Four spaces per level, no tabs	
Comparison	Don't compare explicitly to True or False	if my_var: if not my_var:

Quick Check

- Which of the following variable and function names do you think are not good Pythonic style? Why?

bar()	varName	VERYLONGVARNAME	foobar
longvarname	foo_bar()	really_very_long_var_name	

Summary

- The basic syntax summarized above is enough to start writing Python code.
- Python syntax is predictable and consistent.
- Because the syntax offers few surprises, many programmers can get started writing code surprisingly quickly.

Lists, tuples, and sets

Chapter 5

This chapter covers

- Manipulating lists and list indices
- Modifying lists
- Sorting
- Using common list operations
- Handling nested lists and deep copies
- Using tuples
- Creating and using sets

Lists are like arrays

- A list in Python is an ordered collection of objects.
- You create a list by enclosing a comma-separated list of elements in square brackets.
- Python lists can contain different types of elements; a list element can be any Python object.
- Probably the most basic built-in list function is the len() function, which returns the number of elements in a list

```
1  # This assigns a three-element list to x
2  x = [1, 2, 3]
3
4  # First element is a number, second is a string,
5  # third is another list.
6  x = [2, "two", [1, 2, 3]]
7
8  # get the length/size of list
9  print(len(x))
```

List indices

- Python list index start with 0.
- If indices are negative numbers, they indicate positions counting from the end of the list, with –1 being the last position in the list, –2 being the second-to-last position, and so forth.

```
11  x = ["first", "second", "third", "fourth"]
12  print(x[0])
13  print(x[2])
14
15  a = x[-1]
16  print(a)
17  print(x[-2])
```

Slicing

- Creating a subset from the list.

```
19  x = ["first", "second", "third", "fourth"]
20  y = x[1:-1]
21  print(y)
22
23  y = x[0:3]
24  print(y)
25
26  y = x[-2:-1]
27  print(y)
28
29  print(x[:3])
30  print(x[2:])
```

Try This

- Using what you know about the len() function and list slices, how would you combine the two to get the second half of a list when you don't know what size it is?
- Experiment in the Python shell to confirm that your solution works.

Modifying Lists

- Add
- Append
- Remove
- Extend
- Insert
- Delete
- Reverse

```
36
37  # append
38  x[len(x):] = [5, 6, 7]
39  print(x)
40
41  x[:0] = [-1, 0] # append from front
42  x[1:-1] = []    # remove/clear
43
44  x = [1, 2, 3, 4]
45  y = [5, 6, 7]
46  x.append(y)     # [1, 2, 3, 4, [5, 6, 7]]
47  x.extend(y)     # [1, 2, 3, 4, 5, 6, 7]
48  x.insert(2, "hello")
49
50  del x[1]
```

Try This

- Suppose that you have a list 10 items long.
- How might you move the last three items from the end of the list to the beginning, keeping them in the same order?
- Example:

  nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

- Expected result:

  result = [8, 9, 10, 1, 2, 3, 4, 5, 6, 7]

Sorting Lists

- Lists can be sorted by using the built-in Python sort method.
- To sort a list without changing the original list, you have two options.
  - use the sorted() built-in function,
  - make a copy of the list and sort the copy
- According to the built-in Python rules for comparing complex objects, the sublists are sorted first by ascending first element and then by ascending second element.

```python
# in-place sorting (change the list)
x = [3, 8, 4, 0, 2, 1]
x.sort()

# make copy then sort
x = [2, 4, 1, 3]
y = x[:]
y.sort()

# strings
x = ["Life", "Is", "Enchanting"]
x.sort()

# list of list
x = [[3, 5], [2, 9], [2, 3], [4, 1], [3, 2]
x.sort()
```

The sorted( ) function

- Python also has the built-in function sorted(), which returns a sorted list from any iterable.
- sorted() uses the same key and reverse parameters as the sort method.

```
18  # sorted
19  x = (4, 3, 1, 2)
20  y = sorted(x)
21
22  z = sorted(x, reverse=True)
```

Try This

- Suppose that you have a list in which each element is in turn a list: [[1, 2, 3], [2, 1, 3], [4, 0, 1]].
- If you wanted to sort this list by the second element in each list so that the result would be [[4,0, 1], [2, 1, 3], [1, 2, 3]], what function would you write to pass as the key value to the sort() method?

Other common list operations

- List membership with the in operator
- List concatenation with the + operator
- List initialization with the * operator
- List minimum or maximum with min and max
- List search with index
- List matches with count

```python
# in operator
print(3 in [1, 3, 4, 5])
print(3 in ["one", "two", "three"])
print(3 not in ["one", "two", "three"])

# + operator
print(z = [1, 2, 3] + [4, 5])

# * operator
z = [None] * 4
z = [3, 1] * 2

# min and max
print(min([3, 7, 0, -2, 11]))
print(max([3, 7, 0, -2, 11]))

# index
x = [1, 3, "five", 7, -2]
print(x.index(7))

# count
x = [1, 2, 2, 3, 5, 2, 5]
print(x.count(2))
```

Try This

- What would be the result of len([[1,2]] * 3)?
- What are two differences between using the in operator and a list's index() method?
- Which of the following will raise an exception?:

  min(“a”, “b”, “c”]);
  max([1, 2, “three”]); [1, 2, 3].count(“one”)

Try This

- If you have a list x, write the code to safely remove an item if - and only if - that value is in the list.
- Modify that code to remove the element only if the item occurs in the list more than once.

Nested lists and deep copies

- Lists can be nested.
- One application of nesting is to represent two-dimensional matrices.
- The members of these matrices can be referred to by using two-dimensional indices.

```
1   m = [[0, 1, 2], [10, 11, 12], [20, 21, 22]]
2
3   print(m[0])
4   print(m[0][1])
5   print(m[2])
6   print(m[2][2])
```

Tuples

- Tuples are data structures that are very similar to lists, but they can't be modified; they can only be created.
- Tuples are so much like lists that you may wonder why Python bothers to include them.
- The reason is that tuples have important roles that can't be efficiently filled by lists, such as keys for dictionaries.

Tuples Basics

- Creating a tuple is similar to creating a list: assign a sequence of values to a variable.
- A list is a sequence that's enclosed by [and]; a tuple is a sequence that's enclosed by (and)

```
1   x = ('a', 'b', 'c'
2
3   print(x[2])
4   print(x[1:])
5   print(max(x))
6   print(min(x))
7   print(5 in x)
8   print(5 not in x)
9
10  x[2] = 'd'
11
12  print(x + x)
13  print(2 * x)
```

Sets

- A set in Python is an unordered collection of objects used when membership and uniqueness in the set are main things you need to know about that object.
- Like dictionary keys (discussed in chapter 7), the items in a set must be immutable and hashable.
- This means that ints, floats, strings, and tuples can be members of a set, but lists, dictionaries, and sets themselves can't.

Sets

- Create
- Add
- Remove
- In operator
- Convert list to set
- Logical operations

```
1   x = set([1, 2, 3, 1, 3, 5])
2
3   x.add(6)
4
5   x.remove(5)
6
7   print(1 in x)
8   print(4 in x)
9
10  y = set([1, 7, 8, 9])
11
12  print(x | y)
13  print(x & y)
14  print(x ^ y)
```

Try This

- If you were to construct a set from the following list, how many elements would the set have?: [1, 2, 5, 1, 0, 2, 3, 1, 1, (1, 2, 3)] ?

Lab

Examining a List

Summary

- Lists and tuples are structures that embody the idea of a sequence of elements, as are strings.
- Lists are like arrays in other languages, but with automatic resizing, slice notation, and many convenience functions.
- Tuples are like lists but can't be modified, so they use less memory and can be dictionary keys (see chapter 7).
- Sets are iterable collections, but they're unordered and can't have duplicate elements.

Strings

Chapter 6

This chapter covers

- Understanding strings as sequences of characters
- Using basic string operations
- Inserting special characters and escape sequences
- Converting from objects to strings
- Formatting strings
- Using the byte type

Strings as sequences of characters

- For the purposes of extracting characters and substrings, strings can be considered to be sequences of characters, which means that you can use index or slice notation.
- Python strings can't be modified.

```
53   x = "Hello, World"
54
55   print(x[0])
56   print(x[-1])
57   print(x[1:])
58
59   x = "Goodbye\n"
60   x = x[:-1]
61   print(x)
62   print(len("Goodbye"))
```

Basic string operations

- The simplest (and probably most common) way to combine Python strings is to use the string concatenation operator +

```
64   x = "Hello " + "World"
65   print(x)
66
67   print(8 * "x")
```

Special characters and escape sequences

Escape Sequence	Character represented
\'	Single-quote character
\"	Double-quote character
\\	Backlash character
\a	Bell character
\b	Backspace character
\f	Formfeed character
\n	Newline character
\r	Carriage-return character
\t	Tab character
\v	Vertical tab character

String methods

- join()
- split()

```
1   # join
2   print(" ".join(["join", "puts", "spaces", "between", "elements"]))
3   print("::".join(["Separated", "with", "colons"]))
4   print("".join(["Separated", "by", "nothing"]))
5
6   # split
7   x = "You\t\t can have tabs\t\n \t and newlines \n\n mixed in"
8   print(x.split())
9
10  x = "Mississippi"
11  print(x.split("ss"))
```

Quick Check

- How could you use split and join to change all the whitespace in string x to dashes, such as changing “this is a test” to “this-is-a-test”?

Converting strings to numbers

- Use the functions int and float to convert strings to integer or floating-point numbers, respectively.
- If they're passed a string that can't be interpreted as a number of the given type, these functions raise a ValueError exception.

```
1   x = float('123.456')
2
3   y = float('xxyy')   # ValueError
4
5   # octal
6   print(int('10000', 8))
7
8   # binary
9   print(int('101', 2))
10
11  # hex
12  print(int('ff', 16))
```

Quick Check

- Which of the following will not be converted to numbers, and why?

  int(‘a1’)

  int('12G', 16)

  float("123456789012345678 90")

  int("12*2")

Getting rid of extra whitespace

- strip()
- lstrip()
- rstrip()

```
1   x = " Hello, World\t\t "
2
3   print(x.strip())
4   print(x.lstrip())
5   print(x.rstrip())
6
7   x = "www.python.org"
8   print(x.strip("w"))
```

Quick Check

- If the string x equals "(name, date),\n", which of the following would return a string containing "name, date"?

  x.rstrip("),")

  x.strip("),\n")

  x.strip("\n)(,")

String searching

- find()
- rfind()
- index()
- rindex()
- count()
- startswith()
- endswith()

```
1   x = "Mississippi"
2
3   print(x.find("ss"))
4   print(x.find("zz"))
5   print(x.find("ss", 3))
6   print(x.find("ss", 0, 3))
7   print(x.rfind("ss"))
8   print(x.count("ss"))
9   print(x.startswith("Miss"))
10  print(x.endswith("pi"))
```

Modifying strings

- Strings are immutable, but string objects have several methods that can operate on that string and return a new string that's a modified version of the original string.
- This provides much the same effect as direct modification for most purposes.

```
1   x = "Mississippi"
2   print(x.replace("ss", "+++"))
```

Modifying strings with list manipulations

- Strings are immutable objects.
- Turn the string into a list of characters, do whatever you want, and then turn the resulting list back into a string.

```
x = "Mississippi"
print(x.replace("ss", "+++"))

text = "Hello, World"
wordList = list(text)
print(wordList)

wordList[6:] = []
print(wordList)

wordList.reverse()
print(wordList)

text = "".join(wordList)
print(text)
```

Quick Check

- What would be a quick way to change all punctuation in a string to spaces?

Converting from objects to strings

- In Python, almost anything can be converted to some sort of a string representation by using the built-in repr function.

```
1  print(repr([1, 2, 3]))
2
3  x = [1]
4  x.append(2)
5
6  print(x)
7  x.append([3, 4])
8  print(x)
```

Using the format method

```
1   print("{0} is the {1} of {2}".format("Ambrosia", "food", "the gods"))
2
3   print("{{Ambrosia}} is the {0} of {1}".format("food", "the gods"))
4
5   print("{food} is the food of {user}".format(food="Ambrosia", user="the gods"))
6
7   print("{0} is the food of {user[1]}".format("Ambrosia", user=["men", "the gods", "others"]))
8
9   print("{0:10} is the food of gods".format("Ambrosia"))
10  print("{0:{1}} is the food of gods".format("Ambrosia", 10))
11  print("{food:{width}} is the food of gods".format(food="Ambrosia", width=10))
12  print("{0:>10} is the food of gods".format("Ambrosia"))
13  print("{0:&>10} is the food of gods".format("Ambrosia"))
```

Formatting strings with %

```
15  print("%s is the %s of %s" % ("Ambrosia", "food", "the gods"))
16
17  print("%s is the %s of %s" % ("Nectar", "drink", "gods"))
18
19  print("%s is the %s of the %s" % ("Brussels Sprouts", "food", "foolish"))
20
21  x = [1, 2, "three"]
22  print("The %s contains: %s" % ("list", x))
```

Using formatting sequences

- All formatting sequences are substrings contained in the string on the left side of the central %.
- Each formatting sequence begins with a percent sign and is followed by one or more characters that specify what is to be substituted for the formatting sequence and how the substitution is to be accomplished.

```
25  # sequences
26  print("Pi is <%-6.2f>" % 3.14159) # use of the formatting sequence: %-6.2f
```

Named parameters and formatting sequences

```
28  # named parameters
29  num_dict = {'e': 2.718, 'pi': 3.14159}
30  print("%(pi).2f - %(pi).4f - %(e).2f" % num_dict)
```

Quick Check

- What would be in the variable x after the following snippets of code have executed?

```
x = "%.2f" % 1.1111
x = "%(a).2f" %{'a':1.1111}
x = "%(a).08f" %{'a':1.1111}
```

String interpolation

- Starting in Python 3.6, there's a way to create string constants containing arbitrary values, which is called string interpolation.
- String interpolation is a way to include the values of Python expressions inside literal strings.
- These f-strings, as they're commonly called because they are prefixed with f, use a syntax similar to that of the format method, but with a little less overhead.

```
32  # string interpolation
33  value = 42
34  message = f"The answer is {value}"
35  print(message)
```

Lab

Preprocessing Text

Summary

- Python strings have powerful text-processing features, including searching and replacing, trimming characters, and changing case.
- Strings are immutable; they can't be changed in place.
- Operations that appear to change strings actually return a copy with the changes.
- The re (regular expression) module has even more powerful string capabilities, which are discussed in chapter 16.

Dictionaries

Chapter 7

This chapter covers

- Defining a dictionary
- Using dictionary operations
- Determining what can be used as a key
- Creating sparse matrices
- Using dictionaries as caches
- Trusting the efficiency of dictionaries

What is a dictionary?

- Dictionaries access values by means of integers, strings, or other Python objects called keys, which indicate where in the dictionary a given value is found.
- Both lists and dictionaries can store objects of any type.
- Values stored in a dictionary are not implicitly ordered relative to one another because dictionary keys aren't just numbers.

```
1   # dictionaries
2   x = {1: "one", 2: "two"}
3   print(x)
```

Dictionaries

- A dictionary is a way of mapping from one set of arbitrary objects to an associated but equally arbitrary set of objects.

```
20  english_to_french = {}
21  english_to_french['red'] = 'rouge'
22  english_to_french['blue'] = 'bleu'
23  english_to_french['green'] = 'vert'
24  print("red is", english_to_french['red'])
```

Other dictionary operations

Dictionary Operation	Explanation	Example
{}	Creates an empty dictionary	x = {}
len	Returns the number of entries in a dictionary	len(x)
keys	Returns a view of all keys in a dictionary	x.keys()
values	Returns a view of all values in a dictionary	x.values()
items	Returns a view of all items in a dictionary	x.items()
del	Removes an entry from a dictionary	del(x[key])
in	Tests whether a key exists in a dictionary	'y' in x
get	Returns the value of a key or a configurable default	x.get('y', None)
copy	Makes a shallow copy of a dictionary	y = x.copy()

Quick Check

- Assume that you have a dictionary x = {'a':1, 'b':2, 'c':3, 'd':4} and a dictionary y = {'a':6, 'e':5, 'f':6}. What would be the contents of x after the following snippets of code have executed?:

```
del x['d']
z = x.setdefault('g', 7)
x.update(y)
```

Word counting

```
26  # word counting
27  sample_string = "To be or not to be"
28  occurrences = {}
29  for word in sample_string.split():
30      occurrences[word] = occurrences.get(word, 0) + 1
31
32  for word in occurrences:
33      print("The word", word, "occurs", occurrences[word], "times in the string")
```

What can be used as a key?

Python type	Immutable?	Hashtable?	Dictionary key?
int	Yes	Yes	Yes
float	Yes	Yes	Yes
boolean	Yes	Yes	Yes
complex	Yes	Yes	Yes
str	Yes	Yes	Yes
bytes	Yes	Yes	Yes
bytearray	No	No	No
list	No	No	No
tuple	Yes	Sometimes	Sometimes
set	No	No	no

Sparse matrices

- In mathematical terms, a matrix is a two-dimensional grid of numbers, usually written in textbooks as a grid with square brackets on each side.
- A fairly standard way to represent such a matrix is by means of a list of lists.
- To implement sparse matrices by using dictionaries with tuple indices.

```
36  matrix = [[3, 0, -2, 11], [0, 9, 0, 0], [0, 7, 0, 0], [0, 0, 0, -5]]
37
38  matrix = {(0, 0): 3, (0, 2): -2, (0, 3): 11, (1, 1): 9, (2, 1): 7, (3, 3): -5}
```

Efficiency of dictionaries

- The truth is that the Python dictionary implementation is quite fast.
- Many of the internal language features rely on dictionaries, and a lot of work has gone into making them efficient.
- Because all of Python's data structures are heavily optimized, you shouldn't spend much time worrying about which is faster or more efficient.
- If the problem can be solved more easily and cleanly by using a dictionary than by using a list, do it that way, and consider alternatives only if it's clear that dictionaries are causing an unacceptable slowdown.

Lab

Using Dictionaries

Summary

- Dictionaries are powerful data structures, used for many purposes even within Python itself.
- Dictionary keys must be immutable, but any immutable object can be a dictionary key.
- Using keys means accessing collections of data more directly and with less code than many other solutions.

Control Flow

Chapter 8

This chapter covers

- Repeating code with a while loop
- Making decisions: the if-elif-else statement
- Iterating over a list with a for loop
- Using list and dictionary comprehensions
- Delimiting statements and blocks with indentation
- Evaluating Boolean values and expressions

The while Loop

```
1   u, v, x, y = 0, 0, 100, 30
2
3   while x > y:
4       u = u + y
5       x = x - y
6       if x < y + 2:
7           v = v + x
8           x = 0
9       else:
10          v = v + y + 2
11          x = x - y - 2
12  print(u, v)
```

The if-elif-else statement

```
1   x = 5
2
3   if x < 5:
4       y = -1
5       z = 5
6   elif x > 5:
7       y = 1
8       z = 11
9   else:
10      y = 0
11      z = 10
12
13  print(x, y, z)
```

The for loop

```
1  item_list = [3, "string1", 23, 14.0, "string2", 49, 64, 70]
2
3  for x in item_list:
4      if not isinstance(x, int):
5          continue
6      if not x % 7:
7          print("found an integer divisible by seven: %d" % x)
8          break
```

The range function

```
43  # Using the range() function:
44  for x in range(6):
45      print(x)
46
47  # Using the start parameter:
48  for x in range(2, 6):
49      print(x)
50
51  # Increment the sequence with 3 (default is 1):
52  for x in range(2, 30, 3):
53      print(x)
54
55  # Print all numbers from 0 to 5, and print a message when th
    ended:
56  for x in range(6):
57      print(x)
58  else:
59      print("Finally finished!")
```

Quick Check

- Suppose that you have a list x = [1, 3, 5, 0, -1, 3, -2], and you need to remove all negative numbers from that list. Write the code to do this.
- How would you count the total number of negative numbers in a list y = [[1, -1, 0], [2, 5, -9], [-2, -3, 0]]?
- What code would you use to print very low if the value of x is below -5, low if it's from -5 up to 0, neutral if it's equal to 0, high if it's greater than 0 up to 5, and very high if it's greater than 5?

Boolean values and expressions

- Python has a Boolean object type that can be set to either True or False. Any expression with a Boolean operation returns True or False.
  - The numbers 0, 0.0, and 0+0j are all False; any other number is True.
  - The empty string "" is False; any other string is True.
  - The empty list [] is False; any other list is True.
  - The empty dictionary {} is False; any other dictionary is True.
  - The empty set set() is False; any other set is True.
  - The special Python value None is always False.

Example

Writing a simple program to analyze a text file

Lab

Refactor word_count

Summary

- Python uses indentation to group blocks of code.
- Python has loops using while and for, and conditionals using if-elif-else.
- Python has the Boolean values True and False, which can be referenced by variables.
- Python also considers any 0 or empty value to be False and any nonzero or nonempty value to be True.

Functions

Chapter 9

This chapter covers

- Defining functions
- Using function parameters
- Passing mutable objects as parameters
- Understanding local and global variables
- Creating and using generator functions
- Creating and using lambda expressions
- Using decorators

Basic function definitions

```
1  # syntax
2  # def name(param1, param2,...):
3  #   body
4
5  # function that will return the factorial
6  def fact(n):
7      """ Return the factorial of the given number """
8      r = 1
9      while n > 0:
10         r = r * n
11         n = n -1
12     return r
13
14 print(fact(4))
```

Function with parameters

```
16  # function that will return the power
17  def power(x, y):
18      r = 1
19      while y > 0:
20          r = r * x
21          y = y -1
22      return r
23
```

Function with default parameter

```
# function with default par
def power2(x, y=2):
    r = 1
    while y > 0:
        r = r * x
        y = y -1
    return r

print(power2(3, 3))
print(power2(3))
```

Named parameter

```
26  # function with default param
27  def power2(x, y=2):
28      r = 1
29      while y > 0:
30          r = r * x
31          y = y -1
32      return r
33
34  print(power2(3, 3))
35  print(power2(3))
36
37  # named parameters
38  print(power2(y=4, x =2))
```

Variable numbers of arguments

```
40  # varargs
41  def maximum(*numbers):
42      if len(numbers) == 0:
43          return None
44      else:
45          maxnum = numbers[0]
46          for n in numbers[1:]:
47              if n > maxnum:
48                  maxnum = n
49          return maxnum
50
51  print(maximum(3,2,8))
52  print(maximum(1,5,9,-2,2))
```

Indefinite number of arguments

```
54  # indefinite number of arguments passed by keyword
55  def example_fun(x,y,**other):
56      print("x: {0}, y: {1}, keys in 'other': {2}".format(x,y,list(other.keys())))
57      other_total = 0
58      for k in other.keys():
59          other_total = other_total + other[k]
60      print("The total of values in 'other' is {0}".format(other_total))
61
62  example_fun(2,y="1", foo=3,bar=4)
```

Mutable objects as arguments

```
64  # mutable objects as arguments
65  def f(n, list1, list2):
66      list1.append(3)
67      list2 = [4,5,6]
68      n = n + 1
69
70  x = 5
71  y = [1,2]
72  z = [4,5]
73  f(x,y,z)
74  print(x, y, z)  # y is modified
```

Function as parameter

```
76  # assigning functions to variables
77  def f_to_kelvin(degrees_f):
78      return 273.15 + (degrees_f - 32) * 5 / 9
79
80  def c_to_kelvin(degrees_c):
81      return 273.15 + degrees_c
82
83  abs_temperature = f_to_kelvin
84  print(abs_temperature(32))
85
86  abs_temperature = c_to_kelvin
87  print(abs_temperature(0))
88
89  # place them in a list, tuples or dictionaries
90  t = {'FtoK':f_to_kelvin, 'CtoK':c_to_kelvin}
91  print(t['FtoK'](32))
92  print(t['CtoK'](0))
```

Lambda Expressions

- lambda expressions are anonymous little functions that you can quickly define inline.

```
1  t2 = {'FtoK': lambda deg_f: 273.15 + (deg_f - 32) * 5 / 9,
2        'CtoK': lambda deg_c: 273.15 + deg_c}
3
4  result = t2['FtoK'](32)
5  print(result)
```

Generator functions

- A generator function is a special kind of function that you can use to define your own iterators.
- When you define a generator function, you return each iteration's value using the yield keyword.
- The generator will stop returning values when there are no more iterations, or it encounters either an empty return statement or the end of the function.
- Local variables in a generator function are saved from one call to the next, unlike in normal functions

```python
def four():
    x = 0
    while x < 4:
        print("in generator, x =", x)
        yield x
        x += 1

for i in four():
    print(i)
```

Decorators

- A decorator is syntactic sugar for this process and lets you wrap one function inside another with a one-line addition.
- It still gives you exactly the same effect as the previous code, but the resulting code is much cleaner and easier to read.

```
1   def decorate(func):
2       print("in decorate function, decorating", func.__name__)
3       def wrapper_func(*args):
4           print("Executing", func.__name__)
5           return func(*args)
6       return wrapper_func
7
8   def myfunction(parameter):
9       print(parameter)
10
11  myfunction = decorate(myfunction)
12
13  myfunction("hello")
```

How to use decorators?

- Very simply, using a decorator involves two parts: defining the function that will be wrapping or "decorating" other functions.
- Use an @ followed by the decorator immediately before the wrapped function is defined.
- The decorator function should take a function as a parameter and return a function.
- Example on next slide.

```
17  def decorate(func):
18      print("in decorate function, decorating", func.__name__)
19      def wrapper_func(*args):
20          print("Executing", func.__name__)
21          return func(*args)
22      return wrapper_func
23
24  @decorate
25  def myfunction(parameter):
26      print(parameter)
27
28  myfunction("hello")
```

Lab

Useful functions

Summary

- External variables can easily be accessed within a function by using the global statement.
- Arguments may be passed by position or by parameter name.
- Default values may be provided for function parameters.
- Functions can collect arguments into tuples, giving you the ability to define functions that take an indefinite number of arguments.
- Functions can collect arguments into dictionaries, giving you the ability to define functions that take an indefinite number of arguments passed by parameter name.
- Functions are first-class objects in Python, which means that they can be assigned to variables, accessed by way of variables, and decorated.

Modules and scoping rules

Chapter 10

This chapter covers

- Defining a module
- Writing a first module
- Using the import statement
- Modifying the module search path
- Making names private in modules
- Importing standard library and third-party modules
- Understanding Python scoping rules and namespaces

What is a module?

- A module is a file containing code.
- It defines a group of Python functions or other objects, and the name of the module is derived from the name of the file.

A first module

- Create a text file called mymath.py, and in that text file, enter the Python code in listing 10.1.

```
1  """mymath - our example math module"""
2  pi = 3.14159
3  def area(r):
4      """area(r): return the area of a circle with radius r."""
5      global pi
6      return(pi * r * r)
```

To use the module

- Import the module
- Start using the module

```
1   import mymath
2
3   print(mymath.pi)
4
5   from mymath import pi
6
7   print(pi)
```

Modules

- A module is a file defining Python objects.
- If the name of the module file is modulename.py, the Python name of the module is modulename.
- You can bring a module named modulename into use with the import modulename statement. After this statement is executed, objects defined in the module can be accessed as modulename.objectname.
- Specific names from a module can be brought directly into your program by using the from modulename import objectname statement. This statement makes objectname accessible to your program without your needing to prepend it with modulename, and it's useful for bringing in names that are often used.

Where to place your own modules

- Place your modules in one of the directories that Python normally searches for modules.
- Place all the modules used by a Python program in the same directory as the program.
- Create a directory (or directories) to hold your modules, and modify the sys.path variable so that it includes this new directory (or directories).

Private names in modules

- The exception is that identifiers in the module beginning with an underscore can't be imported with from module import *.
- By starting all internal names (that is, names that shouldn't be accessed outside the module) with an underscore, you can ensure that from module import * brings in only those names that the user will want to access.

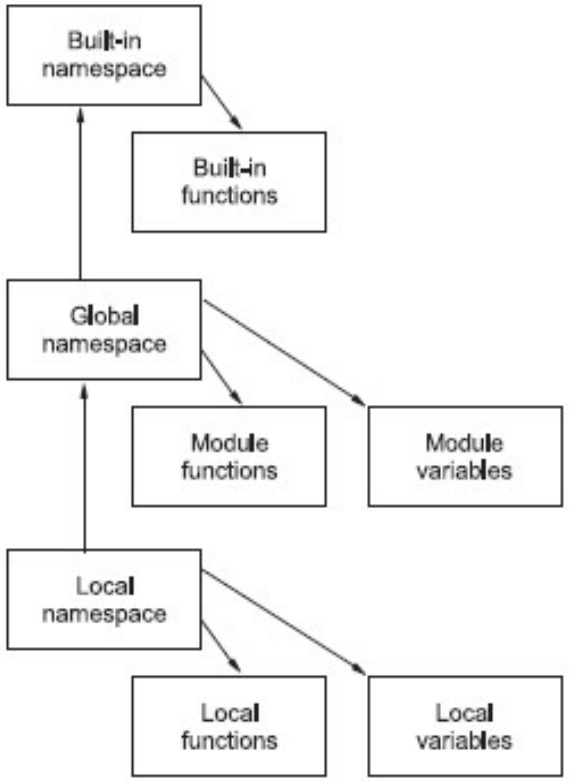
```
1  """modtest: our test module"""
2  def f(x):
3      return x
4
5  def _g(x):
6      return x
7
8  a = 4
9  _b = 2
```

Library and third-party modules

- After you've installed Python, all the functionality in these library modules is available to you.
- All that's needed is to import the appropriate modules, functions, classes, and so forth explicitly, before you use them.
- Available third-party modules and links to them are identified in the Python Package Index (pyPI), which will be discussed in chapter 19.
- You need to download these modules and install them in a directory in your module search path to make them available for import into your programs.

Python scoping rules and namespaces

- A namespace in Python is a mapping from identifiers to objects - that is, how Python keeps track of what variables and identifiers are active and what they point to.



Lab

Create a module

Summary

- Python modules allow you to put related code and objects into a file.
- Using modules also helps prevent conflicting variable names, because imported
- objects are normally named in association with their module.

Python programs

Chapter 11

This chapter covers

- Creating a very basic program
- Combining programs and modules
- Distributing Python applications

Creating a very basic program

- Any group of Python statements placed sequentially in a file can be used as a program, or script.
- But it's more standard and useful to introduce additional structure.
- In its most basic form, this task is a simple matter of creating a controlling function in a file and calling that function.

```
1  def main():
2      print("this is our first test script file")
3
4  main()
```

```
C:\pythoncodes\programs>python basicprogram.py
this is our first test script file
```

Command-line arguments

```
1  import sys
2
3  def main():
4      print("this is our second test script file")
5      print(sys.argv)
6
7  main()
```

```
C:\pythoncodes\programs>python cmdarg.py hello world
this is our second test script file
['cmdarg.py', 'hello', 'world']
```

Using the fileinput module

- It provides support for processing lines of input from one or more files.
- It automatically reads the command-line arguments (out of sys.argv) and takes them as its list of input files.
- Then it allows you to sequentially iterate through these lines.

```
1  import fileinput
2
3  def main():
4      for line in fileinput.input():
5          if not line.startswith('##'):
6              print(line, end="")
7
8  main()
```

Input files

```
1   ## sole1.tst: test data for the sole function
2   0 0 0
3   0 100 0
4   ##
5   0 100 100
```

```
1   ## sole2.tst: more test data for the sole function
2   12 15 0
3   ##
4   100 100 0
```

Quick Check

- Match the following ways of interacting with the command line and the correct use case for each:

Multiple arguments and options	sys.agrv
No arguments or just one argument	Use file_input module
Processing multiple files	Redirect standard input and output
Using the script as a filter	Use argparse module

Programs and Modules

- For small scripts that contain only a few lines of code, a single function works well.
- But if the script grows beyond this size, separating your controlling function from the rest of the code is a good option to take.
- The script in the next listing returns the English-language name for a given number between 0 and 99.

```python
import sys
# conversion mappings
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four', '5': 'five',
             '6': 'six', '7': 'seven', '8': 'eight', '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve', '3': 'thirteen', '4': 'fourteen',
             '5': 'fifteen', '6': 'sixteen', '7': 'seventeen', '8': 'eighteen', '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty', '6': 'sixty',
             '7': 'seventy', '8': 'eighty', '9': 'ninety'}

def num2words(num_string):
    if num_string == '0':
        return'zero'
    if len(num_string) > 2:
        return "Sorry can only handle 1 or 2 digit numbers"
    num_string = '0' + num_string
    tens, ones = num_string[-2], num_string[-1]
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]

def main():
    print(num2words(sys.argv[1]))

if __name__ == '__main__':
    main()
else:
    print("n2w loaded as a module")
```

Distributing Python applications

- Share the source files, of course, probably bundled in a zip or tar file.
- Assuming that the applications were written portably, you could also ship only the bytecode as .pyc files.
  - Wheels packages
  - zipapp and pex
  - py2exe and py2app
  - Creating executable programs with freeze

Summary

- Python scripts and modules in their most basic form are just sequences of Python statements placed in a file.
- Modules can be instrumented to run as scripts, and scripts can be set up so that they can be imported as modules.
- Scripts can be made executable on the UNIX, macOS, or Windows command lines. They can be set up to support command-line redirection of their input and output, and with the argparse module, it's easy to parse out complex combinations of command-line arguments.
- On macOS, you can use the Python Launcher to run Python programs, either individually or as the default application for opening Python files.

Summary

- On Windows, you can call scripts in several ways: by opening them with a double-click, using the Run window, or using a command-prompt window.
- Python scripts can be distributed as scripts, as bytecode, or in special packages called wheels.
- py2exe, py2app, and the freeze tool provide an executable Python program that runs on machines that don't contain a Python interpreter.
- Now that you have an idea of the ways to create scripts and applications, the next step is looking at how Python can interact with and manipulate filesystems.

Using the Filesystem

Chapter 12

This chapter covers

- Managing paths and pathnames
- Getting information about files
- Performing filesystem operations
- Processing all files in a directory subtree

os and os.path vs. pathlib

- The traditional way that file paths and filesystem operations have been handled in Python is by using functions included in the os and os.path modules.
- These functions have worked well enough but often resulted in more verbose code than necessary.
- Since Python 3.5, a new library, pathlib, has been added; it offers a more object-oriented and more unified way of doing the same operations.

Paths and pathnames

- All operating systems refer to files and directories with strings naming a given file or directory.
- Strings used in this manner are usually called pathnames (or sometimes just paths).
- Pathname semantics across operating systems are very similar because the filesystem on almost all operating systems is modeled as a tree structure, with a disk being the root and folders, subfolders, and so on being branches, subbranches, and so on.
- Different operating systems have different conventions regarding the precise syntax of pathnames.

Absolute and relative paths

- These operating systems allow two types of pathnames:
  - Absolute pathnames specify the exact location of a file in a filesystem without any ambiguity; they do this by listing the entire path to that file, starting from the root of the filesystem.
  - Relative pathnames specify the position of a file relative to some other point in the filesystem, and that other point isn't specified in the relative pathname itself; instead, the absolute starting point for relative pathnames is provided bythe context in which they're used.
- As examples, here are two Windows absolute pathnames:

  C:\Program Files\Doom

  D:\backup\June

Absolute and relative paths

- and here are two Linux absolute pathnames and a Mac absolute pathname:

  /bin/Doom

  /floppy/backup/June

  /Applications/Utilities

- and here are two Windows relative pathnames:

  mydata\project1\readme.txt

  games\tetris

- and these are Linux/UNIX/Mac relative pathnames:

  mydata/project1/readme.txt

  games/tetris

  Utilities/Java

The current working directory

- The directory that a Python program is in is called the current working directory for that program.
- This directory may be different from the directory the program resides in.

```
1   import os
2
3   # get current working directory
4   print(os.getcwd())
5
6   # get listing
7   print(os.listdir(os.curdir))
8
9   # change directory
10  os.chdir("foldername")
11  print(os.getcwd())
```

Accessing directories with pathlib

- To get the current directory with pathlib, you could do the following:

```
import pathlib
cur_path = pathlib.Path()
cur_path.cwd()
```

Manipulating pathnames

- To start, construct a few pathnames on different operating systems, using the os.path.join function.
- Note that importing os is sufficient to bring in theos.path submodule also; there's no need for an explicit import os.path statement.

```
import os
print(os.path.join('bin', 'utils', 'disktools'))
```

Manipulating pathnames with pathlib

- Start by constructing a few pathnames on different operating systems, using the path object's methods.

```
from pathlib import Path
cur_path = Path()
print(cur_path.joinpath('bin', 'utils', 'disktools'))
```

Useful constants and functions

- Checks whether the parent of the parent of path is a directory.

  os.path.isdir(os.path.join(path, os.pardir, os.curdir))

- Returns a list of filenames in the current working directory.

  os.listdir(os.curdir)

- The os.name constant returns the name of the Python module imported to handle the operating system–specific details.

  os.name

Getting information about files

- The most commonly used Python path-information functions are
  - os.path.exists
  - os.path.isfile
  - os.path.isdir
  - os.path.islink
  - os.path.ismount
  - os.path.samefile(path1, path2)
  - os.path.isabs(path)
  - os.path.getsize(path)
  - os.path.getmtime(path)
  - os.path.getatime(path)

More filesystem operations

- glob.glob("*")
  - The glob function from the glob module (named after an old UNIX function that did pattern matching) expands Linux/UNIX shell-style wildcard characters and character sequences in a pathname, returning the files in the current working directory that match.
- os.rename
  - To rename or move a file or directory.
- os.remove
  - To remove or delete a data file.
- os.makedirs or os.mkdir
- os.rmdir

Processing all files in a directory subtree

- Finally, a highly useful function for traversing recursive directory structures is the os.walk function.
- You can use it to walk through an entire directory tree, returning three things for each directory it traverses: the root, or path, of that directory; a list of its subdirectories; and a list of its files.
- When called, os.walk creates an iterator that recursively applies itself to all the directories contained in the top parameter. In other words, for each subdirectory subdir in names, os.walk recursively invokes a call to itself, of the form os.walk(subdir, ...).

```
1  import os
2
3  for root, dirs, files in os.walk(os.curdir):
4      print("{0} has {1} files".format(root, len(files)))
5      if ".git" in dirs:
6          dirs.remove(".git")
```

Summary

- Python provides a group of functions and constants that handle filesystem references (pathnames) and filesystem operations in a manner independent of the underlying operating system.
- For more advanced and specialized filesystem operations that typically are tied to a certain operating system or systems, look at the main Python documentation for the os, pathlib, and posix modules.

Reading and writing files

Chapter 13

This chapter covers

- Opening files and file objects
- Closing files
- Opening files in different modes
- Reading and writing text or binary data

Opening files and file objects

- The traditional way that file paths and filesystem operations have been handled in Python is by using functions included in the os and os.path modules.
- These functions have worked well enough but often resulted in more verbose code than necessary.
- Since Python 3.5, a new library, pathlib, has been added; it offers a more object-oriented and more unified way of doing the same operations.

Read text file: KORD.TXT

```
1   CHICAGO O'HARE INTERNATIONAL, IL, United States (KORD) 41-59N 087-55W 200M
2   Oct 10, 2017 - 10:44 PM EDT / 2017.10.11 0244 UTC
3   Wind: from the NE (040 degrees) at 14 MPH (12 KT):0
4   Visibility: 9 mile(s):0
5   Sky conditions: overcast
6   Weather: light rain
7   Precipitation last hour: 0.05 inches
8   Temperature: 59.0 F (15.0 C)
9   Dew Point: 55.9 F (13.3 C)
10  Relative Humidity: 89%
11  Pressure (altimeter): 30.03 in. Hg (1016 hPa)
12  ob: KORD 110244Z 04012KT 9SM -RA BKN013 BKN060 OVC100 15/13 A3003 RMK AO2 P0005 T01500133
13  cycle: 2
```

Program to read text file

```
1  myfile = "../data/KORD.TXT"
2
3  with open(myfile, 'r') as file_object:
4      # read the first line
5      line = file_object.readline()
6      print(line)
7
8      # keep reading
9      while file_object.readline() != "" :
10         line = file_object.readline()
11         print(line)
12
13 # close
14 file_object.close()
```

Opening files in write or other modes

- The second argument of the open command is a string denoting how the file should be opened.
    - 'r' means “Open the file for reading,”
    - 'w' means “Open the file for writing” (any data already in the file will be erased),
    - and 'a' means “Open the file for appending” (new data will be appended to the end of any data already in the file).
    - If you want to open the file for reading, you can leave out the second argument; 'r' is the default.

Reading and writing with pathlib

- In addition to its path-manipulation powers discussed in chapter 12, a Path object can be used to read and write text and binary files.
- This capability can be convenient because no open or close is required, and separate methods are used for text and binary operations.
- One limitation, however, is that you have no way to append by using Path methods, because writing replaces any existing content.

```python
from pathlib import Path

myfile = "mytextfile.txt"
mybinfile = "mybinfile"

# open text file
p_text = Path(myfile)

# write to file
p_text.write_text('Text file contents')

# read file
line = p_text.read_text()
print(line)

# open binary file
p_binary = Path(mybinfile)
p_binary.write_bytes(b'Binary file contents')

# read bin file
binline = p_binary.read_bytes()
print(binline)
```

Screen input/output and redirection

- Use the built-in input method to prompt for and read an input string.

```
1  # get string input
2  x = input("Enter file name to use: ")
3  print(x)
4
5  # get number
6  x = int(input("Enter your number: "))
7  print(x)
```

```
1  import sys
2
3  print("Write to the standard output.")
4
5  sys.stdout.write("Write to the standard output.\n")
6
7  # get user input
8  s = sys.stdin.readline()
9  print(s)
```

Summary

- File input and output in Python uses various built-in functions to open, read, write, and close files.
- In addition to reading and writing text, the struct module gives you the ability to read or write packed binary data.
- The pickle and shelve modules provide simple, safe, and powerful ways of saving and accessing arbitrarily complex Python data structures.

Exceptions

Chapter 14

This chapter covers

- Understanding exceptions
- Handling exceptions in Python
- Using the with keyword

General philosophy of errors and exception handling

- SOLUTION 1: DON'T HANDLE THE PROBLEM
  - The simplest way to handle this disk-space problem is to assume that there'll always be adequate disk space for whatever files you write and that you needn't worry about it.
  - Unfortunately, this option seems to be the most commonly used.
- SOLUTION 2: ALL FUNCTIONS RETURN SUCCESS/FAILURE STATUS
  - There are numerous ways to do this, but a typical method is to have
  - each function or procedure return a status value that indicates whether that function or procedure call executed successfully.
- SOLUTION 3: THE EXCEPTION MECHANISM
  - The code checks for errors on each attempted file write and passes an error status message back up to the calling procedure if an error is detected.

A more formal definition of exceptions

- The act of generating an exception is called raising or throwing an exception.
- The act of responding to an exception is called catching an exception, and the code that handles an exception is called exception-handling code or just an exception handler.

Handling different types of exceptions

- Depending on exactly what event causes an exception, a program may need to take different actions.
- An exception raised when disk space is exhausted needs to be handled quite differently from an exception that's raised if you run out of memory, and both of these exceptions are completely different from an exception that arises when a divide-by-zero error occurs.
- One way to handle these different types of exceptions is to globally record an error message indicating the cause of the exception, and have all exception handlers examine this error message and take appropriate action. In practice, a different method has proved to be much more flexible.

Exceptions in Python

- An exception is an object generated automatically by Python functions with a raise statement.
- After the object is generated, the raise statement, which raises an exception, causes execution of the Python program to proceed in a manner different from what would normally occur.
- Instead of proceeding with the next statement after the raise or whatever generated the exception, the current call chain is searched for a handler that can handle the generated exception.
- If such a handler is found, it's invoked and may access the exception object for more information.
- If no suitable exception handler is found, the program aborts with an error message.

Types of Python exceptions

- The Python exception set is hierarchically structured, as reflected by the indentation in this list of exceptions.
- Each type of exception is a Python class, which inherits from its parent exception type.
- This hierarchy is deliberate: Most exceptions inherit from Exception, and it's strongly recommended that any user-defined exceptions also subclass Exception, not BaseException.

```
BaseException
    SystemExit
    KeyboardInterrupt
    GeneratorExit
    Exception
        StopIteration
        ArithmeticError
            FloatingPointError
            OverflowError
            ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        ImportError
            ModuleNoteFoundError
        LookupError
            IndexError
            KeyError
        MemoryError
        NameError
            UnboundLocalError
        OSError
            BlockingIOError
            ChildProcessError
            ConnectionError
                BrokenPipeError
                ConnectionAbortedError
                ConnectionRefusedError
                ConnectionResetError
            FileExistsError
            FileNotFoundError
            InterruptedError
            IsADirectoryError
            NotADirectoryError
            PermissionError
```

Raising exceptions

- Error-checking code built into Python detects that the second input line requests an element at a list index that doesn't exist and raises an IndexError exception.

```
>>> alist = [1, 2, 3]
>>> element = alist[7]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```

Catching and handling exceptions

- By defining appropriate exception handlers, you can ensure that commonly encountered exceptional circumstances don't cause the program to fail; perhaps they display an error message to the user or do something else, perhaps even fix the problem, but they don't crash the program.

```
try:
    body
except exception_type1 as var1:
    exception_code1
except exception_type2 as var2:
    exception_code2
    .
    .
    .
except:
    default_exception_code
else:
    else_body
finally:
    finally_body
```

Where to use exceptions

- Exceptions are natural choices for handling almost any error condition.
- It's an unfortunate fact that error handling is often added when the rest of the program is largely complete, but exceptions are particularly good at intelligibly managing this sort of after-the-fact error-handling code (or, more optimistically, when you're adding more error handling after the fact).
- Exceptions are also highly useful in circumstances where a large amount of processing may need to be discarded after it becomes obvious that a computational branch in your program has become untenable.

Context managers using the with keyword

- Some situations, such as reading files, follow a predictable pattern with a set beginning and end.
- In the case of reading from a file, quite often the file needs to be open only one time: while data is being read.
- Then the file can be closed.
- In terms of exceptions, you can code this kind of file access like this:

```
try:
    infile = open(filename)
    data = infile.read()
finally:
    infile.close()
```

Context managers using the with keyword

- Python 3 offers a more generic way of handling situations like this: context managers.
- Context managers wrap a block and manage requirements on entry and departure from the block and are marked by the with keyword.
- File objects are context managers, and you can use that capability to read files:

```
with open(filename) as infile:
    data = infile.read()
```

Lab

Custom Exceptions

Summary

- Python's exception-handling mechanism and exception classes provide a rich system to handle runtime errors in your code.
- By using try, except, else, and finally blocks, and by selecting and even creating the types of exceptions caught, you can have very fine-grained control over how exceptions are handled and ignored.
- Python's philosophy is that errors shouldn't pass silently unless they're explicitly silenced.
- Python exception types are organized in a hierarchy because exceptions, like all objects in Python, are based on classes.

Classes and object-oriented programming

Chapter 15

This chapter covers

- Defining classes
- Using instance variables and @property
- Defining methods
- Defining class variables and methods
- Inheriting from other classes
- Making variables and methods private
- Inheriting from multiple classes

Defining classes

- A class in Python is effectively a data type.
- All the data types built into Python are classes, and Python gives you powerful tools to manipulate every aspect of a class's behavior.
- You define a class with the class statement:

```
class MyClass:
    body
```

- Create a new object of the class type (an instance of the class) by calling the class name as a function:

```
instance = MyClass()
```

Example class

- Class instances can be used as structures or records.
- Unlike C structures or Java classes, the data fields of an instance don't need to be declared ahead of time; they can be created on the fly.
- The following short example defines a class called Circle, creates a Circle instance, assigns a value to the radius field of the circle, and then uses that field to calculate the circumference of the circle

```
1   # define a class
2   class Circle:
3       def __init__(self):
4           self.radius = 1
5
6   # create an instance of a class
7   my_circle = Circle()
8   print(2 * 3.14 * my_circle.radius)
9
10  # set property
11  my_circle.radius = 5
12  print(2 * 3.14 * my_circle.radius)
```

Instance variables

- Take a look at the Circle class again, radius is an instance variable of Circle instances.
- That is, each instance of the Circle class has its own copy of radius, and the value stored in that copy may be different from the values stored in the radius variable in other instances.
- In Python, you can create instance variables as necessary by assigning to a field of a class instance:

```
instance.variable = value
```

- If the variable doesn't already exist, it's created automatically, which is how __init__ creates the radius variable.

Methods

- A method is a function associated with a particular class.
- You've already seen the special __init__ method, which is called on a new instance when that instance is created.

```
15  class Circle(Shape):
16      """Circle Class: inherits from Shape and has method area"""
17      pi = 3.14159
18      def __init__(self, r=1, x=0, y=0):
19          Shape.__init__(self, x, y)
20          self.radius = r
21      def area(self):
22          """Circle area method: returns the area of the circle."""
23          return self.radius * self.radius * self.pi
24      def __str__(self):
25          return "Circle of radius %s at coordinates (%d, %d)"\
26                  % (self.radius, self.x, self.y)
```

Class variables

- A class variable is a variable associated with a class, not an instance of a class, and is accessible by all instances of the class.
- A class variable might be used to keep track of some class-level information, such as how many instances of the class have been created at any point.
- A class variable is created by an assignment in the class body, not in the __init__ function.

```
15  class Circle(Shape):
16      """Circle Class: inherits from Shape and has method area"""
17      pi = 3.14159
18      def __init__(self, r=1, x=0, y=0):
19          Shape.__init__(self, x, y)
20          self.radius = r
21      def area(self):
22          """Circle area method: returns the area of the circle."""
23          return self.radius * self.radius * self.pi
24      def __str__(self):
25          return "Circle of radius %s at coordinates (%d, %d)"\
26                  % (self.radius, self.x, self.y)
```

Static methods and class methods

- You can invoke static methods even though no instance of that class has been created, although you can call them by using a class instance.
- To create a static method, use the @staticmethod decorator.

```
@staticmethod
def total_area():
    """Static method to total the areas of all Circles """
    total = 0
    for c in Circle.all_circles:
        total = total + c.area()
    return total
```

```
>>> circle.Circle.total_area()
31.415899999999997
```

Static methods and class methods

- Class methods are similar to static methods in that they can be invoked before an object of the class has been instantiated or by using an instance of the class.
- But class methods are implicitly passed the class they belong to as their first parameter, so you can code them more simply, as here.

```
@classmethod
def total_area(cls):
    total = 0
    for c in cls.all_circles:
        total = total + c.area()
    return total
```

```
>>> circle_cm.Circle.total_area()
15.70795
>>> c2.radius = 3
>>> circle_cm.Circle.total_area()
31.415899999999997
```

Try This

- Write a class method similar to total_area() that returns the total circumference of all circles.

Inheritance

- Parent and children relationship.
- Also known as superclass and subclass.

Square class

- Properties x and y are the coordinates of the Square object to be drawn on canvas.

```
1  class Square:
2      def __init__(self, side=1, x=0, y=0):
3          self.side = side
4          self.x = x
5          self.y = y
6
```

Circle class

- Circle class also have coordinates x and y.

```
 7  class Circle:
 8      def __init__(self, radius=1, x=0, y=0):
 9          self.radius = radius
10          self.x = x
11          self.y = y
```

Shape class

- Move all common properties to a more generic class.
- This class will be the parent (superclass).

```
15  class Shape:
16      def __init__(self, x, y):
17          self.x = x
18          self.y = y
19
```

Shape class (updated)

- Make Square class inherits from Shape class by pass the superclass's name in the bracket.
- The call superclass's init method to initialize it.

```
20  class Square(Shape):
21      def __init__(self, side=1, x=0, y=0):
22          super().__init__(x, y)
23          self.side = side
```

Circle class (updated)

- Do the same thing to Circle class.

```
25  class Circle(Shape):
26      def __init__(self, r=1, x=0, y=0):
27          super().__init__(x, y)
28          self.radius = r
```

Try This

- Rewrite the code for a Rectangle class to inherit from Shape. Because squares and rectangles are related, would it make sense to inherit one from the other? If so, which would be the base class, and which would inherit?
- How would you write the code to add an area() method for the Square class? Should the area method be moved into the base Shape class and inherited by circle, square, and rectangle? If so, what issues would result?

Inheritance with class and instance variables

- Inheritance allows an instance to inherit attributes of the class.
- Instance variables are associated with object instances, and only one instance variable of a given name exists for a given instance.

```
31  class P:
32      z = "Hello"
33      def set_p(self):
34          self.x = "Class P"
35      def print_p(self):
36          print(self.x)
37
38  class C(P):
39      def set_c(self):
40          self.x = "Class C"
41      def print_c(self):
42          print(self.x)
```

Private variables and methods

- A private variable or private method is one that can't be seen outside the methods of the class in which it's defined.
- Private variables and methods are useful for two reasons:
  - They enhance security and reliability by selectively denying access to important or delicate parts of an object's implementation,
  - and they prevent name clashes that can arise from the use of inheritance.
- A class may define a private variable and inherit from a class that defines a private variable of the same name, but this doesn't cause a problem, because the fact that the variables are private ensures that separate copies of them are kept.
- Any method or instance variable whose name begins—but doesn't end—with a double underscore (__) is private; anything else isn't private.

Mine class with private variables

```
1 class Mine:
2     def __init__(self):
3         self.x = 2
4         self.__y = 3
5     def print_y(self):
6         print(self.__y)
```

Try This

- Modify the Rectangle class's code to make the dimension variables private. What restriction will this modification impose on using the class?

Using @property for more flexible instance variables

- Python allows you as the programmer to access instance variables directly, without the extra machinery of the getter and setter methods often used in Java and other object-oriented languages.
- This lack of getters and setters makes writing Python classes cleaner and easier, but in some situations, using getter and setter methods can be handy.
- The answer is to use a property.
- A property combines the ability to pass access to an instance variable through methods like getters and setters and the straightforward access to instance variables through dot notation.

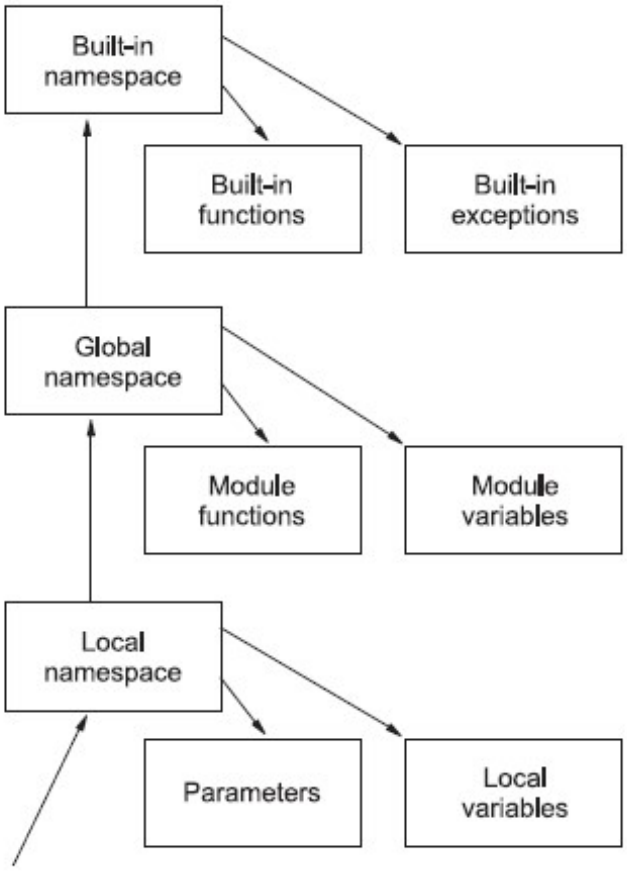
Using @property for more flexible instance variables

- To create a property, you use the property decorator with a method that has the property's name.

```
1  class Temperature:
2      def __init__(self):
3          self._temp_fahr = 0
4
5      @property
6      def temp(self):
7          return (self._temp_fahr - 32) * 5 / 9
8
9      @temp.setter
10     def temp(self, new_temp):
11         self._temp_fahr = new_temp * 9 / 5 + 32
```

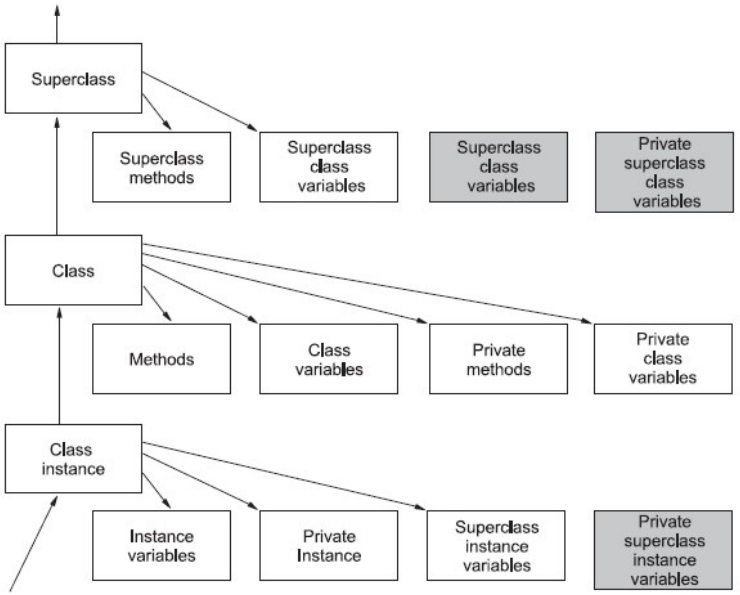
Scoping rules and namespaces for class instances

- When you're in a method of a class, you have direct access to the local namespace (parameters and variables declared in the method), the global namespace (functions and variables declared at the module level), and the built-in namespace (built-in functions and built-in exceptions). These three namespaces are searched in the following order: local, global, and built-in.



Scoping rules and namespaces for class instances

- You also have access through the self variable to the instance's namespace (instance variables, private instance variables, and superclass instance variables), its class's namespace (methods, class variables, private methods, and private class variables), and its superclass's namespace (superclass methods and superclass class variables).
- These three namespaces are searched in the order instance, class, and then superclass.



Destructors and memory management

- You've already seen class initializers (the __init__ methods).
- A destructor can be defined for a class as well.
- Python provides automatic memory management through a reference-counting mechanism.
- That is, it keeps track of the number of references to your instance; when this number reaches zero, the memory used by your instance is reclaimed, and any Python objects referenced by your instance have their reference counts decremented by one.
- You almost never need to define a destructor

Multiple inheritance

- Python places no such restrictions on multiple inheritance.
- A class can inherit from any number of parent classes in the same way that it can inherit from a single parent class.
- In the simplest case, none of the involved classes, including those inherited indirectly through a parent class, contains instance variables or methods of the same name.

Lab

HTML Classes

Summary

- Defining a class in effect creates a new data type.
- __init__ is used to initialize data when a new instance of a class is created, but it isn't a constructor.
- The self parameter refers to the current instance of the class and is passed as the first parameter to methods of a class.
- Static methods can be called without creating an instance of the class, so they don't receive a self parameter.
- Class methods are passed a cls parameter, which is a reference to the class, instead of self.

Summary

- All Python methods are virtual. That is, if a method isn't overridden in the subclass or private to the superclass, it's accessible by all subclasses.
- Class variables are inherited from superclasses unless they begin with two underscores (__), in which case they're private and can't be seen by subclasses. Methods can be made private in the same way.
- Properties let you have attributes with defined getter and setter methods, but they still behave like plain instance attributes.
- Python allows multiple inheritance, which is often used with mixin classes.

Regular expressions

Chapter 16

This chapter covers

- Understanding regular expressions
- Creating regular expressions with special characters
- Using raw strings in regular expressions
- Extracting matched text from strings
- Substituting text with regular expressions

What is a regular expression?

- A regular expression (regex) is a way of recognizing and often extracting data from certain patterns of text.
- A regex that recognizes a piece of text or a string is said to match that text or string.
- A regex is defined by a string in which certain characters (the so-called metacharacters) can have a special meaning, which enables a single regex to match many different specific strings.

Example

- Here's a program with a regular expression that counts how many lines in a text file contain the word hello.
- A line that contains hello more than once is counted only once.

```
1   import re
2
3   regexp = re.compile("hello")
4   count = 0
5
6   file = open("textfile", 'r')
7
8   for line in file.readlines():
9       if regexp.search(line):
10          count = count + 1
11
12  file.close()
13  print(count)
```

Regular expressions with special characters

- The previous example has a small flaw: It counts how many lines contain "hello" but ignores lines that contain "Hello" because it doesn't take capitalization into account.
- One way to solve this problem would be to use two regular expressions—one for "hello" and one for "Hello"—and test each against every line.
- A better way is to use the more advanced features of regular expressions.

```
3   regexp = re.compile("hello")
4   regexp = re.compile("hello|Hello")
5   regexp = re.compile("(h|H)ello")
6   regexp = re.compile("[hH]ello")
```

Regular expressions and raw strings

- A raw string looks similar to a normal string except that it has a leading r character immediately preceding the initial quotation mark of the string.
- Here are some raw strings:

```
r"Hello"
r"""\tTo be\n\tor not to be"""
r'Goodbye’
r'''12345'''
```

Extracting matched text from strings

- One of the most common uses of regular expressions is to perform simple pattern-based parsing on text.

surname, firstname middlename: phonenumber

```
regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"
                    r" (?P<first>[-a-zA-Z]+)"
                    r"( (?P<middle>([-a-zA-Z]+)))?"
                    r": (?P<phone>(\(\d{3}-)?\d{3}-\d{4})"
)
```

```
import re

regexp = re.compile(r"(?P<last>[-a-zA-Z]+),"
                    r" (?P<first>[-a-zA-Z]+)"
                    r"( (?P<middle>([-a-zA-Z]+)))?"
                    r": (?P<phone>(\(\d{3}-)?\d{3}-\d{4})"
)

file = open("textfile", 'r')
for line in file.readlines():
    result = regexp.search(line)
    if result == None:
        print("Oops, I don't think this is a record")
    else:
        lastname = result.group('last')
        firstname = result.group('first')
        middlename = result.group('middle')
        if middlename == None:
            middlename = ""
        phonenumber = result.group('phone')
    print('Name:', firstname, middlename, lastname,' Number:', phonenumber)
file.close()
```

Try This

- Making international calls usually requires a + and the country code.
- Assuming that the country code is two digits, how would you modify the code above to extract the + and the country code as part of the number? (Again, not all numbers have a country code.)
- How would you make the code handle country codes of one to three digits?

Substituting text with regular expressions

- In addition to extracting strings from text, you can use Python's regex module to find strings in text and substitute other strings in place of those that were found.

```
1  import re
2
3  string = "If the the problem is textual, use the the re module"
4
5  pattern = r"the the"
6
7  regexp = re.compile(pattern)
8
9  regexp.sub("the", string)
```

Try This

- In the previous activity, you extended a phone-number regular expression to also recognize a country code.
- How would you use a function to make any numbers that didn't have a country code now have +1 (the country code for the United States and Canada)?

Lab

Phone-Number Normalizer

Summary

- For a complete list and explanation of the regex special characters, refer to the Python documentation.
- In addition to the search and sub methods, many other methods can be used to split strings, extract more information from match objects, look for the positions of substrings in the main argument string, and precisely control the iteration of a regex search over an argument string.
- Besides the \d special sequence, which can be used to indicate a digit character, many other special sequences are listed in the documentation.
- There are also regex flags, which you can use to control some of the more esoteric aspects of how extremely sophisticated matches are carried out.

Data types as objects

Chapter 17

This chapter covers

- Treating types as objects
- Using types
- Creating user-defined classes
- Understanding duck typing
- Using special method attributes
- Subclassing built-in types

Types are objects, too

- This example is the first time you've seen the built-in type function in Python.
- It can be applied to any Python object and returns the type of that object.
- In this example, the function tells you that 5 is an int (integer) and that ['hello', 'goodbye'] is a list—things that you probably already knew.

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

Types are objects, too

- The object returned by type is an object whose type happens to be <class 'type'>; you can call it a type object.
- A type object is another kind of Python object who's only outstanding feature is the confusion that its name sometime causes.

```
>>> type(5)
<class 'int'>
>>> type(['hello', 'goodbye'])
<class 'list'>
```

Using types

- Now that you know that data types can be represented as Python type objects, what can you do with them?
- You can compare them, because any two Python objects can be compared.

```
>>> type("Hello") == type("Goodbye")
True
>>> type("Hello") == type(5)
False
```

Types and user-defined classes

- The most common reason to be interested in the types of objects, particularly
- instances of user-defined classes, is to find out whether a particular object is an instance of a class.
- After determining that an object is of a particular type, the code can treat it appropriately.

```
>>> class C:
...     pass
...
>>> class D:
...     pass
...
>>> class E(D):
...     pass
...
>>> x = 12
>>> c = C()
>>> d = D()
>>> e = E()
>>> isinstance(x, E)
False
>>> isinstance(c, E)
False
>>> isinstance(e, E)
True
>>> isinstance(e, D)
True
>>> isinstance(d, E)
False
>>> y = 12
>>> isinstance(y, type(5))
True
```

What is a special method attribute?

- A special method attribute is an attribute of a Python class with a special meaning to Python.
- It's defined as a method but isn't intended to be used directly as such.
- Special methods aren't usually directly invoked; instead, they're called automatically by Python in response to a demand made on an object of that class.
- Perhaps the simplest example is the __str__ special method attribute.
- If it's defined in a class, any time an instance of that class is used where Python requires a user-readable string representation of that instance, the __str__ method attribute is
- invoked, and the value it returns is used as the required string.

Example

```
1 class Color:
2     def __init__(self, red, green, blue):
3         self._red = red
4         self._green = green
5         self._blue = blue
6     def __str__(self):
7         return "Color: R={0:d}, G={1:d}, B={2:d}".format (self._red, self._green, self._blue)
```

```
>>> c = Color(15, 35, 3)
>>> print(c)
Color: R=15, G=35, B=3
```

Subclassing from built-in types

- Instead of creating a class for a typed list from scratch, as you did in the previous examples, you can subclass the list type and override all the methods that need to be aware of the allowed type.
- One big advantage of this approach is that your class has default versions of all list operations because it's a list already.
- The main thing to keep in mind is that every type in Python is a class, and if you need a variation on the behavior of a built-in type, you may want to consider subclassing that type.

```
class TypedListList(list):
    def __init__(self, example_element, initial_list=[]):
        self.type = type(example_element)
        if not isinstance(initial_list, list):
            raise TypeError("Second argument of TypedList must "
                            "be a list.")
        for element in initial_list:
            self.__check(element)
        super().__init__(initial_list)

    def __check(self, element):
        if type(element) != self.type:
            raise TypeError("Attempted to add an element of "
        "incorrect type to a typed list.")
        def __setitem__(self, i, element):
            self.__check(element)
            super().__setitem__(i, element)
```

Summary

- Python has the tools to do type checking as needed in your code, but by taking advantage of duck typing, you can write more flexible code that doesn't need to be as concerned with type checking.
- Special method attributes and subclassing built-in classes can be used to add list-like behavior to user-created classes.
- Python's use of duck typing, special method attributes, and subclassing makes it possible to construct and combine classes in a variety of ways.

Packages

Chapter 18

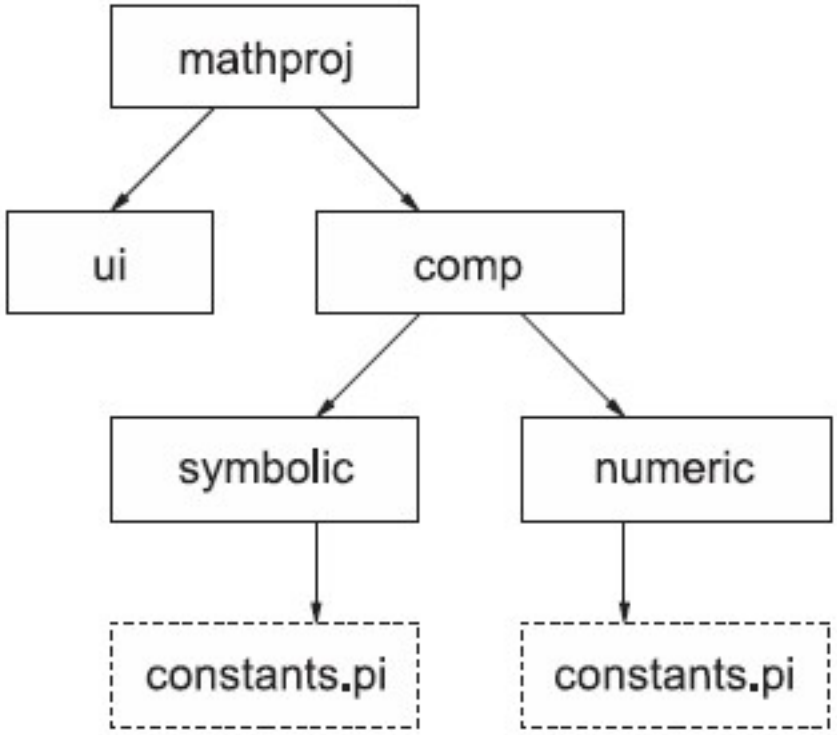
This chapter covers

- Defining a package
- Creating a simple package
- Exploring a concrete example
- Using the __all__ attribute
- Using packages properly

What is a package?

- A module is a file containing code.
- A module defines a group of usually related Python functions or other objects.
- The name of the module is derived from the name of the file.
- A package is a directory containing code and possibly further subdirectories.
- A package contains a group of usually related code files (modules).
- The name of the package is derived from the name of the main package directory.
- Packages are a natural extension of the module concept and are designed to handle very large projects.
- Just as modules group related functions, classes, and variables, packages group related modules.

A first example



Proper use of packages

- Packages shouldn't use deeply nested directory structures. Except for absolutely huge collections of code, there should be no need to do so. For most packages, a single top-level directory is all that's needed. A two-level hierarchy should be able to effectively handle all but a few of the rest. As written in The Zen of Python, by Tim Peters (see appendix A), "Flat is better than nested."
- Although you can use the __all__ attribute to hide names from from ... import * by not listing those names, doing so probably is not a good idea, because it's inconsistent. If you want to hide names, make them private by prefacing them with an underscore.

Lab

Create a Package

Summary

- Packages let you create libraries of code that span multiple files and directories.
- Using packages allows better organization of large collections of code than single modules would allow.
- You should be wary of nesting directories in your packages more than one or two levels deep unless you have a very large and complex library.

Using Python libraries

Chapter 19

This chapter covers

- Managing various data types—strings, numbers, and more
- Manipulating files and storage
- Accessing operating system services
- Using internet protocols and formats
- Developing and debugging tools
- Accessing PyPI (a.k.a. “The Cheese Shop”)
- Installing Python libraries and virtual environments using pip and venv

"Batteries included": The standard library

- In Python, what's considered to be the library consists of several components, including built-in data types and constants that can be used without an import statement, such as numbers and lists, as well as some built-in functions and exceptions.
- The largest part of the library is an extensive collection of modules.
- If you have Python, you also have libraries to manipulate diverse types of data and files, to interact with your operating system, to write servers and clients for many internet protocols, and to develop and debug your code.

String services modules

Modules	Description and possible uses
string	Compare with string constants, such as digits or whitespace; format strings (see chapter 6)
re	Search and replace text using regular expressions (see chapter 16)
struct	Interpret bytes as packed binary data, and read and write structured data to/from files
difflib	Use helpers for computing deltas, find differences between strings or sequences, and create patches and diff files
textwrap	Wrap and fill text, and format text by breaking lines or adding spaces

Data types modules

Modules	Description and possible uses
datetime, calendar	Date, time, and calendar operations
c	Container data types
enum	Allows creation of enumerator classes that bind symbolic names to constant values
array	Efficient arrays of numeric values
sched	Event scheduler
queue	Synchronized queue class
copy	Shallow and deep copy operations
pprint	Data pretty printer
typing	Support for annotating code with hints as to the types of objects, particularly of function parameters and return values

Numeric and mathematical modules

Modules	Description and possible uses
numbers	Numeric abstractbase classes
math, cmath	Mathematical functions for real and complex numbers
decimal	Decimal fixed-point and floating-point arithmetic
statistics	Functions for calculating mathematical statistics
fractions	Rational numbers
queue	Synchronized queue class
random	Generate pseudorandom numbers and choices, and shuffle sequences
itertools	Functions that create iterators for efficient looping
functools	Higher-order functions and operations on callable objects
operator	Standard operators as functions

File and storage modules

Modules	Description and possible uses
os.path	Perform common pathname manipulations
pathlib	Deal with pathnames in an object-oriented way
fileinput	Iterate over lines from multiple input streams
filecmp	Compare files and directories
tempfile	Generate temporary files and directories
glob, fnmatch	Use UNIX-style pathname and filename pattern handling
linecache	Gain random access to text lines
csv	Read and write CSV files
sqlite3	Work with a DB-API 2.0 interface for SQLite databases
zlib, gzip, bz2, zipfile, tarfile	Work with archive files and compressions

Operating system modules

Modules	Description and possible uses
os	Miscellaneous operating system interfaces
io	Core tools for working with streams
time	Time access and conversions
optparse	Powerful command-line option parser
logging	Logging facility for Python
getpass	Portable password input
curses	Terminal handling for character-cell displays
platform	Access to underlying platform's identifying data
Ctypes	Foreign function library for Python
select	Waiting for I/O completion

Modules supporting internet protocols and formats

Modules	Description and possible uses
socket, ssl	Low-level networking interface and SSL wrapper for socket Objects
email	Email and MIME handling package
json	JSON encoder and decoder
html.parser, html.entities	Parse HTML and XHTML
cgi, cgitb	Common Gateway Interface support
urllib.request, urllib.parse	Open and parse URLs
socketserver	Framework for network servers
http.server	HTTP servers

Development, debugging, and runtime modules

Modules	Description and possible uses
pydoc	Documentation generator and online help system
doctest	Test interactive Python examples
unittest	Unit testing framework
test.support	Utility functions for tests
pdb	Python debugger
trace	Trace or track Python statement execution
sys	System-specific parameters and functions
gc	Garbage collector interface
inspect	Inspect live objects

Installing Python libraries using pip and venv

- Python offers pip as the current solution to both problems. pip tries to find the module in the Python Package index (more about that soon), downloads it and any dependencies, and takes care of the installation.
- The basic syntax of pip is quite simple.
- To install the popular requests library from the command line, for example, all you have to do is

```
python3.6 -m pip install requests
python3.6 -m pip install –-upgrade requests
python3.6 -m pip install requests==2.11.1
python3.6 -m pip install --user requests
```

Virtual environments

- You have another, better option if you need to avoid installing libraries in the system Python.
- This option is called a virtual environment (virtualenv).
- A virtual environment is a self-contained directory structure that contains both an installation of Python and its additional packages.
- Because the entire Python environment is contained in the virtual environment, the libraries and modules installed there can't conflict with those in the main system or in other virtual environments, allowing different applications to use different versions on both Python and its packages.

```
python3.6 -m venv test-env
test-env\Scripts\activate.bat
pip install requests
```

PyPI (a.k.a. "The Cheese Shop")

- Although distutils packages get the job done, there's one catch: You have to find the correct package, which can be a chore.
- And after you've found a package, it would be nice to have a reasonably reliable source from which to download that package.
- To meet this need, various Python package repositories have been made available over the years.
- Currently, the official (but by no means the only) repository for Python code is the Python Package Index, or PyPI (formerly also known as "The Cheese Shop," after the Monty Python sketch) on the Python website.
- You can access it from a link on the main page or directly at https://pypi.python.org.
- PyPI is the logical next stop if you can't find the functionality you want with a search of the standard library.

Summary

- Python has a rich standard library that covers more common situations than many other languages, and you should check what's in the standard library carefully before looking for external modules.
- If you do need an external module, prebuilt packages for your operating system are the easiest option, but they're sometimes older and often hard to find.
- The standard way to install from source is to use pip, and the best way to prevent conflicts among multiple projects is to create virtual environments with the venv module.
- Usually, the logical first step in searching for external modules is the Python Package Index (PyPI).

Basic file wrangling

Chapter 20

This chapter covers

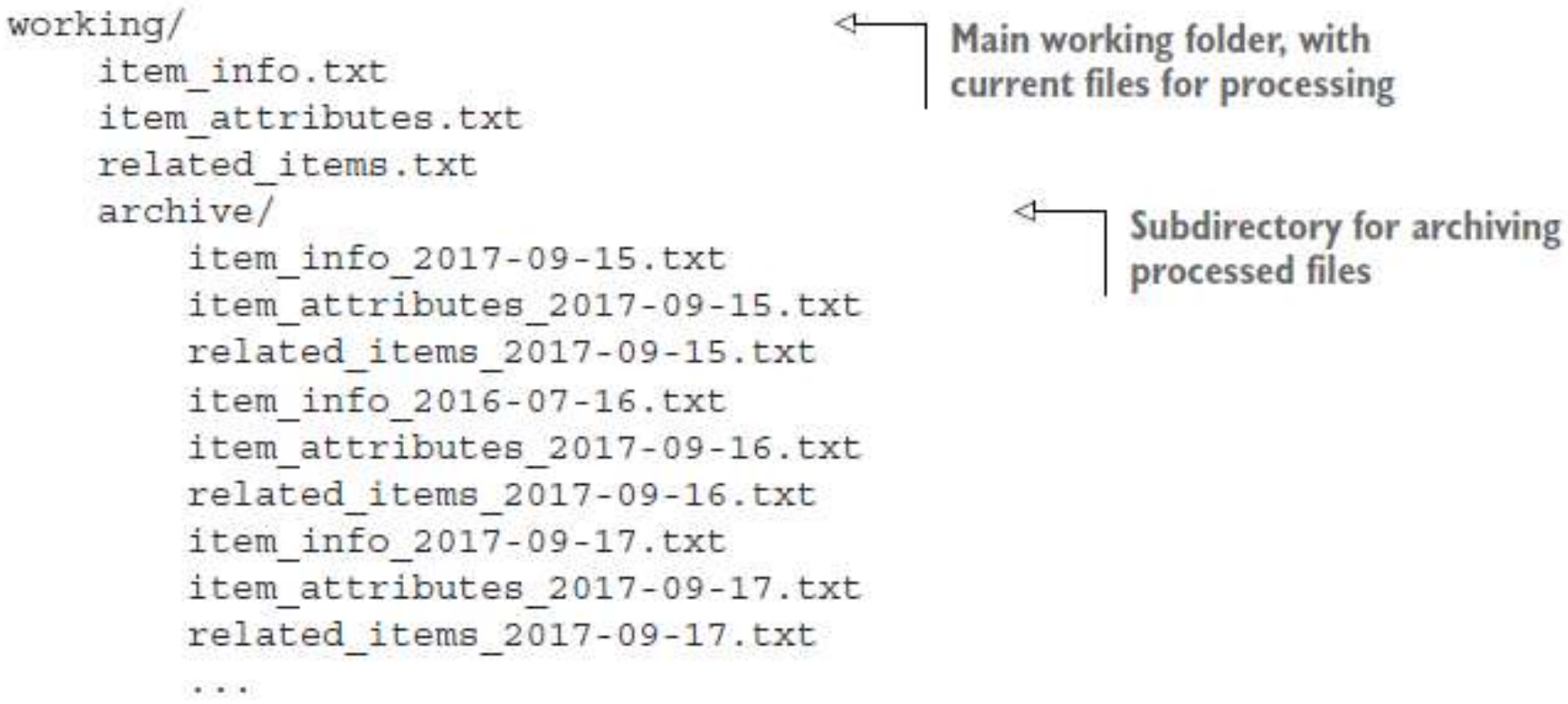
- Moving and renaming files
- Compressing and encrypting files
- Selectively deleting files

The problem: The never-ending flow of data files

- Many systems generate a continuous series of data files.
- These files might be the log files from an e-commerce server or a regular process; they might be a nightly feed of product information from a server; they might be automated feeds of items for online advertising; historical data of stock trades; or they might come from a thousand other sources.
- They're often flat text files, uncompressed, with raw data that's either an input or a byproduct of other processes.
- In spite of their humble nature, however, the data they contain has some potential value, so the files can't be discarded at the end of the day—which means that every day, their numbers grow.
- Over time, files accumulate until dealing with them manually becomes unworkable and until the amount of storage they consume becomes unacceptable.

Scenario: The product feed from hell

- A typical situation example is a daily feed of product data.
- This data might be coming in from a supplier or going out for online marketing, but the basic aspects are the same.
- The simplest thing you might do is mark the files with the dates on which they were received and move them to an archive folder.
- That way, each new set of files can be received, processed, renamed, and moved out of the way so that the process can be repeated with no loss of data.
- After a few repetitions, the directory structure might look something like this:


working/
item_info.txt
item_attributes.txt
related_items.txt
archive/
item_info_2017-09-15.txt
item_attributes_2017-09-15.txt
related_items_2017-09-15.txt
item_info_2016-07-16.txt
item_attributes_2017-09-16.txt
related_items_2017-09-16.txt
item_info_2017-09-17.txt
item_attributes_2017-09-17.txt
related_items_2017-09-17.txt
...
Main working folder, with current files for processing
Subdirectory for archiving processed files

How to solve it?

- First, you need to rename the files so that the current date is added to the filename.
- To do that, you need to get the names of the files you want to rename; then you need to get the stem of the filenames without the extensions.
- When you have the stem, you need to add a string based on the current date, add the extension back to the end, and then actually change the filename and move it to the archive directory.

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")
    paths = cur_path.glob(FILE_PATTERN)

    for path in paths:
        new_filename = "{}_{}{}".format(path.stem, date_string, path.suffix)
        new_path = cur_path.joinpath(ARCHIVE, new_filename)
        path.rename(new_path)
```

Sets the pattern to match files and the archive directory

A directory named "archive" must exist for this code to run.

Uses the date object from the datetime library to create a date string based on today's date

Creates a new path from the current path, the archive directory, and the new filename

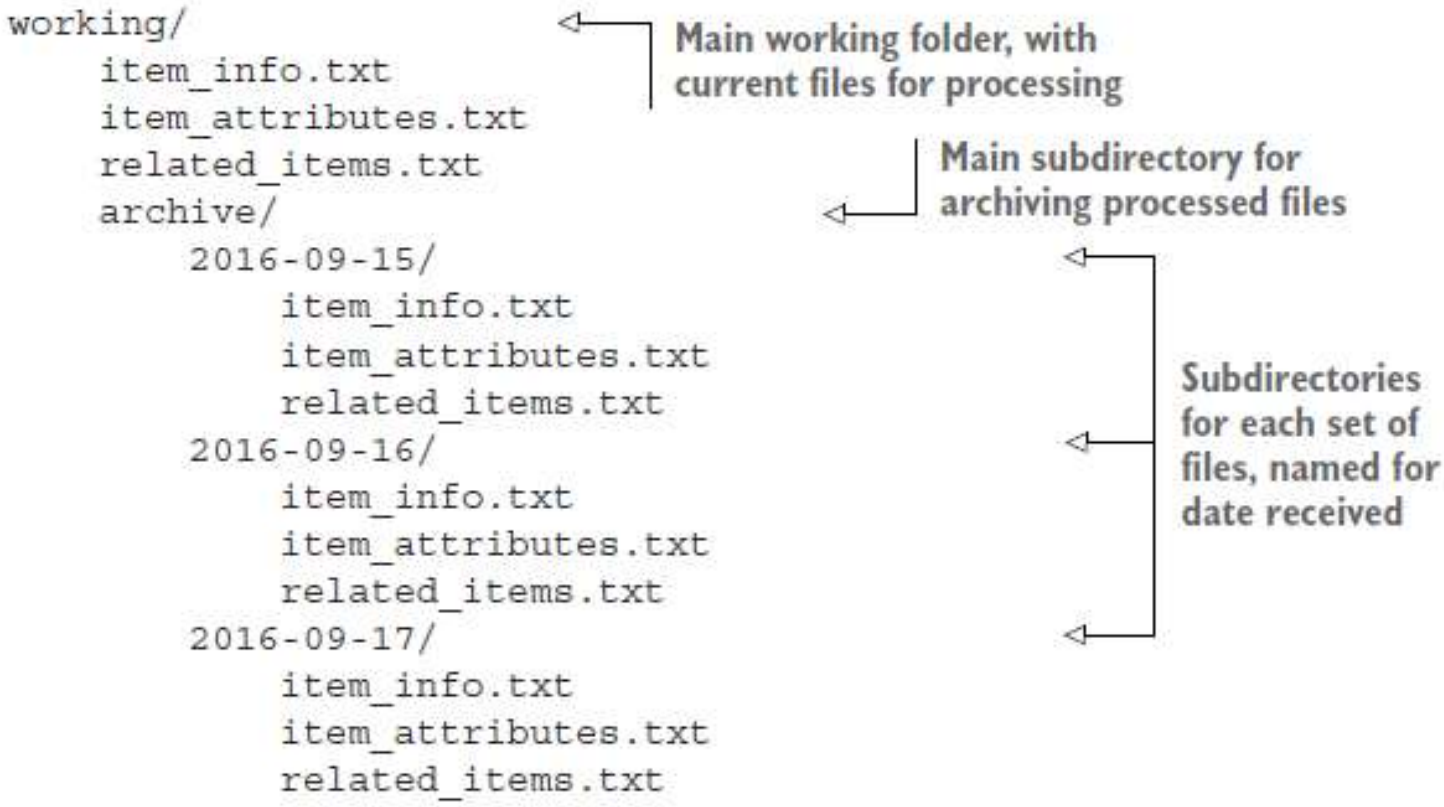
Renames (and moves) the file as one step

More organization

- The solution to storing files described in the previous section works, but it does have some disadvantages.
- For one thing, as the files accumulate, managing them might become a bit more trouble, because over the course of a year, you'd have 365 sets of related files in the same directory, and you could find the related files only by inspecting their names.
- If the files arrive more frequently, of course, or if there are more related files in a set, the hassle would be even greater.

A better solution

- To mitigate this problem, you can change the way you archive the files.
- Instead of changing the filenames to include the dates on which they were received, you can create a separate subdirectory for each set of files and name that subdirectory after the date received.
- Your directory structure might look like this (next slide)

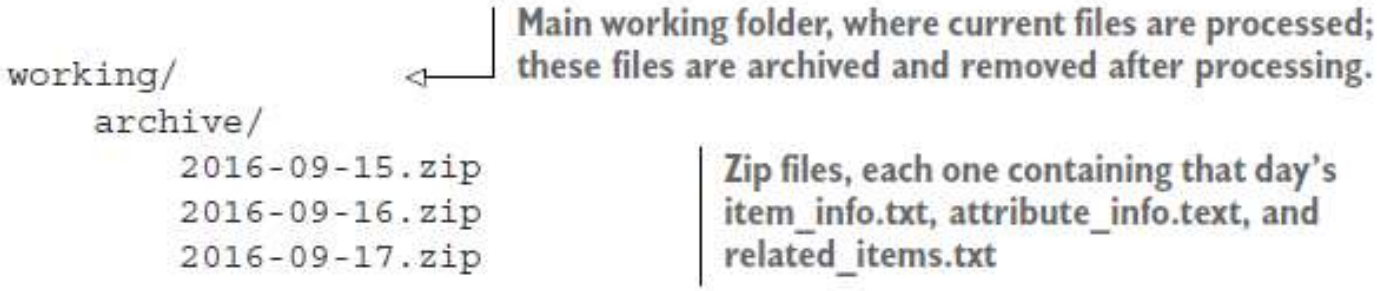

working/
item_info.txt
item_attributes.txt
related_items.txt
archive/
2016-09-15/
item_info.txt
item_attributes.txt
related_items.txt
2016-09-16/
item_info.txt
item_attributes.txt
related_items.txt
2016-09-17/
item_info.txt
item_attributes.txt
related_items.txt
Main working folder, with current files for processing
Main subdirectory for archiving processed files
Subdirectories for each set of files, named for date received

```
import datetime
import pathlib

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")

    new_path = cur_path.joinpath(ARCHIVE, date_string)
    new_path.mkdir()

    paths = cur_path.glob(FILE_PATTERN)

    for path in paths:
        path.rename(new_path.joinpath(path.name))
```

Note that this directory needs to be created only once, before the files are moved into it.

Compressing files

- If the space that the files are taking up is an issue, the next approach you might consider is compressing them.



```
import datetime
import pathlib
import zipfile

FILE_PATTERN = "*.txt"
ARCHIVE = "archive"

if __name__ == '__main__':

    date_string = datetime.date.today().strftime("%Y-%m-%d")

    cur_path = pathlib.Path(".")
    paths = cur_path.glob(FILE_PATTERN)

    zip_file_path = cur_path.joinpath(ARCHIVE, date_string + ".zip")
    zip_file = zipfile.ZipFile(str(zip_file_path), "w")

 for path in paths:
     zip_file.write(str(path))
     path.unlink()
```

Imports zipfile library

Creates the path to the zip file in the archive directory

Opens the new zip file object for writing; str() is needed to convert a Path to a string.

Writes the current file to the zip file

Removes the current file from the working directory

Grooming files @ Deleting

- The process of removing files after they reach a certain age is sometimes called grooming.
- Suppose that after several months of receiving a set of data files every day and archiving them in a zip file, you're told that you should retain only one file a week of the files that are more than one month old.
- The simplest grooming script removes any files that you no longer need - in this case, all but one file a week for anything older than a month old.

```
from datetime import datetime, timedelta
import pathlib
import zipfile

FILE_PATTERN = "*.zip"
ARCHIVE = "archive"
ARCHIVE_WEEKDAY = 1
if __name__ == '__main__':
    cur_path = pathlib.Path(".")
    zip_file_path = cur_path.joinpath(ARCHIVE)

    paths = zip_file_path.glob(FILE_PATTERN)
    current_date = datetime.today()

    for path in paths:
        name = path.stem
        path_date = datetime.strptime(name, "%Y-%m-%d")
        path_timedelta = current_date - path_date
        if path_timedelta > timedelta(days=30) and path_date.weekday() !=
    ARCHIVE_WEEKDAY:
            path.unlink()
```

Gets a datetime object for the current day

strptime parses a string into a datetime object based on the format string.

path.stem returns the filename without any extension.

Subtracting one date from another yields a timedelta object.

timedelta(days=30) creates a timedelta object of 30 days; the weekday() method returns an integer for the day of the week, with Monday = 0.

Summary

- The pathlib module can greatly simplify file operations such as finding the root and extension, moving and renaming, and matching wildcards.
- As the number and complexity of files increase, automated archiving solutions are vital, and Python offers several easy ways to create them.
- You can dramatically save storage space by compressing and grooming data files.

Processing data files

Chapter 21

This chapter covers

- Using ETL (extract-transform-load)
- Reading text data files (plain text and CSV)
- Reading spreadsheet files
- Normalizing, cleaning, and sorting data
- Writing data files

Welcome to ETL

- The need to get data out of files, parse it, turn it into a useful format, and then do something with it has been around for as long as there have been data files.
- In fact, there is a standard term for the process: extract-transform-load (ETL).
- The extraction refers to the process of reading a data source and parsing it, if necessary.
- The transformation can be cleaning and normalizing the data, as well as combining, breaking up, or reorganizing the records it contains.
- The loading refers to storing the transformed data in a new place, either a different file or a database.

Text encoding: ASCII, Unicode, and others

- The Unicode encoding called UTF-8 accepts the basic ASCII characters without any change but also allows an almost unlimited set of other characters and symbols according to the Unicode standard.
- Even with Unicode, there'll be occasions when your text contains values that can't be successfully encoded.

```
open('test.txt', 'wb').write(bytes([65, 66, 67, 255, 192,193]))
open('test.txt', errors='ignore').read()
open('test.txt', errors='replace').read()
open('test.txt', errors='surrogateescape').read()
open('test.txt', errors='backslashreplace').read()
```

Unstructured text

- Unstructured text files are the easiest sort of data to read but the hardest to extract information from.
- Processing unstructured text can vary enormously, depending on both the nature of the text and what you want to do with it, so any comprehensive discussion of text processing is beyond the scope of this course.

Call me Ishmael. Some years ago--never mind how long precisely--
having little or no money in my purse, and nothing particular
to interest me on shore, I thought I would sail about a little
and see the watery part of the world. It is a way I have
of driving off the spleen and regulating the circulation.
Whenever I find myself growing grim about the mouth;
whenever it is a damp, drizzly November in my soul; whenever I
find myself involuntarily pausing before coffin warehouses,
and bringing up the rear of every funeral I meet;
and especially whenever my hypos get such an upper hand of me,

Reads all of file as a single string

Splits on two newlines together

```
>>> moby_text = open("moby_01.txt").read()
>>> moby_paragraphs = moby_text.split("\n\n")
>>> print(moby_paragraphs[1])
There now is your insular city of the Manhattoes, belted round by wharves
as Indian isles by coral reefs--commerce surrounds it with her surf.
Right and left, the streets take you waterward.  Its extreme downtown
is the battery, where that noble mole is washed by waves, and cooled
by breezes, which a few hours previous were out of sight of land.
Look at the crowds of water-gazers there.
```

Reads all of the file as a single string

Makes everything lowercase

```
>>> moby_text = open("moby_01.txt").read()
>>> moby_paragraphs = moby_text.split("\n\n")
>>> moby = moby_paragraphs[1].lower()
```

Removes periods

Removes commas

```
>>> moby = moby.replace(".", "")
>>> moby = moby.replace(",", "")
>>> moby_words = moby.split()
>>> print(moby_words)
['there', 'now', 'is', 'your', 'insular', 'city', 'of', 'the', 'manhattoes,',
     'belted', 'round', 'by', 'wharves', 'as', 'indian', 'isles', 'by',
     'coral', 'reefs--commerce', 'surrounds', 'it', 'with', 'her', 'surf',
     'right', 'and', 'left,', 'the', 'streets', 'take', 'you', 'waterward',
     'its', 'extreme', 'downtown', 'is', 'the', 'battery,', 'where', 'that',
     'noble', 'mole', 'is', 'washed', 'by', 'waves,', 'and', 'cooled', 'by',
     'breezes,', 'which', 'a', 'few', 'hours', 'previous', 'were', 'out',
     'of', 'sight', 'of', 'land', 'look', 'at', 'the', 'crowds', 'of',
     'water-gazers', 'there']
```

Delimited flat files

- This file is a simple example of temperature data in delimited format:

```
State|Month Day, Year Code|Avg Daily Max Air Temperature (F)|Record Count for
     Daily Max Air Temp (F)
Illinois|1979/01/01|17.48|994
Illinois|1979/01/02|4.64|994
Illinois|1979/01/03|11.05|994
Illinois|1979/01/04|9.51|994
Illinois|1979/05/15|68.42|994
Illinois|1979/05/16|70.29|994
Illinois|1979/05/17|75.34|994
Illinois|1979/05/18|79.13|994
Illinois|1979/05/19|74.94|994
```

Example solution

- Whatever character is being used as the delimiter, if you know what character it is, you can write your own code in Python to break each line into its fields and return them as a list.
- In the previous case, you can use the string split() method to break a line into a list of values:

```
>>> line = "Illinois|1979/01/01|17.48|994"
>>> print(line.split("|"))
['Illinois', '1979/01/01', '17.48', '994']
```

The csv module

- The csv module is a perfect case of Python's "batteries included" philosophy.
- The csv module has been tested and optimized, and it has features that you probably wouldn't bother to write if you had to do it yourself, but that are truly handy and time-saving when available.

```
>>> results = []
>>> for line in open("temp_data_pipes_00a.txt"):
...     fields = line.strip().split("|")

...     results.append(fields)
...
>>> results
[['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature (F)',
     'Record Count for Daily Max Air Temp (F)'], ['Illinois', '1979/01/01',
     '17.48', '994'], ['Illinois', '1979/01/02', '4.64', '994'], ['Illinois',
     '1979/01/03', '11.05', '994'], ['Illinois', '1979/01/04', '9.51',
     '994'], ['Illinois', '1979/05/15', '68.42', '994'], ['Illinois', '1979/
     05/16', '70.29', '994'], ['Illinois', '1979/05/17', '75.34', '994'],
     ['Illinois', '1979/05/18', '79.13', '994'], ['Illinois', '1979/05/19',
     '74.94', '994']]
```

```
>>> import csv
>>> results = [fields for fields in
     csv.reader(open("temp_data_pipes_00a.txt", newline=''), delimiter="|")]
>>> results
[['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature (F)',
     'Record Count for Daily Max Air Temp (F)'], ['Illinois', '1979/01/01',
     '17.48', '994'], ['Illinois', '1979/01/02', '4.64', '994'], ['Illinois',
     '1979/01/03', '11.05', '994'], ['Illinois', '1979/01/04', '9.51',
     '994'], ['Illinois', '1979/05/15', '68.42', '994'], ['Illinois', '1979/
     05/16', '70.29', '994'], ['Illinois', '1979/05/17', '75.34', '994'],
     ['Illinois', '1979/05/18', '79.13', '994'], ['Illinois', '1979/05/19',
     '74.94', '994']]
```

Reading a csv file as a list of dictionaries

- In the preceding examples, you got a row of data back as a list of fields.
- This result works fine in many cases, but sometimes it may be handy to get the rows back as dictionaries where the field name is the key.
- For this use case, the csv library has a DictReader, which can take a list of fields as a parameter or can read them from the first line of the data. If you want to open the data with a DictReader, the code would look like this:

```
>>> results = [fields for fields in csv.DictReader(open("temp_data_01.csv",
     newline=''))]
>>> results[0]
OrderedDict([('Notes', ''), ('State', 'Illinois'), ('State Code', '17'),
     ('Month Day, Year', 'Jan 01, 1979'), ('Month Day, Year Code', '1979/01/
     01'), ('Avg Daily Max Air Temperature (F)', '17.48'), ('Record Count for
     Daily Max Air Temp (F)', '994'), ('Min Temp for Daily Max Air Temp (F)',
     '6.00'), ('Max Temp for Daily Max Air Temp (F)', '30.50'), ('Avg Daily
     Min Air Temperature (F)', '2.89'), ('Record Count for Daily Min Air Temp
     (F)', '994'), ('Min Temp for Daily Min Air Temp (F)', '-13.60'), ('Max
     Temp for Daily Min Air Temp (F)', '15.80'), ('Avg Daily Max Heat Index
     (F)', 'Missing'), ('Record Count for Daily Max Heat Index (F)', '0'),
     ('Min for Daily Max Heat Index (F)', 'Missing'), ('Max for Daily Max
     Heat Index (F)', 'Missing'), ('Daily Max Heat Index (F) % Coverage',
     '0.00%')])
```

Excel files

- The other common file format that I discuss in this chapter is the Excel file, which is the format that Microsoft Excel uses to store spreadsheets.
- As it happens, Python's standard library doesn't have a module to read or write Excel files.
- To read that format, you need to install an external module.
- Fortunately, several modules are available to do the job.
- For this example, you use one called OpenPyXL, which is available from the Python package repository.
- You can install it with the following command from a command line

```
$pip install openpyxl
```

```
>>> from openpyxl import load_workbook
>>> wb = load_workbook('temp_data_01.xlsx')
>>> results = []
>>> ws = wb.worksheets[0]
>>> for row in ws.iter_rows():
...     results.append([cell.value for cell in row])
...
>>> print(results)
[['Notes', 'State', 'State Code', 'Month Day, Year', 'Month Day, Year Code',
     'Avg Daily Max Air Temperature (F)', 'Record Count for Daily Max Air
     Temp (F)', 'Min Temp for Daily Max Air Temp (F)', 'Max Temp for Daily
     Max Air Temp (F)', 'Avg Daily Max Heat Index (F)', 'Record Count for
     Daily Max Heat Index (F)', 'Min for Daily Max Heat Index (F)', 'Max for
     Daily Max Heat Index (F)', 'Daily Max Heat Index (F) % Coverage'],
     [None, 'Illinois', 17, 'Jan 01, 1979', '1979/01/01', 17.48, 994, 6,
     30.5, 'Missing', 0, 'Missing', 'Missing', '0.00%'], [None, 'Illinois',
     17, 'Jan 02, 1979', '1979/01/02', 4.64, 994, -6.4, 15.8, 'Missing', 0,
     'Missing', 'Missing', '0.00%'], [None, 'Illinois', 17, 'Jan 03, 1979',
     '1979/01/03', 11.05, 994, -0.7, 24.7, 'Missing', 0, 'Missing',
     'Missing', '0.00%'], [None, 'Illinois', 17, 'Jan 04, 1979', '1979/01/
     04', 9.51, 994, 0.2, 27.6, 'Missing', 0, 'Missing', 'Missing', '0.00%'],
     [None, 'Illinois', 17, 'May 15, 1979', '1979/05/15', 68.42, 994, 61,
     75.1, 'Missing', 0, 'Missing', 'Missing', '0.00%'], [None, 'Illinois',
     17, 'May 16, 1979', '1979/05/16', 70.29, 994, 63.4, 73.5, 'Missing', 0,
     'Missing', 'Missing', '0.00%'], [None, 'Illinois', 17, 'May 17, 1979',
     '1979/05/17', 75.34, 994, 64, 80.5, 82.6, 2, 82.4, 82.8, '0.20%'],
     [None, 'Illinois', 17, 'May 18, 1979', '1979/05/18', 79.13, 994, 75.5,
     82.1, 81.42, 349, 80.2, 83.4, '35.11%'], [None, 'Illinois', 17, 'May 19,
     1979', '1979/05/19', 74.94, 994, 66.9, 83.1, 82.87, 78, 81.6, 85.2,
     '7.85%']]
```

Data cleaning

- One common problem you'll encounter in processing text-based data files is dirty data.
- By dirty, it means that there are all sorts of surprises in the data, such as null values, values that aren't legal for your encoding, or extra whitespace.
- The data may also be unsorted or in an order that makes processing difficult.
- The process of dealing with situations like these is called data cleaning.

Data cleaning steps

Cleaning

Sorting

Try This

- How would you handle the fields with 'Missing' as possible values for math calculations? Can you write a snippet of code that averages one of those columns?
- What would you do with the average column at the end so that you could also report the average coverage?
- In your opinion, would the solution to this problem be at all linked to the way that the 'Missing' entries were handled?

Data cleaning issues and pitfalls

Beware of whitespace and null characters.

Beware punctuation.

Break down and debug the steps.

Writing data files

- These files may be used as input for other applications and analysis, either by people or by other applications.
- Usually, you have a particular file specification listing what fields of data should be included, what they should be named, what format and constraints there should be for each, and so on.

```
>>> temperature_data = [['State', 'Month Day, Year Code', 'Avg Daily Max Air
     Temperature (F)', 'Record Count for Daily Max Air Temp (F)'],
     ['Illinois', '1979/01/01', '17.48', '994'], ['Illinois', '1979/01/02',
     '4.64', '994'], ['Illinois', '1979/01/03', '11.05', '994'], ['Illinois',
     '1979/01/04', '9.51', '994'], ['Illinois', '1979/05/15', '68.42',
     '994'], ['Illinois', '1979/05/16', '70.29', '994'], ['Illinois', '1979/
     05/17', '75.34', '994'], ['Illinois', '1979/05/18', '79.13', '994'],
     ['Illinois', '1979/05/19', '74.94', '994']]
>>> csv.writer(open("temp_data_03.csv", "w",
     newline='')).writerows(temperature_data)
```

```
>>> fields = ['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature
     (F)', 'Record Count for Daily Max Air Temp (F)']
>>> dict_writer = csv.DictWriter(open("temp_data_04.csv", "w"),
     fieldnames=fields)
>>> dict_writer.writeheader()
>>> dict_writer.writerows(data)
>>> del dict_writer
```

```
>>> from openpyxl import Workbook
>>> data_rows = [fields for fields in csv.reader(open("temp_data_01.csv"))]
>>> wb = Workbook()
>>> ws = wb.active
>>> ws.title = "temperature data"
>>> for row in data_rows:
...     ws.append(row)
...
>>> wb.save("temp_data_02.xlsx")
```

Packaging data files

- If you have several related data files, or if your files are large, it may make sense to package them in a compressed archive.
- Although various archive formats are in use, the zip file remains popular and almost universally accessible to users on almost every platform.

Lab

Weather Observations

Summary

- ETL (extract-transform-load) is the process of getting data from one format, making sure that it's consistent, and then putting it in a format you can use. ETL is the basic step in most data processing.
- Encoding can be problematic with text files, but Python lets you deal with some encoding problems when you load files.
- Delimited or CSV files are common, and the best way to handle them is with the csv module.
- Spreadsheet files can be more complex than CSV files but can be handled much the same way.
- Currency symbols, punctuation, and null characters are among the most common data cleaning issues; be on the watch for them.
- Presorting your data file can make other processing steps faster.

Data over the network

Chapter 22

This chapter covers

- Fetching files via FTP/SFTP, SSH/SCP, and HTTPS
- Getting data via APIs
- Structured data file formats: JSON and XML

Fetching files

- Before you can do anything with data files, you have to get them.
- Sometimes, this process is very easy, such as manually downloading a single zip archive, or maybe the files have been pushed to your machine from somewhere else.
- Quite often, however, the process is more involved.
- Maybe a large number of files needs to be retrieved from a remote server, files need to be retrieved regularly, or the retrieval process is sufficiently complex to be a pain to do manually.
- In any of those cases, you might well want to automate fetching the data files with Python.

Using Python to fetch files from an FTP server

```
>>> import ftplib
>>> ftp = ftplib.FTP('tgftp.nws.noaa.gov')
>>> ftp.login()
'230 Login successful.'
>>> ftp.cwd('data')                          >>>
'250 Directory successfully changed.'   '250
>>> ftp.nlst()                               >>>
['climate', 'fnmoc', 'forecasts', 'hurricane_products', 'ls_SS_services',
     'marine', 'nsd_bbsss.txt', 'nsd_cccc.txt', 'observations', 'products',
     'public_statement', 'raw', 'records', 'summaries', 'tampa',
     'watches_warnings', 'zonecatalog.curr', 'zonecatalog.curr.tar']
>>> x = ftp.retrbinary('RETR observations/metar/decoded/KORD.TXT',
     open('KORD.TXT', 'wb').write)
'226 Transfer complete.'
```

Fetching files with SFTP

- If the data requires more security, such as in a corporate context in which business data is being transferred over the network, it's fairly common to use SFTP.
- SFTP is a full-featured protocol that allows file access, transfer, and management over a Secure Shell (SSH) connection.
- Python doesn't have an SFTP/SCP client module in its standard library, but a community-developed library called paramiko manages SFTP operations as well as SSH connections.

```
>>> import paramiko
>>> t = paramiko.Transport((hostname, port))
>>> t.connect(username, password)
>>> sftp = paramiko.SFTPClient.from_transport(t)
```

Retrieving files over HTTP/HTTPS

- The requests library is by far the easiest and most reliable way to access HTTP/HTTPS servers from Python code.
- Again, requests is easiest to install with pip install requests.
- The following example code fetches the monthly temperature data for Heathrow Airport since 1948 - a text file that's served via a web server.

```
>>> import requests
>>> response = requests.get("http://www.metoffice.gov.uk/pub/data/weather/uk/
     climate/stationdata/heathrowdata.txt")
```

```
>>> print(response.text)
Heathrow (London Airport)
Location 507800E 176700N, Lat 51.479 Lon -0.449, 25m amsl

Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic Kipp & Zonen sensor marked with a #,
     otherwise sunshine data taken from a Campbell Stokes recorder.
   yyyy  mm    tmax    tmin      af    rain     sun
                degC    degC    days      mm   hours
   1948   1     8.9     3.3     ---    85.0     ---
   1948   2     7.9     2.2     ---    26.0     ---
   1948   3    14.2     3.8     ---    14.0     ---
   1948   4    15.4     5.1     ---    35.0     ---
   1948   5    18.1     6.9     ---    57.0     ---
```

Fetching data via an API

```
>>> import requests
>>> response = requests.get("http://marsweather.ingenology.com/v1/latest/
    ?format=json")
>>> response.text
'{"report": {"terrestrial_date": "2017-01-08", "sol": 1573, "ls": 295.0,
    "min_temp": -74.0, "min_temp_fahrenheit": -101.2, "max_temp": -2.0,
    "max_temp_fahrenheit": 28.4, "pressure": 872.0, "pressure_string":
    "Higher", "abs_humidity": null, "wind_speed": null, "wind_direction": "-
    -", "atmo_opacity": "Sunny", "season": "Month 10", "sunrise": "2017-01-
    08T12:29:00Z", "sunset": "2017-01-09T00:45:00Z"}}'
>>> response = requests.get("http://marsweather.ingenology.com/v1/archive/
    ?sol=155&format=json")
>>> response.text
'{"count": 1, "next": null, "previous": null, "results":
    [{"terrestrial_date": "2013-01-18", "sol": 155, "ls": 243.7, "min_temp":
    -64.45, "min_temp_fahrenheit": -84.01, "max_temp": 2.15,
    "max_temp_fahrenheit": 35.87, "pressure": 9.175, "pressure_string":
    "Higher", "abs_humidity": null, "wind_speed": 2.0, "wind_direction":
    null, "atmo_opacity": null, "season": "Month 9", "sunrise": null,
    "sunset": null}]}'
```

Structured data formats

- Although APIs sometimes serve plain text, it's much more common for data served from APIs to be served in a structured file format.
- The two most common file formats are JSON and XML.
- Both of these formats are built on plain text but structure their contents so that they're more flexible and able to store more complex information.

JSON data

```
>>> import json
>>> import requests
>>> response = requests.get("http://marsweather.ingenology.com/v1/latest/
     ?format=json")
>>> weather = json.loads(response.text)
>>> weather
{'report': {'terrestrial_date': '2017-01-10', 'sol': 1575, 'ls': 296.0,
     'min_temp': -58.0, 'min_temp_fahrenheit': -72.4, 'max_temp': 0.0,
     'max_temp_fahrenheit': None, 'pressure': 860.0, 'pressure_string':
     'Higher', 'abs_humidity': None, 'wind_speed': None, 'wind_direction': '-
     -', 'atmo_opacity': 'Sunny', 'season': 'Month 10', 'sunrise': '2017-01-
     10T12:30:00Z', 'sunset': '2017-01-11T00:46:00Z'}}
>>> weather['report']['sol']
1575
```

Pretty Printing

```
>>> from pprint import pprint as pp
>>> pp(weather)
{'report': {'abs_humidity': None,
            'atmo_opacity': 'Sunny',
            'ls': 296.0,
            'max_temp': 0.0,
            'max_temp_fahrenheit': None,
            'min_temp': -58.0,
            'min_temp_fahrenheit': -72.4,
            'pressure': 860.0,
            'pressure_string': 'Higher',
            'season': 'Month 10',
            'sol': 1575,
            'sunrise': '2017-01-10T12:30:00Z',
            'sunset': '2017-01-11T00:46:00Z',
            'terrestrial_date': '2017-01-10',
            'wind_direction': '--',
            'wind_speed': None}}
```

XML data

- XML (eXtensible Markup Language) has been around since the end of the 20th century.
- XML uses an angle-bracket tag notation similar to HTML, and elements are nested within other elements to form a tree structure.
- XML was intended to be readable by both machines and humans, but XML is often so verbose and complex that it's very difficult for people to understand.
- Nevertheless, because XML is an established standard, it's quite common to find data in XML format.
- And although XML is machine-readable, it's very likely that you'll want to translate it into something a bit easier to deal with.

```
<dwml xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
     www.w3.org/2001/XMLSchema-instance" version="1.0"
     xsi:noNamespaceSchemaLocation="http://www.nws.noaa.gov/forecasts/xml/
     DWMLgen/schema/DWML.xsd">
  <head>
    <product srsName="WGS 1984" concise-name="glance" operational-
     mode="official">
      <title>
NOAA's National Weather Service Forecast at a Glance
      </title>
      <field>meteorological</field>
      <category>forecast</category>
      <creation-date refresh-frequency="PT1H">2017-01-08T02:52:41Z</creation-
     date>
    </product>
    <source>
      <more-information>http://www.nws.noaa.gov/forecasts/xml/</more-
     information>
      <production-center>
Meteorological Development Laboratory
<sub-center>Product Generation Branch</sub-center>
      </production-center>
      <disclaimer>http://www.nws.noaa.gov/disclaimer.html</disclaimer>
      <credit>http://www.weather.gov/</credit>
      <credit-logo>http://www.weather.gov/images/xml_logo.gif</credit-logo>
      <feedback>http://www.weather.gov/feedback.php</feedback>
    </source>
  </head>
  <data>
    <location>
      <location-key>point1</location-key>
      <point latitude="41.78" longitude="-88.65"/>
    </location>
    …
  </data>
</dwml>
```

How to read XML data?

- For simple data extraction, the handiest utility I've found is a library called xmltodict, which parses your XML data and returns a dictionary that reflects the tree.
- In fact, behind the scenes it uses the standard library's expat XML parser, parses your XML document into a tree, and uses that tree to create the dictionary.

```
>>> import xmltodict
>>> data = xmltodict.parse(open("observations_01.xml").read())
```

Lab

Track Curiosity's Weather

Summary

- Using a Python script may not be the best choice for fetching files. Be sure to consider the options.
- Using the requests module is your best bet for fetching files by using HTTP/HTTPS and Python.
- Fetching files from an API is very similar to fetching static files.
- Parameters for API requests often need to be quoted and added as a query string to the request URL.
- JSON-formatted strings are quite common for data served from APIs, and XML is also used.
- Scraping sites that you don't control may not be legal or ethical and requires consideration not to overload the server.

Saving data

Chapter 23

This chapter covers

- Storing data in relational databases
- Using the Python DB-API
- Accessing databases through an Object
- Relational Mapper (ORM)
- Understanding NoSQL databases and how they differ from relational databases

Relational databases

- Relational databases have long been a standard for storing and manipulating data.
- They're a mature technology and a ubiquitous one.
- Python can connect with a number relational databases, but we don't have the time or the inclination to go through the specifics of each one in this course.
- Instead, because Python handles databases in a mostly consistent way, we are going to illustrate the basics with one of them - sqlite3 - and then discuss some differences and considerations in choosing and using a relational database for data storages.

The Python Database API

- Python handles SQL database access very similarly across several database implementations because of PEP-249 (www.python.org/dev/peps/pep-0249/), which specifies some common practices for connecting to SQL databases.
- Commonly called the Database API or DB-API, it was created to encourage "code that is generally more portable across databases, and a broader reach of database connectivity."
- Thanks to the DB-API, the examples of SQLite that you see in this chapter are quite similar to what you'd use for PostgreSQL, MySQL, or several other databases.

SQLite: Using the sqlite3 database

- Although it's not suited for large, high-traffic applications, sqlite3 has two advantages:
  - Because it's part of the standard library, it can be used anywhere you need a database without worrying about adding dependencies.
  - sqlite3 stores all of its records in a local file, so it doesn't need both a client and server, which would be the case for PostgreSQL, MySQL, and other larger databases.
- To use a sqlite3 database, the first thing you need is a Connection object.
- Getting a Connection object requires only calling the connect function with the name of file that will be used to store the data.

```
import sqlite3
conn = sqlite3.connect("datafile.db")
```

```python
import sqlite3


def connect_db():
    conn = sqlite3.connect("datafile.db")
    return conn


def get_cursor(conn):
    cursor = conn.cursor()
    print(cursor)
    return cursor


def create_table(conn, cursor):
    # cursor.execute(
    #     "create table people (id integer primary key, name text, count integer)")
    cursor.execute("insert into people (name, count) values ('Bob', 1)")
    cursor.execute(
        "insert into people (name, count) values (?, ?)", ("Jill", 15))
    cursor.execute("insert into people (name, count) values (:username, :usercount)", {
                   "username": "Joe", "usercount": 10})
    conn.commit()
```

```
26  def get_data(conn, cursor):
27      result = cursor.execute("select * from people")
28      print(result.fetchall())
29      result = cursor.execute(
30          "select * from people where name like :name", {"name": "bob"})
31      print(result.fetchall())
32
33
34  def update_data(conn, cursor):
35      cursor.execute("update people set count=? where name=?", (20, "Jill"))
36      result = cursor.execute("select * from people")
37      print(result.fetchall())
38
39
40  def get_all(conn, cursor):
41      result = cursor.execute("select * from people")
42      for row in result:
43          print(row)
```

```
46  if __name__ == "__main__":
47      conn = connect_db()
48      cursor = get_cursor(conn)
49      # create_table(conn, cursor)
50      get_data(conn, cursor)
51      update_data(conn, cursor)
52      get_data(conn, cursor)
53      get_all(conn, cursor)
```

Making database handling easier with an ORM

- There are a few problems with the DB-API database client libraries mentioned earlier in this chapter and their requirement to write raw SQL.
  - Different SQL databases have implemented SQL in subtly different ways.
  - The second drawback is the need to use raw SQL statements.
  - The need to write SQL means that you need to think in at least two languages: Python and a specific SQL variant.
- Given those issues, people wanted a way to handle databases in Python that was easier to manage and didn't require anything more than writing regular Python code.
- The solution is an Object Relational Mapper (ORM), which converts, or maps, relational database types and structures to objects in Python.
- Two of the most common ORMs in the Python world are the Django ORM and SQLAlchemy, although of course there are many others.

SQLAlchemy

- SQLAlchemy is the other big-name ORM in the Python space.
- SQLAlchemy's goal is to automate redundant database tasks and provide Python object-based interfaces to the data while still allowing the developer control of the database and access to the underlying SQL.
- You can install SQLAlchemy in your environment with pip:

```
pip install sqlalchemy
```

NoSQL databases

- Although relational databases are all about normalizing data within related tables, other approaches look at data differently.
- Quite commonly, these types of databases are referred to as NoSQL databases, because they usually don't adhere to the row/column/table structure that SQL was created to describe.
- Rather than handle data as collections of rows, columns, and tables, NoSQL databases can look at the data they store as key-value pairs, as indexed documents, and even as graphs.
- Many NoSQL databases are available, all with somewhat different ways of handling data.
- In general, they're less likely to be strictly normalized, which can make retrieving information faster and easier.

Lab

Create a Database

Summary

- Python has a Database API (DB-API) that provides a generally consistent interface for clients of several relational databases.
- Using an Object Relational Mapper (ORM) can make database code even more standard across databases.
- Using an ORM also lets you access relational databases through Python code and objects rather than SQL queries.
- Tools such as Alembic work with ORMs to use code to make reversible changes to a relational database schema.
- Key:value stores such as Redis provide quick in-memory data access.
- MongoDB provides scalability without the strict structure of relational databases.

Exploring data

Chapter 24

This chapter covers

- Python's advantages for handling data
- Jupyter Notebook
- pandas
- Data aggregation
- Plots with matplotlib

Python's advantages for exploring data

- Python has become one of the leading languages for data science and continues to grow in that area.
- However, Python isn't always the fastest language in terms of raw performance.
- Conversely, some data-crunching libraries, such as NumPy, are largely written in C and heavily optimized to the point that speed isn't an issue.
- In addition, considerations such as readability and accessibility often outweigh pure speed; minimizing the amount of developer time needed is often more important.
- Python is readable and accessible, and both on its own and in combination with tools developed in the Python community, it's an enormously powerful tool for manipulating and exploring data.

Python can be better than a spreadsheet

- Spreadsheets have been the tools of choice for ad-hoc data manipulation for decades.
- People who are skilled with spreadsheets can make them do truly impressive tricks: spreadsheets can combine different but related data sets, pivot tables, use lookup tables to link data sets, and much more.
- But although people everywhere get a vast amount of work done with them every day, spreadsheets do have limitations, and Python can help you go beyond those limitations;
    - Most spreadsheet software has a row limit—currently, about 1 million rows.
    - Spreadsheets are two-dimensional grids, rows and columns, or at best stacks of grids, which limits the ways you can manipulate and think about complex data.
- With Python, you can code your way around the limitations of spreadsheets and manipulate data the way you want.

Python and pandas

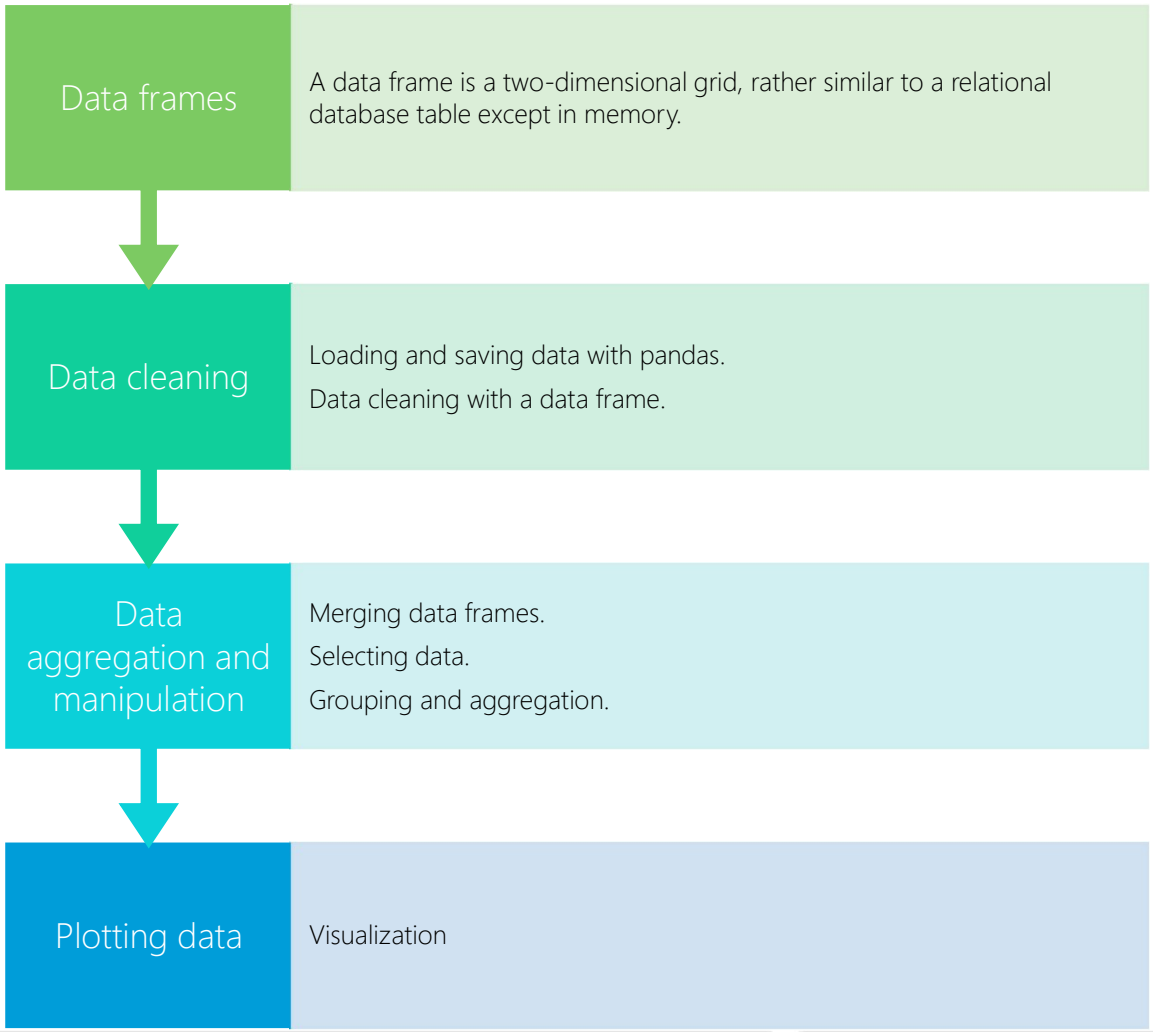
- One of the now-standard tools for handling data in Python – pandas - was created to automate the boring heavy lifting of handling data sets.
- pandas was created to make manipulating and analyzing tabular or relational data easy by providing a standard framework for holding the data, with convenient tools for frequent operations.
- As a result, it's almost more of an extension to Python than a library, and it changes the way you can interact with data.
- The plus side is that after you grok how pandas work, you can do some impressive things and save a lot of time.

Installing pandas

- pandas is easy to install with pip.
- It's often used along with matplotlib for plotting, so you can install both tools from the command line with this code:

```
pip install pandas matplotlib
```

Next steps – let's crunch it!



Summary

- Python offers many benefits for data handling, including the ability to handle very large data sets and the flexibility to handle data in ways that match your needs.
- Jupyter notebook is a useful way to access Python via a web browser, which also makes improved presentation easier.
- pandas is a tool that makes many common data-handling operations much easier, including cleaning, combining, and summarizing data.
- pandas also makes simple plotting much easier.

Thank you!