

TTPS4870 Applied Python for Data Science

TTPS4870: Student Guide



All written content, source code, and formatting are the property of © 2019 by C.J. Associates, All Rights Reserved. No portion of this material may be duplicated or reused in any way without the express written consent of author or authoring firm. For information please contact Info@triveratech.com or visit www.triveratech.com.

All software or hardware products referenced herein are trademarks of their respective holders. Products and company names are the trademarks and registered trademarks of their respective owners. Trivera Technologies has used its best efforts to distinguish proprietary trademarks from descriptive names by following the capitalization style used by the manufacturer.

Copyright © 2019 Trivera Technologies LLC. All rights reserved.

Trivera Technologies LLC

Collaborative IT Training, Mentoring & Courseware Solutions

Trivera Technologies LLC is a premier provider of **IT education and training, structured mentoring programs, custom courseware design & licensing** and **specialized education** services and solutions. Our extensive team of subject matter experts, engaging instructors and courseware designers brings years of current, practical, real-world experience into every classroom.

Our goal is to make you a success. Whether you're a manager arranging training for your company, a trainer or training firm using our courseware materials at your valued client site, or a student attending a course presented by one of our experts, **our core mission is to develop and deliver exactly the program you need to achieve your goals**, held to the highest industry quality standards, with complete support and assistance before, during and after your event.

We are a skills-oriented services firm that offers introductory through advanced level training in hundreds of IT topics. Our courses are fully customizable to suit your unique requirements, and can be presented live in your classroom, virtually for private delivery for your team, or in public training courses available around the globe. We **wholly-own** the majority of the courses we teach, providing us with the ultimate flexibility to create the most effective, targeted programs. Our expert delivery and management team services your event from end to end, producing and delivering targeted events with limited burden on your own firm, providing you with the most value for your training dollar.

Trivera Technologies Services Include

- **World Wide Instructor Led Training offered Onsite, Online, Self-Paced & Blended Solutions**
- **Extensive Open Enrollment Public Schedule**
- **Collaborative Mentoring Programs, Skills Immersion Camps & Code Academy Programs**
- **Skills Assessment Services; Gap Training Solutions and Corporate Training Plans**
- **Corporate Training & Vendor Management Solutions**
- **Custom Course Development Services**
- **Courseware Licensing – Corporate or OnDemand**
- **Flexible Pricing Programs and Options**
- **Satisfaction Guaranteed Services**

Trivera Technologies

Educate. Collaborate. Accelerate!

Training • Mentoring • Courseware • Programs
In-Person • Online Live • Public Schedule • Custom

Training@triveratech.com | +609.647.7572 phone
www.triveratech.com



100% Woman-Owned Small Business

A Sample of our Course Offerings includes....

> Application Development, Programming & Coding:

Intro to Programming • Java / JEE • C # / .Net • ASP.Net • VB.Net • C++ • C • COBOL • R • Scala • SQL • Cloud • Spring • Spring Batch / Boot / Security / Cloud • Hibernate • JSF • Struts • Secure Coding • Microservices • Web Services • MVC • RESTful Services • Design Patterns • Cloud

> Agile & TDD: Agile Development • TDD / BDD • Unit

Testing • Frameworks • Scrum • SAFE • Kanban

> Web Development & Design: HTML5 / CSS3 • JavaScript •

JQuery • Angular • React / Redux • MEANStack / MERNStack • Node.JS • UX / UI • Bootstrap • XML / XSLT • Responsive Design • Python • Perl • PHP • XML

> Python: Python for Web • Python Networking / Sys Admin

• Python Data Science • Python for Security Pros

> Security: CyberSecurity • Secure Software Design • Secure

Coding • Secure Web Development • Database Security • OWASP • STIG • CCSK

> Mobile: Android • IOS • PhoneGap • Kotlin • Swift •

Mobile Application Testing • Secure Mobile Development

> Big Data / Data Science, AI , Machine Learning, & Deep

Learning: Machine Learning • Hadoop Admin • Hadoop Development • Pig / Hive / MapReduce • Scala • Spark • R for Data Science • Data Science • Python for Data Science • AI – Artificial Intelligence

> SOA: SOA • Architecture • Analysis • Design • Governance •

Implementation • SOA Security

> Databases: DB Design • DB Security • DB2 • SQL • PLSQL •

Oracle • MySQL • SQLServer • MongoDB • NoSQL • MariaDB • Cassandra • Oracle • JDBC

> MainFrame: C • COBOL • DB2 • Assembler • Transition to

Web • Java for COBOL

> O/S & SysAdmin: Windows • Linux • UNIX • Mainframe •

Z/OS • Administration • IOS

> Oracle: DBA • SQL • PL/SQL • New Features • OBIEE

> **Tools** • Eclipse • Ant / Maven • Oracle • Oracle BI • IntelliJ • NetBeans • Tableau • Selenium • Cucumber

> **DevOps**: DevOps • GIT • GitHub • Jenkins • JIRA • Docker

> **IT Skills**: Agile • Scrum • ITIL • Project Management • Leadership • Soft Skills • End User

> Software Architecture, Design & Engineering: Architecture

• Analysis • Requirements • Estimation • Use Cases • UML • OO • BDD • Data Modeling & Design • Software Design

> **Software Testing**: QA • Test Automation • Unit Testing • TDD • Selenium • Cucumber • Gherkin

Please visit www.triveratech.com for the complete course catalog.

Python 3 for Scientists

Version 1.0, October 2019

Table of Contents

About this course	1
Welcome!	2
Classroom etiquette	3
Course Outline	4
Student files	6
Extracting the student files	7
Examples	8
Lab Exercises	9
Appendices	10
Chapter 1: The Python Environment	11
Starting Python	12
If the interpreter is not in your PATH	13
Using the interpreter	15
Trying out a few commands	16
Running Python scripts (explicit)	17
Running Python scripts (implicit)	18
Using pydoc	21
Python Editors and IDEs	22
Chapter 2: Getting Started	25
Using variables	26
Keywords and Builtins	27
Variable typing	29
Strings	30
Single-delimited string literals	31
Triple-delimited string literals	32
Raw string literals	33
Unicode characters	34
String operators and methods	37
String Methods	39
Numeric literals	43
Math operators and expressions	45
Converting among types	48

Writing to the screen	48
String Formatting	51
Legacy String Formatting	53
Command line parameters	56
Reading from the keyboard	57
Chapter 3: Flow Control	61
About flow control	62
What's with the white space?	63
if and elif	64
Conditional Expressions	66
Relational Operators	67
Boolean operators	69
while loops	71
Alternate ways to exit a loop	72
Chapter 4: Array types	77
About Array Types	78
Lists	80
Indexing and slicing	83
Iterating through a sequence	87
Tuples	89
Iterable Unpacking	91
Nested sequences	92
Functions for all sequences	95
Using enumerate()	98
Operators and keywords for sequences	101
The range() function	104
List comprehensions	107
Generator Expressions	110
Chapter 5: Working with Files	117
Text file I/O	118
Opening a text file	119
The <i>with</i> block	120
Reading a text file	121
Writing to a text file	126
Chapter 6: Dictionaries	131

About dictionaries	132
When to use dictionaries?	133
Creating dictionaries	134
Getting dictionary values	138
Iterating through a dictionary	141
Reading file data into a dictionary	143
Counting with dictionaries	145
About sets	147
Creating Sets	148
Working with sets	149
Chapter 7: Functions, Modules and Packages	155
Functions	156
Function parameters	159
Default parameters	163
Python Function parameter behavior (from PEP 3102)	165
Name resolution (AKA Scope)	166
The global statement	169
Modules	170
Using import	171
How <i>import *</i> can be dangerous	175
Module search path	177
Executing modules as scripts	178
Packages	180
Configuring import with <code>__init__.py</code>	182
Documenting modules and packages	185
Python style	186
Chapter 8: Errors and Exception Handling	189
Syntax errors	190
Exceptions	191
Handling exceptions with try	192
Handling multiple exceptions	193
Handling generic exceptions	194
Ignoring exceptions	195
Using else	196
Cleaning up with finally	198

Chapter 9: Using the Standard Library	205
The sys module	206
Interpreter Information	206
STDIO	207
Launching external programs	208
Paths, directories and filenames	210
Walking directory trees	214
Grabbing data from the web	217
Sending email	220
math functions	226
Random values	227
Dates and times	230
Zipped archives	233
Chapter 10: Pythonic Programming	237
The Zen of Python	238
Tuples	239
Iterable unpacking	240
Unpacking function arguments	242
The sorted() function	246
Custom sort keys	247
Lambda functions	252
List comprehensions	254
Dictionary comprehensions	256
Set comprehensions	258
Iterables	259
Generator Expressions	261
Generator functions	263
String formatting	266
f-strings	268
Chapter 11: An Introduction to Python Classes	273
About O-O programming	274
Defining classes	275
Constructors	277
Instance methods	278
Properties	281

Class methods and data	285
Static Methods	287
Private methods	288
Inheritance	289
Untangling the nomenclature	292
Chapter 12: Developer Tools	295
Objectives	295
Program development	296
Comments	297
pylint	298
Customizing pylint	299
Using pyreverse	300
The Python debugger	302
Starting debug mode	303
Stepping through a program	304
Setting breakpoints	305
Profiling	306
Benchmarking	308
Chapter 13: Excel Spreadsheets	313
Objectives	313
The openpyxl module	314
Reading an existing spreadsheet	315
Working with ranges	318
Worksheet info	320
Modifying a worksheet	321
Working with formulas	323
Creating a new spreadsheet	325
Setting Styles	326
Chapter 14: Serializing Data	331
Objectives	331
About XML	332
Normal Approaches to XML	333
Which module to use?	334
Getting Started With ElementTree	335
How ElementTree Works	336

Elements	337
Creating a New XML Document	340
Parsing An XML Document	343
Navigating the XML Document	344
Using XPath	348
About JSON	352
Reading JSON	353
Writing JSON	356
Customizing JSON	359
Reading and writing YAML	363
Reading CSV data	369
Nonstandard CSV	371
Using csv.DictReader	373
Writing CSV Data	375
Pickle	377
Chapter 15: iPython and Jupyter	381
About iPython	382
Features of iPython	383
The many faces of iPython	384
Starting iPython	385
Getting Help	386
Tab Completion	387
Magic Commands	388
Benchmarking	389
External commands	390
Enhanced help	391
Jupyter notebooks	392
Jupyter Demo	392
Chapter 16: Introduction to numpy	393
Python's scientific stack	394
NumPy overview	395
Creating Arrays	396
Creating ranges	400
Working with arrays	403
Shapes	406

Slicing and indexing	410
Indexing with Booleans	413
Stacking	417
Iterating	419
Array creation shortcuts	421
Matrices	424
Data Types	428
Numpy Example List with Doc	430
Chapter 17: Introduction to SciPy	433
About scipy	434
Polynomials	435
Vectorizing functions	439
Working with SciPy	442
SciPy Subpackages	445
Getting Help	446
Tour of SciPy subpackages	448
SciPy in action	448
Chapter 18: Introduction to Pandas	449
About pandas	450
Pandas architecture	451
Series	452
DataFrames	458
Data alignment	463
Index objects	467
Basic Indexing	471
Broadcasting	475
Removing entries	479
Time Series	481
Reading Data	487
Chapter 19: Introduction to Matplotlib	493
About matplotlib	494
matplotlib architecture	495
Matplotlib Terminology	496
Matplotlib Keeps State	497
What Else Can You Do?	498

Appendix A: Where do I go from here?	501
Resources for learning Python	501
Appendix B: Python Bibliography	503
Index	509

About this course

Welcome!

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

{django-version} {django-version} {django-version}

Instructor name:

Instructor e-mail:



Have Fun!

Classroom etiquette

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

Please turn off cell phone ringers and other noisemakers.

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

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

IMPORTANT

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

Course Outline

Day 1

Chapter 1 The Python Environment

Chapter 2 Getting started

Chapter 3 Flow control

Chapter 4 Array types

Day 2

Chapter 5 Working with files

Chapter 6 Dictionaries and sets

Chapter 7 Functions, modules, and packages

Chapter 8 Errors and exception handling

Day 3

Chapter 9 Using the standard library

Chapter 10 Pythonic programming

Chapter 11 Intermediate classes

Chapter 12 Developer tools

Day 4

Chapter 13 Serializing data

Chapter 14 Multiprogramming

Chapter 15 Excel spreadsheets

Chapter 16 iPython and Jupyter

Day 5

Chapter 17 NumPy

Chapter 18 SciPy

Chapter 19 Pandas

Chapter 20 Matplotlib

Ad hoc topics

- Dependency management with conda
- Type checking

- Cython and performance management

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **py3forsci**.

What's in the files?

py3forsci contains data and other files needed for the exercises

py3forsci/EXAMPLES/ contains the examples from the course manuals.

py3forsci/ANSWERS/ contains sample answers to the labs.

py3forsci/DATA/ contains data used in examples and answers

py3forsci/SETUP/ contains any needed setup scripts (may be empty)

py3forsci/TEMP/ initially empty; used by some examples for output files

py3forsci/NOTEBOOKS/ Jupyter notebooks for use in class

NOTE

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

Extracting the student files

Windows

Open the file **py3forsci.zip**. Extract all files to your desktop. This will create the folder **py3forsci**.

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

Copy or download py3forsci.tgz to your home directory. In your home directory, type

```
tar xzvf py3forsci.tgz
```

This will create the **py3forsci** directory under your home directory.

Examples

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

It will look like this:

Example

cmd_line_args.py

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

import sys ①

print(sys.argv) ②

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

- ① Import the **sys** module
- ② Print all parameters, including script itself
- ③ Get the first actual parameter

cmd_line_args.py apple mango 123

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

Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in `py3forsci/ANSWERS`

Appendices

- Appendix A: Python Bibliography

Chapter 1: The Python Environment

Objectives

- Using the interpreter
- Getting help
- Running scripts on Windows, Linux, and Mac
- Learning best editors and IDEs

I think the real key to Python's platform independence is that it was conceived right from the start as only very loosely tied to Unix.

— Guido van Rossum

Starting Python

- Type **python** (or **python3**) at a command prompt
- **python** should be in your PATH
- If python was not found, install it or add directory to PATH

To start the Python 3 interpreter, just type **python** (or **python3**) at the command prompt. If you get an error message, one of two things has happened:

- Python 3 is not installed on your computer
- The Python 3 interpreter is not in your PATH variable

If Python is not installed, the solution is evident.

If the interpreter is not in your PATH

If the directory where the interpreter lives is not in your PATH variable, you have several choices.

Type the full path

Start **python** by typing the full path to the interpreter (e.g., C:\python35\python or /usr/bin/python)

Add the directory to PATH temporarily

Windows (at a command prompt)

```
set PATH="%PATH%";c:\python35
```

non-Windows

```
PATH="$PATH:/usr/dev/bin"  sh,ksh,bash  
setenv PATH "$PATH:/usr/dev/bin"  csh,tcsh
```

Add the directory to PATH permanently

Windows

Right-click on the **My Computer** icon. Select **Properties**, and then select the **Advanced** tab. Click on the **Environment Variables** button, then double-click on **PATH** in the **System Variables** area at the bottom of the dialog. Add the directory for the Python interpreter to the existing value and click OK. Be sure to separate the path from existing text with a semicolon.

non-Windows

Add a line to your shell startup file (e.g. `.bash_profile`, `.profile`, etc.) to add the directory containing the Python interpreter to your PATH variable .

The command should look something like

```
PATH="$PATH:/path/to/python"
```

Using the interpreter

- Type any Python statement or expression
- Prompt is >>>
- Command line editing supported
- Ctrl-D (Unix) or Ctrl-Z <Enter> (Windows) to exit

Once you have started the Python interpreter, it provides an interactive interpreter. The prompt is ">>>". You can type in any Python commands at this prompt.

For Windows, it supports the editing keys on a standard keyboard, which include Home, End, etc., as well as the arrow keys. Normal PC shortcuts such as Ctrl-RightArrow to jump to the next word also work.

For other systems, Python supports **GNU readline** editing, which uses emacs-style commands. These commands are detailed in the table below.

On all versions, you can use arrow keys and backspace to edit the line.

As of version 3.4, the interpreter does autocomplete when you press the TAB key

Table 1. emacs-style command line editing

Emacs-mode Command	Function
^P	Previous command
^N	Next command
^F	Forward 1 character
^B	Back 1 character
^A	Beginning of line
^E	End of line
^D	Delete character under cursor
^K	Delete to end of line

Trying out a few commands

Try out the following commands in the interpreter:

```
>>> print("Hello, world")
Hello, world
>>> print(4 + 3)
7
>>> print(10/3)
3.3333333333333335
>>>
```

You don't really need **print()**

```
>>> "Hello, world"
'Hello, world'
>>> 4 + 3
7
>>>
```

When you press <Enter>, the interpreter evaluates and prints out whatever you typed in.

Running Python scripts (explicit)

- Use Python interpreter
- Same for any OS

To run a Python script (a file with the extension **.py**, merely call the Python interpreter with the script as its argument:

```
python __myscript.py__
```

This *explicit* execution will work on any operating system.

NOTE

If you are sure that Python is installed, and the above technique does not work, it might be because the python interpreter is not in your path. See the earlier discussion about adding python to your path.

Running Python scripts (implicit)

- Use extension `.py`
- Launch script from command line or file browser
- On non-Windows, make script executable

Windows

Running a Python script implicitly on Windows is simple. Give the script the extension `.py`, and type its name (or double-click it from a file browser).

The script doesn't need any special permissions.

TIP

Implicit execution works because the `.py` extension is associated with the Python interpreter. When some applications are installed, they "take over" the extension, so when you type `spam.py`, it might open the file in an editor or IDE.

Non-Windows

- End scripts with '.py'
- Add `#!` line
- Add execute permission
- run with `script.py` or `./script.py`

To execute a Python script implicitly (without typing **python** in front of it) on a non-Windows OS (Linux, OS X, etc.), there are two steps:

First, add a line that starts with `#!` (called shebang) as the first line in the file, and specify the Python interpreter. It is usually best to use `/usr/bin/env` to locate the interpreter, but you can also specify the actual path to an interpreter.

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

Second, set the permissions on the file to include execute, with a command such as

```
chmod 755 __scriptname__.py
```

or

```
chmod +x __scriptname__.py
```

Then, just type the name of the script. If the current directory is not included in the **PATH** variable, prefix the script name with `./` (dot-slash).

Example

```
vi hello  
chmod 755 hello.py
```

If `.` is in PATH

```
hello.py
```

If `.` is not in PATH

```
./hello.py
```

You can also specify the actual path to a Python interpreter:

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

However, this is not as portable as the `/usr/bin/env` solution above.

Using pydoc

From the Python interpreter

Type

```
>>> help(thing)
```

Where *thing* can be either the name (in quotes) of a function, module or package, or the actual (imported) function, module, or package object.

```
>>>help(len)
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

From a command line (Unix, Windows, or OS X)

Use `pydoc <name>` to display the documentation for `<name>`. `<name>` may be the name of a function, module, package, or method or attribute of an object. If the argument contains a slash (back- or forward-), it is taken as the path to a Python source file.

```
$ pydoc len
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
```

TIP

Run `pydoc -k <keyword>` to search packages by keyword

Python Editors and IDEs

- Editor is programmer's most-used tool
- Select Python-aware editor or IDE
- Many open source and commercial choices

There are two pages on the Python Wiki that discuss editors and IDEs:

```
http://wiki.python.org/moin/PythonEditors  
http://wiki.python.org/moin/IntegratedDevelopmentEnvironments
```

PyCharm Community Edition is the most full-featured free IDE available. Other good multi-platform IDEs include Spyder, Eclipse, Visual Studio Code, and Sublime Edit. These work on Windows, Unix/Linux, and Mac platforms and probably some others.

Chapter 1 Exercises

Exercise 1-1 (hello.py)

Using any editor, write a "Hello, world" python script.

Run the script explicitly

Run the script implicitly

Open the script in your IDE and run it from there.

TIP

In PyCharm, you can right-click (Ctrl-click on Mac) the script's tab and select **Run**

Chapter 2: Getting Started

Objectives

- Using variables
- Understanding dynamic typing
- Working with text
- Working with numbers
- Writing output to the screen
- Getting command line parameters
- Reading keyboard input

Using variables

- Variables are created when assigned to
- May hold any type of data
- Names are case sensitive
- Names may be any length

Variables in Python are created by assigning a value to them. They are created and destroyed as needed by the interpreter. Variables may hold any type of data, including string, numeric, or Boolean. The data type is dynamically determined by the type of data assigned.

Variable names are case sensitive, and may be any length. **Spam SPAM** and **spam** are three different variables.

A variable *must* be assigned a value. A value of **None** (null) may be assigned if no particular value is needed. It is good practice to make variable names consistent. The Python style guide ([Pep 8](https://www.python.org/dev/peps/pep-0008/) [https://www.python.org/dev/peps/pep-0008/]) suggests:

```
all_lower_case_with_underscores
```

Example

```
quantity = 5
historian = "AJP Taylor"
final_result = 123.456
program_status = None
```

Keywords and Builtins

- Keywords are reserved
- Using a keyword as a variable is a syntax error
- 72 builtin functions
- Builtins *may* be overwritten (but it's not a big deal)

Python keywords may not be used as names. You cannot say `class = 'Sophomore'`.

On the other hand, any of Python's 72 builtin functions, such as `len()` or `int()` may be used as identifiers, but that will overwrite the builtin's functionality, so you shouldn't do that.

TIP

Be especially careful not to use **dir**, **file**, **id**, **len**, **max**, **min**, and **sum** as variable names, as these are all builtin function names.

Python 3 Keywords

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

Table 2. Builtin functions

<code>abs()</code>	<code>float()</code> *	<code>object()</code> *
<code>all()</code>	<code>format()</code>	<code>oct()</code>
<code>any()</code>	<code>frozenset()</code> *	<code>open()</code>
<code>ascii()</code>	<code>getattr()</code>	<code>ord()</code>
<code>bin()</code>	<code>globals()</code>	<code>pow()</code>
<code>bool()</code> *	<code>hasattr()</code>	<code>print()</code>
<code>bytearray()</code> *	<code>hash()</code>	<code>property()</code> *
<code>bytes()</code> *	<code>help()</code>	<code>quit()</code>
<code>callable()</code>	<code>hex()</code>	<code>range()</code> *
<code>chr()</code>	<code>id()</code>	<code>repr()</code>
<code>classmethod()</code> *	<code>input()</code>	<code>reversed()</code> *
<code>compile()</code>	<code>int()</code> *	<code>round()</code>
<code>complex()</code> *	<code>isinstance()</code>	<code>set()</code> *
<code>copyright()</code>	<code>issubclass()</code>	<code>setattr()</code>
<code>credits()</code>	<code>iter()</code>	<code>slice()</code> *
<code>delattr()</code>	<code>len()</code>	<code>sorted()</code>
<code>dict()</code> *	<code>license()</code>	<code>staticmethod()</code> *
<code>dir()</code>	<code>list()</code> *	<code>str()</code> *
<code>divmod()</code>	<code>locals()</code>	<code>sum()</code>
<code>enumerate()</code> *	<code>map()</code> *	<code>super()</code> *
<code>eval()</code>	<code>max()</code>	<code>tuple()</code> *
<code>exec()</code>	<code>memoryview()</code> *	<code>type()</code> *
<code>exit()</code>	<code>min()</code>	<code>vars()</code>
<code>filter()</code> *	<code>next()</code>	<code>zip()</code> *

*These functions are class constructors

Variable typing

- Python is strongly and dynamically typed
- Type based on assigned value

Python is a strongly typed language. That means that whenever you assign a value to a name, it is given a *type*. Python has many types built into the interpreter, such as **int**, **str**, and **float**. There are also many packages providing types, such as **date**, **re**, or **urllib**.

Certain operations are only valid with the appropriate types.

WARNING

Python does not automatically convert strings to numbers or numbers to strings.

Strings

- All strings are Unicode
- String literals
 - Single-delimited (single-line only)
 - Triple-delimited (can be multi-line)
- Use single-quote or double-quote symbols
- Backslashes introduce *escape sequences*
- Strings can be raw (escape sequences not interpreted)

All python strings are Unicode strings. They can be initialized with several types of string literals. Strings support escape characters, such as `\t` and `\n`, for non-printable characters.

Single-delimited string literals

- Enclosed in pair of single or double quotes
- May not contain embedded newlines
- Backslash is treated specially.

Single-delimited strings are enclosed in a pair of single or double quotes.

Escape codes, which start with a backslash, are interpreted specially. This makes it possible to include control characters such as tab and newline in a string.

Single-delimited strings may not contain an embedded newline; that is, they may not be spread over multiple physical lines. They may contain `\n`, the escape code for a new line.

There is no difference in meaning between single and double quotes. The term "single-quoted" in the Python documentation means that there is one quote symbol at each end of the sting literal.

TIP | [Adjacent string literals are concatenated.](#)

Example

```
name = "John Smith"
title = 'Grand Poobah'
color = "red"
size = "large"
poem = "I think that I will never see\na poem lovely as a tree"
```

Triple-delimited string literals

- Used for multi-line strings
- Can have embedded quote characters
- Used for docstrings

Triple-delimited strings use three double or single quotes at each end of the text. They are the same as single-delimited strings, except that individual single or double quotes are left alone, and that embedded newlines are preserved.

Triple-delimited text is used for text containing literal quotes as well as documentation and boiler-plate text.

Example

```
name = """James Earl "Jimmy" Carter"""
warning = """
Professional driver on closed course
Do not attempt
Your mileage may vary
Ask your doctor if Python is right for you
"""

query = '''
from contacts
where zipcode = '90210'
order by lname
'''
```

NOTE

The quotes on both ends of the text must match – use either all single or all double quotes, whether it's a normal or a triple-delimited literal.

Raw string literals

- Start with **r**
- Do not interpret backslashes

If a literal starts with **r** before the quote marks, then it is a raw string literal. Backslashes are not interpreted.

This is handy if the text to be output contains literal backslashes, such as many regular expression patterns, or Windows path names.

Example

```
pat = r"\w+\s+\w+"  
loc = r"c:\temp"  
msg = r"please put a newline character (\n) after each line"
```

This is similar to the use of single quotes in some other languages.

Unicode characters

- Use `\uXXXX` to specify non-ASCII Unicode characters
- `XXXX` is Unicode value in hex
- `\N{NAME}` also OK

Unicode characters may be embedded in literal strings. Use the Unicode value for the character in the form `\uXXXX`, where `XXXX` is the hex version of the character's code point.

You can also specify the Unicode character name using the syntax `\N{name}`.

For code points above `FFFF`, use `\UXXXXXXXX` (note capital "U").

Raw strings accept the `\u` or `\U` notation, but do not accept `\N{}`.

See <http://www.unicode.org/charts> for lists of Unicode character names

Example

unicode.py

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

print('26\u00B0') ①
print('26\N{DEGREE SIGN}') ②
print(r'26\u00B0\n') ③
print()

print('we spent \u20ac1.23M for an original C\u00e9zanne') ④
print("Romance in F\u266F Major")
print()

data = ['\U0001F95A', '\U0001F414'] ⑤
print(sorted(data))
```

- ① Use \uXXXX where XXXX is the Unicode value in hex
- ② The Unicode entity name can be used, enclosed in \N{}
- ③ \N{} is not expanded in raw strings
- ④ More examples.
- ⑤ Python answers the age-old question.

unicode.py

```
26°
26°
26\u00B0\n

we spent €1.23M for an original Cézanne
Romance in F   Major

[' ', ' ', ' ']
```

Table 3. *Escape Sequences*

Sequence	Description
<code>\newline</code>	Embedded newline
<code>\\</code>	Backslash
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\a</code>	BEL
<code>\b</code>	BACKSPACE
<code>\f</code>	FORMFEED
<code>\n</code>	LINEFEED
<code>\N{name}</code>	Unicode named code point <i>name</i>
<code>\r</code>	Carriage Return
<code>\t</code>	TAB
<code>\uxxxx</code>	16-bit Unicode code point
<code>\Uxxxxxxxx</code>	32-bit Unicode code point
<code>\ooo</code>	Char with octal ASCII value <i>ooo</i>
<code>\xhh</code>	Character with hex ASCII value <i>hh</i>

String operators and methods

- Methods called from string objects
- Some builtin functions apply to strings
- Strings cannot be modified in-place
- Modified copies of strings are returned

Python has a rich set of operators and methods for manipulating strings.

Methods are called from string objects (variables) using "dot notation" – `STR.method()`. Some builtin functions are not called from strings, such as **len()**.

Strings are *immutable* – they can not be changed (modified in-place). Many string functions return a modified copy of the string.

Use `+` (plus) to concatenate two strings.

String methods may be chained. That is, you can call a string method on the string returned by another method.

If you need a substring function, that is provided by the **slice** operation in the **Array Types** chapter.

String methods may be called on literal strings as well

```
s = 'Barney Rubble'
print(s.upper())
print(s.count('b'))
print(s.lower().count('b'))

print(",".join(some_list))
print("abc".upper())
```

Example

strings.py

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

a = "My hovercraft is full of EELS"

print("original:", a)
print("upper:", a.upper())
print("lower:", a.lower())
print("swapcase:", a.swapcase()) ①
print("title:", a.title()) ②
print("e count (normal):", a.count('e'))
print("e count (lower-case):", a.lower().count('e')) ③
print("found EELS at:", a.find('EELS'))
print("found WOLVERINES at:", a.find('WOLVERINES')) ④

b = "graham"
print("Capitalized:", b.capitalize()) ⑤
```

- ① Swap upper and lower case
- ② All words are capitalized
- ③ Methods can be chained. The next method is called on the object returned by the previous method.
- ④ Returns -1 if substring not found
- ⑤ Capitalizes first character of string, only if it is a letter

strings.py

```
original: My hovercraft is full of EELS
upper: MY HOVERCRAFT IS FULL OF EELS
lower: my hovercraft is full of eels
swapcase: mY HOVERCRAFT IS FULL OF eels
title: My Hovercraft Is Full Of Eels
e count (normal): 1
e count (lower-case): 3
found EELS at: 25
found WOLVERINES at: -1
Capitalized: Graham
```

String Methods

Table 4. string methods

Method	Description
<code>S.capitalize()</code>	Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case.
<code>S.casefold()</code>	Return a version of S suitable for caseless comparisons.
<code>S.center(width[, fillchar])</code>	Return S centered in a string of length width. Padding is done using the specified fill character (default is a space)
<code>S.count(sub, [, start[, end]])</code>	Return the number of non-overlapping occurrences of substring sub. Optional arguments start and end specify a substring to search.
<code>S.encode(encoding='utf-8', errors='strict')</code>	Encode S using the codec registered for encoding. Default encoding is 'utf-8'. errors may be given to set a different error handling scheme. Default is 'strict' meaning that encoding errors raise a <code>UnicodeEncodeError</code> . Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with <code>codecs.register_error</code> that can handle <code>UnicodeEncodeErrors</code> .
<code>S.endswith(suffix[, start[, end]])</code>	Return True if S ends with the specified suffix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. suffix can also be a tuple of strings to try.
<code>S.expandtabs(tabsize=8)</code>	Return a copy of S where all tab characters are expanded using spaces. If tabsize is not given, a tab size of 8 characters is assumed.
<code>S.find(sub[, start[, end]])</code>	Return the lowest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation. Returns -1 on failure.
<code>S.format(*args, **kwargs)</code>	Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').
<code>S.format_map(mapping)</code>	Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}').
<code>S.index(sub[, start[, end]])</code>	Like <code>find()</code> but raise <code>ValueError</code> when the substring is not found.
<code>S.isalnum()</code>	Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

Method	Description
<code>S.isalpha()</code>	Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.
<code>S.isdecimal()</code>	Return True if there are only decimal characters in S, False otherwise.
<code>S.isdigit()</code>	Return True if all characters in S are digits and there is at least one character in S, False otherwise.
<code>S.isidentifier()</code>	Return True if S is a valid identifier according to the language definition.
<code>S.islower()</code>	Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.
<code>S.isnumeric()</code>	Return True if there are only numeric characters in S, False otherwise.
<code>S.isprintable()</code>	Return True if all characters in S are considered printable in <code>repr()</code> or S is empty, False otherwise.
<code>S.isspace()</code>	Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.
<code>S.istitle()</code>	Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.
<code>S.isupper()</code>	Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.
<code>S.join(iterable)</code>	Return a string which is the concatenation of the strings in the iterable. The separator between elements is the string from which <code>join()</code> is called
<code>S.ljust(width[, fillchar])</code>	Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).
<code>S.lower()</code>	Return a copy of the string S converted to lowercase.
<code>S.lstrip([chars])</code>	Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead.

Method	Description
<code>S.partition(sep)</code>	Search for the separator <code>sep</code> in <code>S</code> , and return the part before it, the separator itself, and the part after it. If the separator is not found, return <code>S</code> and two empty strings.
<code>S.replace(old, new[, count])</code>	Return a copy of <code>S</code> with all occurrences of substring <code>old</code> replaced by <code>new</code> . If the optional argument <code>count</code> is given, only the first <code>count</code> occurrences are replaced.
<code>S.rfind(sub[, start[, end]])</code>	Return the highest index in <code>S</code> where substring <code>sub</code> is found, such that <code>sub</code> is contained within <code>S[start:end]</code> . Optional arguments <code>start</code> and <code>end</code> are interpreted as in slice notation. Return <code>-1</code> on failure.
<code>S.rindex(sub[, start[, end]])</code>	Like <code>rfind()</code> but raise <code>ValueError</code> when the substring is not found.
<code>S.rjust(width[, fillchar])</code>	Return <code>S</code> right-justified in a string of length <code>width</code> . Padding is done using the specified fill character (default is a space).
<code>S.rpartition(sep)</code>	Search for the separator <code>sep</code> in <code>S</code> , starting at the end of <code>S</code> , and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and
<code>S.rsplit(sep=None, maxsplit=-1)</code>	Return a list of the words in <code>S</code> , using <code>sep</code> as the delimiter string, starting at the end of the string and working to the front. If <code>maxsplit</code> is given, at most <code>maxsplit</code> splits are done. If <code>sep</code> is not specified, any whitespace string is a separator.
<code>S.rstrip([chars])</code>	Return a copy of the string <code>S</code> with trailing whitespace removed. If <code>chars</code> is given and not <code>None</code> , remove characters in <code>chars</code> instead.
<code>S.split(sep=None, maxsplit=-1)</code>	Return a list of the words in <code>S</code> , using <code>sep</code> as the delimiter string. If <code>maxsplit</code> is given, at most <code>maxsplit</code> splits are done. If <code>sep</code> is not specified or is <code>None</code> , any whitespace string is a separator and empty strings are removed from the result.
<code>S.splitlines([keepends])</code>	Return a list of the lines in <code>S</code> , breaking at line boundaries. Line breaks are not included in the resulting list unless <code>keepends</code> is given and true.
<code>S.startswith(prefix[, start[, end]])</code>	Return <code>True</code> if <code>S</code> starts with the specified prefix, <code>False</code> otherwise. With optional <code>start</code> , test <code>S</code> beginning at that position. With optional <code>end</code> , stop comparing <code>S</code> at that position. <code>prefix</code> can also be a tuple of strings to try.

Method	Description
<code>S.strip([chars])</code>	Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead.
<code>S.swapcase()</code>	Return a copy of S with uppercase characters converted to lowercase and vice versa.
<code>S.title()</code>	Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.
<code>S.translate(table)</code>	Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted.
<code>S.upper()</code>	Return a copy of S converted to uppercase.
<code>S.zfill(width)</code>	Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

Numeric literals

- Four kinds of numeric objects
 - Booleans
 - Integers
 - Floats
 - Complex numbers
- Integer literals can be decimal, octal, or hexadecimal
- Floating point can be traditional or scientific notation

Boolean

Boolean values can be 1 (true) or 0 (false). The keywords True and False can be used to represent these values, as well.

Integers

Integers can be specified as decimal, octal, or hexadecimal. Prefix the number with 0o for octal, 0x for hex, or 0b for binary. Integers are signed, and can be arbitrarily large.

Floats

Floating point integers may be specified in traditional format or in scientific notation.

Complex Numbers

Complex numbers may be specified by adding J to the end of the number.

Example

numeric.py

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

a = 5
b = 10
c = 20.22
d = 0o123      ①
e = 0xdeadbeef ②
f = 0b10011101 ③

print("a, b, c", a, b, c)
print("a + b", a + b)
print("a + c", a + c)
print("d", d)
print("e", e)
print("f", f)
```

① Octal

② Hex

③ Binary

numeric.py

```
a, b, c 5 10 20.22
a + b 15
a + c 25.22
d 83
e 3735928559
f 157
```


Math operators and expressions

- Many built-in operators and expressions
- Operations between integers and floats result in floats

Python has many math operators and functions. Later in this course we will look at some libraries with extended math functionality.

Most of the operators should look familiar; a few may not:

Division

Division (/) always returns a float result.

Assignment-with-operation

Python supports C-style assignment-with-operation. For instance, `x += 5` adds 5 to variable `x`. This works for nearly any operator in the format:

VARIABLE OP=VALUE e.g. `x += 1`

is equivalent to

VARIABLE = VARIABLE OP VALUE e.g. `x = x + 1`

Exponentiation

To raise a number to a power, use the `**` (exponentiation) operator or the `pow()` function.

Floored Division

Using the floored division operator `//`, the result is always rounded down to the nearest whole number.

Order of operations

Please Excuse My Drunk Aunt Sally!

Parentheses, Exponents, Multiplication or Division, Addition or Subtraction (but use parentheses for readability)

Example

math_operators.py

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

x = 22
x += 10 ①

y = 5
y *= 3 ①

print("x:", x)
print("y:", y)

print("2 ** 16", 2 ** 16)

print("x / y", x / y)
print("x // y", x // y) ②
```

① Same as $x = x + 1$, $y = y * 3$, etc.

② Returns floored result (rounded down to nearest whole number)

math_operators.py

```
x: 32
y: 15
2 ** 16 65536
x / y 2.1333333333333333
x // y 2
```

NOTE

Python does not have the `++` and `--` (post-increment and post-decrement) operators common to many languages derived from C.

Table 5. Python Math Operators and Functions

Operator or Function	What it does
$x + y$	sum of x and y
$x - y$	difference of x and y
$x * y$	product of x and y
x / y	quotient of x and y
$x // y$	(floored) quotient of x and y
$x \% y$	remainder of x / y
$-x$	x negated
$+x$	x unchanged
<code>abs(x)</code>	absolute value or magnitude of x
<code>int(x)</code>	x converted to integer
<code>float(x)</code>	x converted to floating point
<code>complex(re,im)</code>	a complex number with real part re, imaginary part im. im defaults to zero.
<code>c.conjugate()</code>	conjugate of the complex number c
<code>divmod(x, y)</code>	the pair $(x // y, x \% y)$
<code>pow(x, y)</code> $x ** y$	x raised to the power y

Converting among types

- No automatic conversion between numbers and strings
- Builtin functions
 - `int()` convert string or number to integer
 - `float()` convert string or number to float
 - `str()` convert anything to string
 - `bool()` convert anything to bool
 - `complex()` convert string or number to complex

Python is dynamically typed; if you assign a number to a variable, it will raise an error if you use it with a string operator or function; likewise, if you assign a string, you can't use it with numeric operators.

There are built-in functions to do these conversions. Use `int(s)` to convert string `s` to an integer. Use `str(n)` to convert anything to a string, and so forth.

There are functions to convert to all builtin types; the above four are the most often used.

If the string passed to `int()` or `float()` contains characters other than digits or minus sign, a runtime error is raised. Leading or trailing whitespace, however, are ignored. Thus " 123 " is OK, but "123ABC" is not.

Writing to the screen

- Use `print()` function
- Adds spaces between arguments (by default)
- Adds newline at end (by default)
- Use **`sep`** parameter for alternate separator
- Use **`end`** parameter for alternate ending

To output text to the screen, use the `print` function. It takes a list of one or more arguments, and writes them to the screen. By default, it puts a space between them and ends with a newline.

Two special named arguments can modify the default behavior. The `sep` parameter specifies

what is output between items, and *end* specifies what is written after all the arguments.

Example

print_examples.py

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

print("Hello, world")
print("#-----")

print("Hello,", end=' ') ①
print("world")
print("#-----")

print("Hello,", end=' ')
print("world", end='!') ②
print("#-----")

x = "Hello"
y = "world"

print(x, y) ③
print("#-----")

print(x, y, sep=', ') ④
print("#-----")

print(x, y, sep='') ⑤
print("#-----")
```

- ① Print space instead of newline at the end
- ② Print bang instead of newline at end
- ③ Item separator is space instead of comma
- ④ Item separator is comma + space
- ⑤ Item separator is empty string

print_examples.py

```
Hello, world
#-----
Hello, world
#-----
Hello, world!#-----
Hello world
#-----
Hello, world
#-----
Helloworld
#-----
```

String Formatting

- Use the `.format()` method
- Syntax: `"template".format(VALUE)`
- Placeholders: `{left_curly}Num:FlagsWidthType{right_curly}`

Strings have a `format()` method which allows variables and other objects to be embedded in strings and optionally formatted. Parameters to `format()` are numbered starting with 0, and are formatted by the correspondingly numbered placeholders in the string. However, if no numbers are specified, the placeholders will be auto-numbered from left to right, starting with 0. You cannot mix number and non-numbered placeholders in the same format string.

A placeholder looks like this: `{}` (for auto-numbering), or `{n}` (for manual numbering). To add formatting flags, follow the parameter number (if any) with a colon, then the type and other flags. You can also use named parameters, and specify the name rather than the parameter index.

Builtin types do not need to have the type specified, but you may specify the width of the formatted value, the number of decimal points, or other type-specific details.

For instance, `{0}` will use default formatting for the first parameter; `{2:04d}` will format the third parameter as an integer, padded with zeroes to four characters wide.

There are many more ways of using `format()`; this discussion describes some of the basics.

To include literal braces in the string, double them: `{{ }}`.

See [\[string_formatting\]](#) for details on formatting.

TIP

For even more information, check out the PyDoc topic **FORMATTING**, or section 6.1.3.1 [\[https://docs.python.org/3/library/string.html#format-specification-mini-language\]](https://docs.python.org/3/library/string.html#format-specification-mini-language) of The Python Standard Library documentation, the **Format Specification Mini-Language**.

NOTE

Python 3.6 added *f-strings*, which will further simplify embedding variables in strings. See [Pep 0498 \[https://www.python.org/dev/peps/pep-0498/\]](https://www.python.org/dev/peps/pep-0498/)

<<<

Example

string_formatting.py

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

name = "Tim"
count = 5
avg = 3.456
info = 2093

print("Name is [{:<10s}]".format(name)) ①
print("Name is [{:>10s}]".format(name)) ②
print("count is {0:3d} avg is {:.2f}".format(count, avg)) ③

print("info is {0} {0:d} {0:o} {0:x}".format(info)) ④
print("info is {0} {0:d} {0:#o} {0:#x}".format(info)) ⑤

print("${: ,d}".format(38293892)) ⑥

print("It is {temp} in {city}".format(city='Orlando', temp=85)) ⑦
```

① < means left justify (default for non-numbers), 10 is field width, s formats a string

② > means right justify

③ .2f means round a float to 2 decimal points

④ d is decimal, o is octal, x is hex

⑤ # means add 0x, 0o, etc.

⑥ , means add commas to numeric value

⑦ parameters can be selected by name instead of position

string_formatting.py

```
Name is [Tim      ]
Name is [        Tim]
count is 005 avg is 3.46
info is 2093 2093 4055 82d
info is 2093 2093 0o4055 0x82d
$38,293,892
It is 85 in Orlando
```


Legacy String Formatting

- Use the % operator
- Syntax: "template" % (VALUES)
- Similar to printf() in C

Prior to Python 2.6, the % operator was used for formatting. It returns a string that results from filling in a template string with placeholders in specified formats. :

```
%flagW.Ptype
```

where W is width, P is precision (max width or # decimal places)

The placeholders are similar to standard formatting, but are positional rather than numbered, and are specified with a percent sign, rather than braces.

If there is only one value to format, the value does not need parentheses.

WARNING

Legacy string formatting is deprecated as of Python 3.1, and will be removed in the future. It supports many of the same formatting features as the new style, but of course not all.

Table 6. Legacy formatting types

d,i	decimal integer
o	octal integer
u	unsigned decimal integer
x,X	hex integer (lower, UPPER case)
e,E	scientific notation (lower, UPPER case)
f,F	floating point
g,G	autochoose between e and f
c	character
r	string (using repr() method)
s	string (using str() method)
%	literal percent sign

Table 7. Legacy formatting flags

-	left justify (default is right justification)
#	use alternate format
0	left-pad number with zeros
+	precede number with + or -
(blank)	precede positive number with blank, negative with -

Example

string_formatting_legacy.py

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

name = "Tim"
count = 5
avg = 3.456
info = 2093

print("Name is [%-10s]" % name) ①
print("Name is [%10s]" % name) ②
print("count is %03d avg is %.2f" % (count, avg)) ③

print("info is %d %0 %x" % (info, info, info)) ④
print("info is %d %0 %x" % ((info,) * 3)) ⑤

print("info is %d %#0o %#x" % (info, info, info)) ⑥
```

- ① Dash means left justify string
- ② Right justify (default)
- ③ Argument to % is either a single variable or a tuple
- ④ Arguments must be repeated to be used more than once
- ⑤ Obscure way of doing the same thing Note: (x,) is singleton tuple
- ⑥ # means add 0x, 0o, etc.

string_formatting_legacy.py

```
Name is [Tim      ]
Name is [        Tim]
count is 005 avg is 3.46
info is 2093 4055 82d
info is 2093 4055 82d
info is 2093 0o4055o 0x82d
```

Command line parameters

- Use the **argv** list that is part of the sys module
- sys must be imported
- Element 0 is the script name itself

To get the command line parameters, use the list `sys.argv`. This requires importing the `sys` module. To access elements of this list, use square brackets and the element number. The first element (index 0) is the name of the script, so `sys.argv[1]` is the first argument to your script.

Example

`sys_argv.py`

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

import sys

print(sys.argv)
print()

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

① First command line parameter

`sys_argv.py` *Gawain*

```
['/Users/jstrick/curr/courses/python//examples3/sys_argv.py', 'Gawain']

name is Gawain
```

TIP

If you use an index for a non-existent parameter, an error will be raised and your script will exit. In later chapters you will learn how to check the size of a list, as well as how to trap the error.

Reading from the keyboard

- Use `input()`
- Provides a prompt string
- Use `int()` or `float()` to convert input to numeric values

To read a line from the keyboard, use `input()`. The parameter is a prompt string, and it returns the text that was entered. You can use `int()` or `float()` to convert the input to an integer or a floating-point number.

Example

`keyboard_input.py`

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

name = input("What is your name: ")
quest = input("What is your quest? ")
print(name, "seeks", quest)

raw_num = input("Enter number: ") ①
num = int(raw_num) ②

print("2 times", num, "is ", 2 * num)
```

① `input` is always a string

② convert to numbers as needed

`keyboard_input.py`

```
What is your name: Sir Lancelot
What is your quest? the Grail
Sir Lancelot seeks the Grail
Enter number: 5
2 times 5 is 10
```

TIP

If you use `int()` or `float()` to convert a string, a fatal error will be raised if the string contains any non-numeric characters or any embedded spaces. Leading and trailing spaces will be ignored.

Chapter 2 Exercises

Exercise 2-1 (c2f.py)

Write a Celsius to Fahrenheit converter. Your script should prompt the user for a Celsius temperature, then print out the Fahrenheit equivalent.

What the user types:

```
python c2f.py
```

(or run from PyCharm/Spyder etc)

The program prompts the user, and the user enters the temperature to be converted.

The formula is $F = ((9 * C) / 5) + 32$. Be sure to convert the user-entered value into a float.

Test your script with the following values: 100, 0, 37, -40

Exercise 2-2 (c2f_batch.py)

Create another C to F converter. This time, your script should take the Celsius temperature from the command line and output the Fahrenheit value. What you will type:

```
python c2f_batch.py 100
```

(or run from PyCharm/Spyder etc)

Test with the values from **c2f.py**.

These two programs should be identical, except for the input.

Exercise 2-3 (string_fun.py)

Write a script to prompt the user for a full name. Once the name is read in, do the following:

- Print out the name as-is
- Print the name in upper case
- Print the name in title case
- Print the number of occurrences of 'j'
- Print the length of the name
- Print the position (offset) of "jacob" in the string

Run the program, and enter "john jacob jingleheimer smith"

Chapter 3: Flow Control

Objectives

- Understanding how code blocks are delimited
- Implementing conditionals with the if statement
- Learning relational and Boolean operators
- Exiting a while loop before the condition is false

About flow control

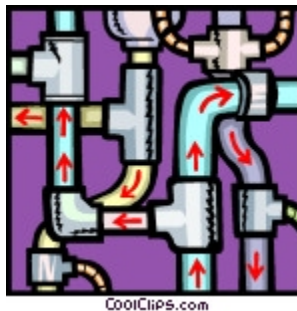
- Controls order of execution
- Conditionals and loops
- Uses Boolean logic

Flow control means being able to conditionally execute some lines of code, while skipping others, depending on input, or being able to repeat some lines of code.

In Python, the flow control statements are `if`, `while`, and `for`.

NOTE

Another kind of flow control is a function, which goes off to some other code, executes it, and returns to the current location. We'll cover functions in a later chapter.



What's with the white space?

- Blocks defined by indenting
- No braces or BEGIN-END keywords
- Enforces what good programmers do anyway
- Be consistent (suggested indent is 4 spaces)

One of the first things that most programmers learn about Python is that whitespace is significant. This might seem wrong to many; however, you will find that it was a great decision by Guido, because it enforces what programmers should be doing anyway.

It's very simple: After a line introducing a block structure (if statement, for/while loop, function definition, or class definition), all indented statements under the line are part of the block. Blocks may be nested, as in any language. The nested block has more indentation. A block ends when the interpreter sees a line with less indentation than the previous line.

Example

```
if value > 6:           start if statement
    print(value)        body of if

linecount = 0
for line in config:     start for loop
    if line.startswith("global"): start if (body of for)
        print(line)      body of if
    linecount += 1      back to body of for
```

TIP

Be consistent with indenting – use either all tabs or all spaces. Most editors can be set to your preference. (Guido suggests using 4 spaces).

if and elif

- The basic conditional statement is if
- Use else for alternatives
- elif provides nested if-else

The basic conditional statement in Python is `if expression:`. If the expression is true, then all statements in the block will be executed.

Example

```
if EXPR:
    statement
    statement
    ...
```

The expression does not require parentheses; only the colon at the end of the if statement is required.

In Python, a value is *false* if it is numeric zero, an empty container (string, list, tuple, dictionary, set, etc.), the builtin **False** object, or **None**. All other values are *true*.

The values **True** and **False** are predefined to have values of 1 and 0, respectively.

If an else statement is present, statements in the else block will be executed when the if statement is false.

For nested if-then, use the elif statement, which combines an if with an else. This is useful when the decision has more than two possibilities.

True and **False** are case-sensitive.

WARNING

Don't say

```
if x == True:
```

unless you really mean that x could only be 0 (False) or 1 (True). Just say

```
if x:
```

Conditional Expressions

- Used for simple if-then-else conditions

When you have a simple if-then-else condition, you can use the conditional expression. If the condition is true, the first expression is returned; otherwise the second expression is returned.

```
expr1 if condition else expr2
```

Example

```
print(long_message if DEBUGGING else short_message)

audience = 'j' if is_juvenile(curr_book_rec) else 'a'

mode = '>>' if APPEND_TO_LOG else '>'
```

Relational Operators

- Compare two objects
- Overloaded for different types of data
- Numbers are "less than" strings

== != < > >= <=

Python has six relational operators, implementing equality or greater than/less than comparisons. They can be used with most types of objects. All relational operators return **True** or **False**.

Example

if_else.py

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

raw_temp = input("Enter the temperature: ")
temp = int(raw_temp)

if temp < 76:
    print("Don't go swimming")

num = int(input("Enter a number: "))
if num > 1000000:
    print(num, "is a big number")
else:
    print("your number is", num)

raw_hour = input("Enter the hour: ")
hour = int(raw_hour)

if hour < 12:
    print("Good morning")
elif hour < 18: ①
    print("Good afternoon")
elif hour < 23:
    print("Good evening")
else:
    print("You're up late")
```

① **elif** is short for "else if", and always requires an expression to check

if_else.py

```
Enter the temperature: 50
Don't go swimming
Enter a number: 9999999
9999999 is a big number
Enter the hour: 8
Good morning
```


Boolean operators

- Combine Boolean values
- Can be used with any expressions
- Short-circuit
- Return last operand evaluated

The Boolean operators **and**, **or**, and **not** may be used to combine Boolean values. These do not need to be of type `bool` – the values will be converted as necessary.

These operators short-circuit; they only evaluate the right operand if it is needed to determine the value. In the expression **`a()` or `b()`**, if **`a()`** returns `True`, **`b()`** is not called.

The return values of Boolean operators are the last operand evaluated. **`4 and 5`** returns `5`. **`0 or 4`** returns `4`.

Table 8. Boolean Operators

Expression	Value
AND	
12 and 5	5
5 and 12	12
0 and 12	0
12 and 0	0
"" and 12	""
12 and ""	""
OR	
12 or 5	12
5 or 12	5
0 or 12	12
12 or 0	12
"" or 12	12
12 or ""	12

while loops

- Loop while some condition is **True**
- Used for getting input until user quits
- Used to create services (AKA daemons)

```
while EXPR:  
    statement  
    statement  
    ...
```

The **while** loop is used to execute code as long as some expression is true. Examples include reading input from the keyboard until the users signals they are done, or a network server looping forever with a **while True:** loop.

In Python, the **for** loop does much of the work done by a while loop in other languages.

NOTE | Unlike many languages, reading a file in Python generally uses a **for** loop.

Alternate ways to exit a loop

- **break** exits loop completely
- **continue** goes to next iteration

Sometimes it is convenient to exit a loop without regard to the loop expression. The **break** statement exits the smallest enclosing loop.

This is used when repeatedly requesting user input. The loop condition is set to **True**, and when the user enters a specified value, the break statement is executed.

Other times it is convenient to abandon the current iteration and go back to the top of the loop without further processing. For this, use the **continue** statement.

Example

`while_loop_examples.py`

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

print("Welcome to ticket sales\n")

while True: ①
    raw_quantity = input("Enter quantity to purchase (or q to quit): ")
    if raw_quantity == '':
        continue ②
    if raw_quantity.lower() == 'q':
        print("goodbye!")
        break ③
    if raw_quantity.isdigit():
        quantity = int(raw_quantity)
        print("sending {} ticket(s)".format(quantity))
    else:
        print("Please enter a number")
```

- ① Loop "forever"
- ② Skip rest of loop; start back at top
- ③ Exit loop

while_loop_examples.py

```
Welcome to ticket sales
```

```
Enter quantity to purchase (or q to quit): 4
```

```
sending 4 ticket(s)
```

```
Enter quantity to purchase (or q to quit): 1
```

```
sending 1 ticket(s)
```

```
Enter quantity to purchase (or q to quit): _
```

```
Enter quantity to purchase (or q to quit): _abc
```

```
Please enter a number
```

```
Enter quantity to purchase (or q to quit): q
```

```
goodbye!
```

Chapter 3 Exercises

Exercise 3-1 (c2f_loop.py)

Redo `c2f.py` to repeatedly prompt the user for a Celsius temperature to convert to Fahrenheit and then print. If the user just presses **Return**, go back to the top of the loop. Quit when the user enters "q".

TIP

read in the temperature, test for "q" or "", and only then convert the temperature to a float.#

Exercise 3-2 (guess.py)

Write a guessing game program. You will think of a number from 1 to 25, and the computer will guess until it figures out the number. Each time, the computer will ask "Is this your number? "; You will enter "l" for too low, "h" for too high, or "y" when the computer has got it. Print appropriate prompts and responses.

TIP

1. Start with `max_val = 26` and `min_val = 0`
2. `guess` is always `(max_val + min_val)//2` *Note integer division operator*
3. If current guess is too high, next guess should be halfway between lowest and current guess, and we know that the number is less than guess, so set `max_val = guess`
4. If current guess is too low, next guess should be halfway between current and maximum, and we know that the number is more than guess, so set `min_val = guess`

TIP

If you need more help, see next page for pseudocode. When you get it working for 1 to 25, try it for 1 to 1,000,000. (Set `max_value` to 1000001).

Exercise 3-3 (guessx.py)

Get the maximum number from the command line *or* prompt the user to input the maximum.

Pseudocode for guess.py

```
MAXVAL=26
MINVAL=0
while TRUE
    GUESS = int((MAXVAL + MINVAL)/2)
    prompt "Is your guess GUESS? "
    read ANSWER
    if ANSWER is "y"
        PRINT "I got it!"
        EXIT LOOP
    if ANSWER is "h"
        MAXVAL=GUESS
    if ANSWER is "l"
        MINVAL=GUESS
```


Chapter 4: Array types

Objectives

- Using single and multidimensional lists and tuples
- Indexing and slicing sequential types
- Looping over sequences
- Tracking indices with `enumerate()`
- Using `range()` to get numeric lists
- Transforming lists

About Array Types

- Array types
 - str
 - bytes
 - list
 - tuple
- Common properties of array types
 - Same syntax for indexing/slicing
 - Share some common methods and functions
 - All can be iterated over with a for loop

Python provides many data types for working with multiple values. Some of these are array types. These hold values in a sequence, such that they can be retrieved by a numerical index.

A str is an array of characters. A bytes object is array of bytes.

All array types may be indexed in the same way, retrieving a single item or a slice (multiple values) of the sequence.

Array types have some features in common with other container types, such as dictionaries and sets. These other container types will be covered in a later chapter.

All array types support iteration over their elements with a for loop.

Example

`typical_arrays.py`

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

fruits = ['apple', 'cherry', 'orange', 'kiwi', 'banana', 'pear', 'fig']
name = "Eric Idle"
knight = 'King', 'Arthur', 'Britain'

print(fruits[3]) ①
print(name[0]) ②
print(knight[1]) ③
```

`typical_arrays.py`

```
kiwi
E
Arthur
```

Lists

- Array of objects
- Create with []
- Add items with append(), extend(), or insert
- Remove items with del, pop(), or remove()

A list is one of the fundamental Python data types. Lists are used to store multiple values. The values may be similar – all numbers, all user names, and so forth; they may also be completely different. Due to the dynamic nature of Python, a list may hold values of any type, including other lists.

Create a list with a pair of square brackets. A list can be Initialized with a comma-separated list of values.

Table 9. List Methods

Method	Description
<code>del s[x]</code>	delete element x of s (keyword, not function)
<code>s.append(x)</code>	add single value x to end of s
<code>s.count(x)</code>	return count of elements whose value is x
<code>s.extend(x)</code>	add sequence x to end of s
<code>s.index(x)</code> <code>s.index(x, i)</code> <code>s.index(x, i, j)</code>	return index of first element whose value is x (after index i, before index j)
<code>s.insert(i, x)</code>	insert element x at offset i
<code>s.pop()</code> <code>s.pop(i)</code>	remove element at index i (default -1) from s and return it
<code>s.remove(x)</code>	remove first element whose value is x
<code>s.clear()</code>	remove all elements and leave the list empty
<code>s.reverse()</code>	reverses s in place
<code>s.sort()</code> <code>s.sort(key=func)</code>	sort s in place – func is function to derive key from one element

Example

creating_lists.py

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

list1 = list() ①
list2 = ['apple', 'banana', 'mango'] ②
list3 = [] ③
list4 = 'apple banana mango'.split() ④

print("list1:", list1)
print("list2:", list2)
print("list3:", list3)
print("list4:", list4)

print("list2[0]:", list2[0]) ⑤
print("list4[2]:", list4[2]) ⑥

print("list4[-1]:", list4[-1]) ⑦
```

- ① Create new empty list
- ② Initialize list
- ③ Create new empty list
- ④ Create list of strings with less typing
- ⑤ First element of **list2**
- ⑥ Third element of **list4**
- ⑦ *Last* element of **list4**

creating_lists.py

```
list1: []
list2: ['apple', 'banana', 'mango']
list3: []
list4: ['apple', 'banana', 'mango']
list2[0]: apple
list4[2]: mango
list4[-1]: mango
```

Indexing and slicing

- Use brackets for index
- Use slice for multiple values
- Same syntax for strings, lists, and tuples

Python is very flexible in selecting elements from a list. All selections are done by putting an index or a range of indices in square brackets after the list's name.

To get a single element, specify the index (0-based) of the element in square brackets:

```
foo = [ "apple", "banana", "cherry", "date", "elderberry",  
        "fig", "grape" ]  
  
foo[1]  the 2nd element of list foo -- banana
```

To get more than one element, use a slice, which specifies the beginning element (inclusive) and the ending element (exclusive):

```
foo[2:5]  foo[2], foo[3], foo[4] but NOT foo[5] – cherry, date, elderberry
```

If you omit the starting index of a slice, it defaults to 0:

```
foo[:5]  foo[0], foo[1], foo[2], foo[3], foo[4] – apple,banana,cherry, date,  
elderberry
```

If you omit the end element, it defaults to the length of the list.

```
foo[4:]  foo[4], foo[5], foo[6] – elderberry, fig, grape
```

A negative offset is subtracted from the length of the list, so -1 is the last element of the list, and -2 is the next-to-the-last element of the list, and so forth:

```
foo[-1] foo[len(foo)-1] or foo[6] - grape  
foo[-3] foo[len(foo)-3] or foo[4] - elderberry
```

The general syntax for a slice is

```
s[start:stop:step]
```

which means all elements $s[N]$, where

```
start <= N < stop,
```

and start is incremented by step

TIP | Remember that start is **IN**clusive but stop is **EX**clusive.

Example

indexing_and_slicing.py

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

pythons = ["Idle", "Cleese", "Chapman", "Gilliam", "Palin", "Jones"]

characters = "Roger", "Old Woman", "Prince Herbert", "Brother Maynard"

phrase = "She turned me into a newt"

print("pythons:", pythons)
print("pythons[0]", pythons[0]) ①
print("pythons[5]", pythons[5]) ②
print("pythons[0:3]", pythons[0:3]) ③
print("pythons[2:]", pythons[2:]) ④
print("pythons[:2]", pythons[:2]) ⑤
print("pythons[1:-1]", pythons[1:-1]) ⑥
print("pythons[0:2]", pythons[0:2]) ⑦
print("pythons[1:2]", pythons[1:2]) ⑧

pythons[3] = "Innes"
print("pythons:", pythons)
print()

print("characters", characters)
print("characters[2]", characters[2])
print("characters[1:]", characters[1:])

# characters[2] = "Patsy" # ERROR -- can't assign to tuple
print()
print("phrase", phrase)
print("phrase[0]", phrase[0])
print("phrase[-1]", phrase[-1]) ⑨
print("phrase[21:25]", phrase[21:25])
print("phrase[21:]", phrase[21:])
print("phrase[:10]", phrase[:10])
print("phrase[::2]", phrase[::2])
```

- ① First element
- ② Sixth element
- ③ First 3 elements
- ④ Third element through the end
- ⑤ First 2 elements
- ⑥ Second through next-to-last element
- ⑦ Every other element, starting with first
- ⑧ Every other element, starting with second
- ⑨ Last element

indexing_and_slicing.py

```

pythons: ['Idle', 'Cleese', 'Chapman', 'Gilliam', 'Palin', 'Jones']
pythons[0] Idle
pythons[5] Jones
pythons[0:3] ['Idle', 'Cleese', 'Chapman']
pythons[2:] ['Chapman', 'Gilliam', 'Palin', 'Jones']
pythons[:2] ['Idle', 'Cleese']
pythons[1:-1] ['Cleese', 'Chapman', 'Gilliam', 'Palin']
pythons[0::2] ['Idle', 'Chapman', 'Palin']
pythons[1::2] ['Cleese', 'Gilliam', 'Jones']
pythons: ['Idle', 'Cleese', 'Chapman', 'Innes', 'Palin', 'Jones']

characters ('Roger', 'Old Woman', 'Prince Herbert', 'Brother Maynard')
characters[2] Prince Herbert
characters[1:] ('Old Woman', 'Prince Herbert', 'Brother Maynard')

phrase She turned me into a newt
phrase[0] S
phrase[-1] t
phrase[21:25] newt
phrase[21:] newt
phrase[:10] She turned
phrase[::2] Setre eit et

```

Iterating through a sequence

- use a **for** loop
- works with lists, tuples, strings, or any other iterable
- Syntax `ifdef::highlight[]`

```
for var in iterable:  
    statement  
    statement  
    ...
```

To iterate through the values of a list, use the **for** statement. The variable takes on each value in the sequence, and keeps the value of the last item when the loop has finished.

To exit the loop early, use the `break` statement. To skip the remainder of an iteration, and return to the top of the loop, use the `continue` statement.

for loops can be used with any iterable object.

TIP

The loop variable retains the last value it was set to in the loop even after the loop is finished. (If the loop is in a function, the loop variable is local; otherwise, it is global).

Example

iterating_over_arrays.py

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

my_list = ["Idle", "Cleese", "Chapman", "Gilliam", "Palin", "Jones"]
my_tuple = "Roger", "Old Woman", "Prince Herbert", "Brother Maynard"
my_str = "She turned me into a newt"

for p in my_list: ①
    print(p)
print()

for r in my_tuple: ②
    print(r)
print()

for ch in my_str: ③
    print(ch, end=' ')
print()
```

- ① Iterate over elements of list
- ② Iterate over elements of tuple
- ③ Iterate over characters of string

iterating_over_arrays.py

```
Idle
Cleese
Chapman
Gilliam
Palin
Jones

Roger
Old Woman
Prince Herbert
Brother Maynard

S h e   t u r n e d   m e   i n t o   a   n e w   t
```

Tuples

- Designed for "records" or "structs"
- Immutable (read-only)
- Create with comma-separated list of objects
- Use for fixed-size collections of related objects
- Indexing, slicing, etc. are same as lists

Python has a second array type, the **tuple**. It is something like a list, but is immutable; that is, you cannot change values in a tuple after it has been created.

A tuple in Python is used for "records" or "structs" — collections of related items. You do not typically iterate over a tuple; it is more likely that you access elements individually, or *unpack* the tuple into variables.

Tuples are especially appropriate for functions that need to return multiple values; they can also be good for passing function arguments with multiple values.

While both tuples and lists can be used for any data, there are some conventions.

- Use a list when you have a collection of similar objects.
- Use a tuple when you have a collection of related, but dissimilar objects.

In a tuple, the position of elements is important; in a list, the position is not important.

For example, you might have a list of dates, where each date was contained in a month, day, year tuple.

To specify a one-element tuple, use a trailing comma; to specify an empty tuple, use empty parentheses.

```
result = 5,  
result = ()
```

TIP

Parentheses are not needed around a tuple unless the tuple is nested in a larger data structure.

Example

creating_tuples.py

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

birth_date = 1901, 5, 5

server_info = 'Linux', 'RHEL', 5.2, 'Melissa Jones'

latlon = 35.99, -72.390

print("birth_date:", birth_date)
print("server_info:", server_info)
print("latlon:", latlon)
```

creating_tuples.py

```
birth_date: (1901, 5, 5)
server_info: ('Linux', 'RHEL', 5.2, 'Melissa Jones')
latlon: (35.99, -72.39)
```

TIP

To specify a one-element tuple, use a trailing comma, otherwise it will be interpreted as a single object:

```
color = 'red',
```

Iterable Unpacking

- Copy elements to variables
- Works with any array-like object
- More readable than numeric indexing

If you have a tuple like this:

```
my_date = 8, 1, 2014
```

You can access the elements with

```
print(my_date[0], my_date[1], my_date[2])
```

It's not very readable though. How do you know which is the month and which is the day?

A better approach is *unpacking*, which is simply copying a tuple (or any other iterable) to a list of variables:

```
month, day, year = my_date
```

Now you can use the variables and anyone reading the code will know what they mean. This is really how tuples were designed to be used.

Nested sequences

- Lists and tuples may contain other lists and tuples
- Use multiple brackets to specify higher dimensions
- Depth of nesting limited only by memory

Lists and tuples can contain any type of data, so a two-dimensional array can be created using a list of lists. A typical real-life scenario consists of reading data into a list of tuples.

There are many combinations – lists of tuples, lists of lists, etc.

To initialize a nested data structure, use nested brackets and parentheses, as needed.

Example

nested_sequences.py

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

people = [
    ('Melinda', 'Gates', 'Gates Foundation'),
    ('Steve', 'Jobs', 'Apple'),
    ('Larry', 'Wall', 'Perl'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Bill', 'Gates', 'Microsoft'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linus', 'Torvalds', 'Linux'),
]

for person in people: ①
    print(person[0], person[1])
print('-' * 60)

for person in people:
    first_name, last_name, product = person ②
    print(first_name, last_name)
print('-' * 60)

for first_name, last_name, product in people: ③
    print(first_name, last_name)
print('-' * 60)
```

① person is a tuple

② unpack person into variables

③ if there is more than one variable in a for loop, each element is unpacked

nested_sequences.py

```
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

```
-----
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

```
-----
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

Functions for all sequences

- Many builtin functions expect a sequence
- Syntax

```
n = len(s)
n = min(s)
n = max(s)
n = sum(s)
s2 = sorted(s)
s2 = reversed(s)
s = zip(s1,s2,...)
```

Many builtin functions accept a sequence as the parameter. These functions can be applied to a list, tuple, dictionary, or set.

len(s) returns the number of elements in s (the number of characters in a string).

min(s) and **max(s)** return the smallest and largest values in s. Types in s must be similar — mixing strings and numbers will raise an error.

sorted(s) returns a sorted list of any sequence s.

NOTE

`min()`, `max()`, and `sorted()` accept a named parameter **key**, which specifies a key function for converting each element of s to the value wanted for comparison. In other words, the key function could convert all strings to lower case, or provide one property of an object.

sum(s) returns the sum of all elements of s, which must all be numeric.

reversed(s) returns an iterator (not a list) that can loop through s in reverse order.

zip(s1,s2,...) returns an iterator consisting of (s1[0],s2[0]),(s1[1], s2[1]), ...). This can be used to "pivot" rows and columns of data.

Example

sequence_functions.py

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

colors = ["red", "blue", "green", "yellow", "brown", "black"]
months = (
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
)

print("colors: len is {}; min is {}; max is {}".format(len(colors), min(colors),
max(colors)))
print("months: len is {}; min is {}; max is {}".format(len(months), min(months),
max(months)))
print()

print("sorted:", end=' ')
for m in sorted(colors): ①
    print(m, end=' ')
print()

phrase = ('dog', 'bites', 'man')
print(" ".join(reversed(phrase))) ②
print()

first_names = "Bill Bill Dennis Steve Larry".split()
last_names = "Gates Joy Richie Jobs Ellison".split()

full_names = zip(first_names, last_names) ③
print("full_names:", full_names)
print()

for first_name, last_name in full_names:
    print("{} {}".format(first_name, last_name))
```

① sorted() returns a sorted list

② reversed() returns a **reversed** iterator

③ zip() returns an iterator of tuples created from corresponding elements

sequence_functions.py

```
colors: len is 6; min is black; max is yellow  
months: len is 12; min is Apr; max is Sep
```

```
sorted: black blue brown green red yellow  
man bites dog
```

```
full_names: <zip object at 0x10e42e148>
```

```
Bill Gates  
Bill Joy  
Dennis Richie  
Steve Jobs  
Larry Ellison
```

Using enumerate()

- Numbers items beginning with 0 (or specified value)
- Returns enumerate object that provides a *virtual* list of tuples

To get the index of each list item, use the builtin function `enumerate(s)`. It returns an **enumerate object**.

```
for t in enumerate(s):  
    print(t[0],t[1])  
  
for i,item in enumerate(s):  
    print(i,item)  
  
for i,item in enumerate(s,1)  
    print(i,item)
```

When you iterate through the following list with `enumerate()`:

```
[x,y,z]
```

you get this (virtual) list of tuples:

```
[(0,x),(1,y),(2,z)]
```

You can give `enumerate()` a second argument, which is added to the index. This way you can start numbering at 1, or any other place.

Example

enumerate.py

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

colors = "red blue green yellow brown black".split()

months = "Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec".split()

for i, color in enumerate(colors): ①
    print(i, color)

print()

for num, month in enumerate(months, 1): ②
    print("{} {}".format(num, month))
```

① enumerate() returns iterable of (index, value) tuples

② Second parameter to enumerate is added to index

enumerate.py

```
0 red
1 blue
2 green
3 yellow
4 brown
5 black
```

```
1 Jan
2 Feb
3 Mar
4 Apr
5 May
6 Jun
7 Jul
8 Aug
9 Sep
10 Oct
11 Nov
12 Dec
```


Operators and keywords for sequences

- Operators + *
- Keywords **del in not in**

del deletes an entire string, list, or tuple. It can also delete one element, or a slice, from a list. **del** cannot remove elements of strings and tuples, because they are immutable.

in returns True if the specified object is an element of the sequence.

not in returns True if the specified object is *not* an element of the sequence.

+ adds one sequence to another

***** multiplies a sequence (i.e., makes a bigger sequence by repeating the original).

```
x in s #note - x can be any Python object
s2 = s1 * 3
s3 = s1 + s2
```

Example

sequence_operators.py

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

colors = ["red", "blue", "green", "yellow", "brown", "black"]

months = (
    "Jan", "Feb", "Mar", "Apr", "May", "Jun",
    "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
)

print("yellow in colors: ", ("yellow" in colors)) ①
print("pink in colors: ", ("pink" in colors))

print("colors: ", ", ".join(colors)) ②

del colors[4] # remove brown ③

print("removed 'brown':", ", ".join(colors))

colors.remove('green') ④

print("removed 'green':", ", ".join(colors))

sum_of_lists = [True] + [True] + [False] ⑤

print("sum of lists:", sum_of_lists)

product = [True] * 5 ⑥

print("product of lists:", product)
```

- ① Test for membership in list
- ② Concatenate iterable using ", " as delimiter
- ③ Permanently remove element with index 4
- ④ Remove element by value
- ⑤ Add 3 lists together; combines all elements
- ⑥ Multiply a list; replicates elements

sequence_operators.py

```
yellow in colors: True
pink in colors: False
colors: red,blue,green,yellow,brown,black
removed 'brown': red,blue,green,yellow,black
removed 'green': red,blue,yellow,black
sum of lists: [True, True, False]
product of lists: [True, True, True, True, True]
```

The range() function

- Provides (virtual) list of numbers
- Slice-like parameters
- Syntax

```
range(stop)
range(start, stop)
range(start, stop, step)
```

The range() function returns a **range object**, that provides a list of numbers when iterated over. The parameters to range() are similar to the parameters for slicing (start, stop, step).

This can be useful to execute some code a fixed number of times.

Example

using_ranges.py

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

print("range(1, 6): ", end=' ')
for x in range(1, 6): ①
    print(x, end=' ')
print()

print("range(6): ", end=' ')
for x in range(6): ②
    print(x, end=' ')
print()

print("range(3, 12): ", end=' ')
for x in range(3, 12): ③
    print(x, end=' ')
print()

print("range(5, 30, 5): ", end=' ')
for x in range(5, 30, 5): ④
    print(x, end=' ')
print()

print("range(10, 0, -1): ", end=' ')
for x in range(10, 0, -1): ⑤
    print(x, end=' ')
print()
```

- ① Start=1, Stop=6 (1 through 5)
- ② Start=0, Stop=6 (0 through 5)
- ③ Start=3, Stop=12 (3 through 11)
- ④ Start=5, Stop=30, Step=5 (5 through 25 by 5)
- ⑤ Start=10, Stop=1, Step=-1 (10 through 1 by 1)

using_ranges.py

```
range(1, 6): 1 2 3 4 5
range(6): 0 1 2 3 4 5
range(3, 12): 3 4 5 6 7 8 9 10 11
range(5, 30, 5): 5 10 15 20 25
range(10, 0, -1): 10 9 8 7 6 5 4 3 2 1
```

List comprehensions

- Shortcut for a for loop
- Optional if clause
- Always returns list
- Syntax

```
[ EXPR for VAR in SEQUENCE if EXPR ]
```

A list comprehension is a Python idiom that creates a shortcut for a for loop. A loop like this:

```
results = []  
for var in sequence:  
    results.append(expr)    # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional if may be added:

```
results = [ expr for var in sequence if expr ]
```

The loop expression can be a tuple. You can nest two or more for loops.

Example

list_comprehensions.py

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

fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

ufruits = [fruit.upper() for fruit in fruits] ①
afruits = [fruit.title() for fruit in fruits if fruit.startswith('a')] ②

print("ufruits:", ufruits)
print("afruits:", afruits)
print()

values = [2, 42, 18, 39.7, 92, '14', "boom", ['a', 'b', 'c']]

doubles = [v * 2 for v in values] ③

print("doubles:", doubles, '\n')

nums = [x for x in values if isinstance(x, int)] ④
print(nums, '\n')

dirty_strings = ['   Gronk   ', 'PULABA   ', '   floog']

clean = [d.strip().lower() for d in dirty_strings]
for c in clean:
    print(">{}<".format(c), end=' ')
print("\n")

suits = 'Clubs', 'Diamonds', 'Hearts', 'Spades'
ranks = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

deck = [(rank, suit) for suit in suits for rank in ranks] ⑤

for rank, suit in deck:
    print("{}-{}".format(rank, suit))
```


- ① Simple transformation of all elements
- ② Transformation of selected elements only
- ③ Any kind of data is OK
- ④ Select only integers from list
- ⑤ More than one **for** is OK

list_comprehensions.py

```

ufruits: ['WATERMELON', 'APPLE', 'MANGO', 'KIWI', 'APRICOT', 'LEMON', 'GUAVA']
afruits: ['Apple', 'Apricot']

doubles: [4, 84, 36, 79.4, 184, '1414', 'boomboom', ['a', 'b', 'c', 'a', 'b', 'c']]

[2, 42, 18, 92]

>gronk< >pulaba< >floog<

2-Clubs
3-Clubs
4-Clubs
5-Clubs
6-Clubs
7-Clubs
8-Clubs
9-Clubs
10-Clubs
J-Clubs
Q-Clubs
K-Clubs
A-Clubs
2-Diamonds
3-Diamonds
4-Diamonds
5-Diamonds
6-Diamonds
7-Diamonds
8-Diamonds
9-Diamonds

```

etc etc

Generator Expressions

- Similar to list comprehensions
- Lazy evaluations – only execute as needed
- Syntax

```
( EXPR for VAR in SEQUENCE if EXPR )
```

A generator expression is very similar to a list comprehension. There are two major differences, one visible and one invisible.

The visible difference is that generator expressions are created with parentheses rather than square brackets. The invisible difference is that instead of returning a list, they return an iterable object.

The object only fetches each item as requested, and if you stop partway through the sequence; it never fetches the remaining items. Generator expressions are thus frugal with memory.

Example

generator_expressions.py

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

fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

ufruits = (fruit.upper() for fruit in fruits) ①
afruits = (fruit.title() for fruit in fruits if fruit.startswith('a'))

print("ufruits:", " ".join(ufruits))
print("afruits:", " ".join(afruits))
print()

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]
doubles = (v * 2 for v in values)

print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print("\n")

nums = (int(s) for s in values if isinstance(s, int))
for n in nums:
    print(n, end=' ')
print("\n")

dirty_strings = ['   Gronk   ', 'PULABA   ', '   floog']

clean = (d.strip().lower() for d in dirty_strings)
for c in clean:
    print(">{}<".format(c), end=' ')
print("\n")

powers = ((i, i ** 2, i ** 3) for i in range(1, 11))
for num, square, cube in powers:
    print("{:2d} {:3d} {:4d}".format(num, square, cube))
print()
```

- ① These are all exactly like the list comprehension example, but return generators rather than lists

generator_expressions.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: Apple Apricot

doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']

2 42 18 92

>gronk< >pulaba< >floog<

1 1 1
2 4 8
3 9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

Chapter 4 Exercises

Exercise 4-1 (pow2.py)

Print out all the powers of 2 from 2^0 through 2^{31} .

Use the `**` operator, which raises a number to a power.

TIP

For exercises 5-2 and 5-3, start with the file `sequences.py`, which has the lists `ctemps` and `fruits` already typed in. You can put all the answers in `sequences.py`

Exercise 4-2 (sequences.py)

`ctemps` is a list of Celsius temperatures. Loop through `ctemps`, convert each temperature to Fahrenheit, and print out both temperatures.

Exercise 4-3 (sequences.py)

Use a list comprehension to copy the list `fruits` to a new list named `clean_fruits`, with all fruits in lower case and leading/trailing white space removed. Print out the new list.

HINT: Use chained methods (`x.spam().ham()`)

Exercise 4-4 (sieve.py)

FOR ADVANCED STUDENTS

The "Sieve of Eratosthenes" is an ancient algorithm for finding prime numbers. It works by starting at 2 and checking each number up to a specified limit. If the number has been marked as non-prime, it is skipped. Otherwise, it is prime, so it is output, and all its multiples are marked as non-prime.

Write a program to implement this algorithm. Specify the limit (the highest number to check) on the script's command line. Supply a default if no limit is specified.

Initialize a list (maybe named `is_prime`) to the size of the limit plus one (use `*` to multiply a single-item list). All elements should be set to `True`.

Use two loops. The outer loop will check each value (element of the array) from 2 to the upper limit. (use the `range()` function).

If the element has a `True` value (is prime), print out its value. Then, execute a second loop iterates through all the multiples of the number, and marks them as `False` (i.e., non-prime).

No action is needed if the element has a `False` value. This will skip the non-prime numbers.

TIP Use `range()` to generate the multiples of the current number.

NOTE In this exercise, the *value* of the element is either `True` or `False` — the *index* is the number be checked for primeness.

See next page for the pseudocode for this program:

Pseudocode for sieve.py

```
if # command line args == 1
    get LIMIT from command line
else
    set LIMIT to 50

Initialize IS_PRIMES list to size LIMIT+1, with all TRUE values

for NUM from 2 to LIMIT+1
    if IS_PRIME[NUM]
        output NUM
        for M from NUM to LIMIT+1, counting by NUM
            IS_PRIME[M] = FALSE
```


Chapter 5: Working with Files

Objectives

- Reading a text file line-by-line
- Reading an entire text files
- Reading all lines of a text file into an array
- Writing to a text file

Text file I/O

- Create a file object with `open`
- Specify modes: read/write, text/binary
- Read or write from file object
- Close file object (or use **with** block)

Python provides a file object that is created by the built-in `open()` function. From this file object you can read or write data in several different ways. When opening a file, you specify the file name and the mode, which says whether you want to read, write, or append to the file, and whether you want text or binary (raw) processing.

NOTE

This chapter is about working with generic files. For files in standard formats, such as XML, CSV, YAML, JSON, and many others, Python has format-specific modules to read them.

Opening a text file

- Specify the file name and the mode
- Returns a file object
- Mode can be read or write
- Specify "b" for binary (raw) mode
- Omit mode for reading

Open a text file with the `open()` command. Arguments are the file name, which may be specified as a relative or absolute path, and the mode. The mode consists of "r" for read, "w" for write, or "a" for append. To open a file in binary mode, add "b" to the mode, as in "rb", "wb", or "ab".

If you omit the mode, "r" is the default.

Example

```
ty = open("tyger.txt", "r")  open for reading in text mode
ty = open("tyger.txt")      open for reading in text mode (default mode)
junk = open("junk.dat", "rb") open for reading in raw mode
stf = open("stuff.txt", "w") open for writing in text mode
moju = open("morejunk.dat", "wb") open for writing in raw mode
config = open("spam.cfg", "a") open for append in text mode
```

The **fileinput** module in the standard library makes it easy to loop over each line in all files specified on the command line, or STDIN if no files are specified. This avoids having to open and close each file.

The *with* block

- Provides "execution context"
- Automatically closes file object
- Not specific to file objects

Because it is easy to forget to close a file object, you can use a **with** block to open your file. This will automatically close the file object when the block is finished. The syntax is

```
with open(filename, mode) as fileobject:  
    # process fileobject
```

Reading a text file

- Iterate through file with for/in

```
for line in file_in
```

- Use methods of the file object

```
file_in.readlines()  read all lines from file_in
file_in.read()       read all of file_in
file_in.read(n)      read n characters from file in text mode; n bytes
from file_in in binary mode
file_in.readline()   read next line from file_in
```

The easiest way to read a file is by looping through the file object with a for/in loop. This is possible because the file object is an iterator, which means the object knows how to provide a sequence of values.

You can also read a text file one line or multiple lines at a time. **readline()** reads the next available line; **readlines()** reads all lines into a list.

read() will read the entire file; **read(n)** will read n bytes from the file (n *characters* if in text mode).

readline() will read the next line from the file.

Example

read_tyger.py

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

with open("../DATA/tyger.txt", "r") as tyger_in: ①
    for line in tyger_in: ②
        print(line, end='') ③
```

- ① **tyger_in** is return value of **open(...)**
- ② **tyger_in** is a *generator*, returning one line at a time
- ③ the line already has a newline, so **print()** does not need one

read_tyger.py

The Tyger

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Could frame thy fearful symmetry?

In what distant deeps or skies
Burnt the fire of thine eyes?
On what wings dare he aspire?
What the hand dare seize the fire?

And what shoulder, & what art,
Could twist the sinews of thy heart?
And when thy heart began to beat,
What dread hand? & what dread feet?

What the hammer? what the chain?
In what furnace was thy brain?
What the anvil? what dread grasp
Dare its deadly terrors clasp?

When the stars threw down their spears
And water'd heaven with their tears,
Did he smile his work to see?
Did he who made the Lamb make thee?

Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand or eye
Dare frame thy fearful symmetry?

by William Blake

Example

reading_files.py

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

print("** About Spam **")
with open("../DATA/spam.txt") as spam_in:
    for line in spam_in:
        print(line.rstrip('\r\n')) ①

with open("../DATA/eggs.txt") as eggs_in:
    eggs = eggs_in.readlines() ②

print("\n\n** About Eggs **")
print(eggs[0].rstrip()) ③
print(eggs[2].rstrip())
```

- ① `rstrip("\n\r")` removes `\r` or `\n` from end of string
- ② `readlines()` reads all lines into an array
- ③ `[:-1]` is another way to skip the newline (but `.rstrip()` does not remove spaces or tabs if they are significant)

reading_files.py

**** About Spam ****

SPAM may be famous now, but it wasn't always that way. Fact is, SPAM hails from some rather humble beginnings.

Flash back to Austin, Minnesota, in 1937.

You're right. There isn't much here, except for an ambitious company called Hormel.

These good folks are about to hit upon an amazing little recipe: a spicy ham packaged in a handy dandy 12-ounce can.

J. C. Hormel, then president, adds the crowning ingredient: He holds a contest to give the product a name as distinctive as its taste.

SPAM soars. In fact, in that very first year of production, it grabs 18 percent of the market.

Over 65 years later, more than 6 billion cans of SPAM have been sold.

**** About Eggs ****

You can scramble, fry, poach and bake eggs or cook them in their shells.

Eggs are also the main ingredient in some dishes that came to the U.S. from other countries, such as a frittata, egg foo yung, quiche or souffle.

Writing to a text file

- Use `write()` or `writelines()`
- Add `\n` manually

To write to a text file, use the `write()` function to write a single string; or `writelines()` to write a list of strings.

`writelines()` will not add newline characters, so make sure the items in your list already have them.

Example

`write_file.py`

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

states = (
    'Virginia',
    'North Carolina',
    'Washington',
    'New York',
    'Florida',
    'Ohio',
)

with open("states.txt", "w") as states_out: ①
    for state in states:
        states_out.write(state + "\n") ②
```

① "w" opens for writing, "a" for append

② `write()` does not add `\n` automatically

write_file.py

cat states.txt (or type states.txt under windows)

```
Virginia  
North Carolina  
Washington  
New York  
Florida  
Ohio
```

"writelines" should have been called "writestrings"

Table 10. File Methods

Function	Description
<code>f.close()</code>	close file <code>f</code>
<code>f.flush()</code>	write out buffered data to file <code>f</code>
<code>s = f.read(n)</code> <code>s = f.read()</code>	read size bytes from file <code>f</code> into string <code>s</code> ; if <code>n</code> is ≤ 0 , or omitted, reads entire file
<code>s = f.readline()</code> <code>s = f.readline(n)</code>	read one line from file <code>f</code> into string <code>s</code> . If <code>n</code> is specified, read no more than <code>n</code> characters
<code>m = f.readlines()</code>	read all lines from file <code>f</code> into list <code>m</code>
<code>f.seek(n)</code> <code>f.seek(n,w)</code>	position file <code>f</code> at offset <code>n</code> for next read or write; if argument <code>w</code> (whence) is omitted or 0, offset is from beginning; if 1, from current file position, if 2, from end of file
<code>f.tell()</code>	return current offset from beginning of file
<code>f.write(s)</code>	write string <code>s</code> to file <code>f</code>
<code>f.writelines(m)</code>	write list of strings <code>m</code> to file <code>f</code> ; does not add line terminators

Chapter 5 Exercises

Exercise 5-1 (line_no.py)

Write a program to display each line of a file preceded by the line number. Allow your program to process one or more files specified on the command line. Be sure to reset the line number for each file.

TIP | Use `enumerate()`.

Test with the following commands:

```
python line_no.py DATA/tyger.txt
python line_no.py DATA/parrot.txt DATA/tyger.txt
```

Test with other files, as desired

Exercise 5-2 (alt_lines.py)

Write a program to create two files, a.txt and b.txt from the file alt.txt. Lines that start with 'a' go in a.txt; the other lines (which all start with 'b') go in b.txt. Compare the original to the two new files.

Exercise 5-3 (count_alice.py, count_words.py)

- A. Write a program to count how many lines of alice.txt contain the word "Alice". (There should be 392).

TIP | Use the `in` operator to test whether a line contains the word "Alice"

- B. Modify count_alice.py to take the first command line parameter as a word to find, and the remaining parameters as filenames. For each file, print out the file name and the number of lines that contain the specified word. Test thoroughly

FOR ADVANCED STUDENTS (icount_words.py) Modify count_words.py to make the search case-insensitive.

Chapter 6: Dictionaries

Objectives

- Creating dictionaries
- Using dictionaries for mapping and counting
- Iterating through key-value pairs
- Reading a file into a dictionary
- Counting with a dictionary
- Using sets

About dictionaries

- A collection
- Associates keys with values
- called "hashes", "hash tables" or "associative arrays" in other languages
- Rich set of functions available

A dictionary is a collection that contains key-value pairs. Dictionaries are not sequential like lists, tuples, and strings; they function more as a lookup table. They map one value to another.

The keys must be immutable – lists and dictionaries may not be used as keys. Any immutable type may be a key, although typically keys are strings.

Prior to version 3.6, the elements of a dictionary are in no particular order. Starting with 3.6, elements are stored in the order added. If you iterate over *dictionary.items()*, it will iterate in the order that the elements were added.

Values can be any Python object – strings, numbers, tuples, lists, dates, or anything else.

For instance, a dictionary might

- map column names in a database table to their corresponding values
- map almost any group of related items to a unique identifier
- map screen names to real names
- map zip codes to a count of customers per zip code
- count error codes in a log file
- count image tags in an HTML file

When to use dictionaries?

- Mapping
- Counting

Dictionaries are very useful for mapping a set of keys to a corresponding set of values. You could have a dictionary where the key is a candidate for office, and value is the state in which the candidate is running, or the value could be an object containing many pieces of information about the candidate.

Dictionaries are also handy for counting. The keys could be candidates and the values could be the number of votes each candidate received.

Creating dictionaries

- Create dictionaries with { } or dict()
- Create from (nearly) any sequence
- Add additional keys by assignment

To create a dictionary, use the dict() function or {}. The dictionary can be created empty, or you can initialize it with one or more key/value pairs, separated by colons.

To add more keys, assign to the dictionary using square brackets.

Remember, braces are only used to create a dictionary; indexing uses brackets like all the other container types. To get the value for a given key, specify the key with square brackets or use the get() method.

Example

creating_dicts.py

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

d1 = dict() ①

airports = {'IAD': 'Dulles', 'SEA': 'Seattle-Tacoma', ②
            'RDU': 'Raleigh-Durham', 'LAX': 'Los Angeles'}

d2 = {}
d3 = dict(red=5, blue=10, yellow=1, brown=5, black=12) ③

pairs = [('Washington', 'Olympia'), ('Virginia', 'Richmond'),
         ('Oregon', 'Salem'), ('California', 'Sacramento')]

state_caps = dict(pairs) ④

print(d3['red']) ⑤
print(airports['LAX'])

airports['SLC'] = 'Salt Lake City' ⑥
airports['LAX'] = 'Lost Angels' ⑦
print(airports['SLC'])
```

- ① create new empty dict
- ② initialize dict with literal key/value pairs (keys can be any string, number or tuple)
- ③ initialize dict with named parameters; keys must be valid identifier names
- ④ initialize dict with an iterable of pairs
- ⑤ print value for given key
- ⑥ assign to new key
- ⑦ overwrite existing key

creating_dicts.py

```
5  
Los Angeles  
Salt Lake City
```

Table 11. Frequently used dictionary functions and operators

Function	Description
<code>len(D)</code>	the number of elements in D
<code>D[k]</code>	the element of D with key k
<code>D[k] = v</code>	set D[k] to v
<code>del D[k]</code>	remove element from D whose key is k
<code>D.clear()</code>	remove all items from a dictionary
<code>k in D</code>	True if key k exists in D
<code>k not in D</code>	True if key k does not exist in D
<code>D.get(k[, x])</code>	D[k] if k in a, else x
<code>D.items()</code>	return an iterator over (key, value) pairs
<code>D.update([b])</code>	updates (and overwrites) key/value pairs from b
<code>D.setdefault(k[, x])</code>	a[k] if k in D, else x (also setting it)

Table 12. Less frequently used dictionary functions

Function	Description
<code>D.keys()</code>	return an iterator over the mapping's keys
<code>D.values()</code>	return an iterator over the mapping's values
<code>D.copy()</code>	a (shallow) copy of D
<code>D.has_key(k)</code>	True if a has D key k, else False (but use in)
<code>D.fromkeys(seq[, value])</code>	Creates a new dictionary with keys from seq and values set to value
<code>D.pop(k[, x])</code>	a[k] if k in D, else x (and remove k)
<code>D.popitem()</code>	remove and return an arbitrary (key, value) pair

Getting dictionary values

- `d[key]`
- `d.get(key,default-value)`
- `d.setdefault(key, default-value)`

There are three main ways to get the value of a dictionary element, given the key.

Using the key as an index retrieves the corresponding value, or raises a `KeyError`.

The `get()` method returns the value, or a default value if the key does not exist. If no default value is specified, and the key does not exist, `get()` returns `None`.

The `setdefault()` method is like `get()`, but if the key does not exist, adds the key and the default value to the dictionary.

Use the **`in`** operator to test whether a dictionary contains a given key.

Example

getting_dict_values.py

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

d1 = dict()

airports = {'IAD': 'Dulles', 'SEA': 'Seattle-Tacoma',
            'RDU': 'Raleigh-Durham', 'LAX': 'Los Angeles'}

d2 = {}
d3 = dict(red=5, blue=10, yellow=1, brown=5, black=12)

pairs = [('Washington', 'Olympia'), ('Virginia', 'Richmond'),
         ('Oregon', 'Salem'), ('California', 'Sacramento')]

state_caps = dict(pairs)

print(d3['red'])
print(airports['LAX'])

airports['SLC'] = 'Salt Lake City'
airports['LAX'] = 'Lost Angels'
print(airports['SLC']) ①

key = 'PSP'
if key in airports:
    print(airports[key]) ②

print(airports.get(key)) ③
print(airports.get(key, 'NO SUCH AIRPORT')) ④

print(airports.setdefault(key, 'Palm Springs')) ⑤
print(key in airports) ⑥
```

- ① print value where key is 'SLC'
- ② print key if key is in dictionary
- ③ get value if key in dict, otherwise get None
- ④ get value if key in dict, otherwise get 'NO SUCH AIRPORT'
- ⑤ get value if key in dict, otherwise get 'Palm Springs' AND set key
- ⑥ check for key in dict

getting_dict_values.py

```
5
Los Angeles
Salt Lake City
None
NO SUCH AIRPORT
Palm Springs
True
```


Iterating through a dictionary

- `d.items()` gives list of key/value tuples
- Key order
 - before 3.6: not predictable
 - 3.6 and later: insertion order

To iterate through tuples containing the key and the value, use the method `DICT.items()`. It generates tuples in the form `(KEY,VALUE)`.

Before 3.6, elements are retrieved in arbitrary order; beginning with 3.6, elements are retrieved in the order they were added.

To do something with the elements in a particular order, the usual approach is to pass ***DICT.items()*** to the ***sorted()*** function and loop over the result.

TIP

If you iterate through the dictionary itself (as opposed to `dictionary.items()`), you get just the keys.

Example

iterating_over_dicts.py

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

airports = {'IAD': 'Dulles', 'SEA': 'Seattle-Tacoma',
            'RDU': 'Raleigh-Durham', 'LAX': 'Los Angeles'}

for abbr, airport in airports.items(): ①
    print(abbr, airport)
```

① `items()` returns a virtual list of key:value pairs

iterating_over_dicts.py

```
IAD Dulles  
SEA Seattle-Tacoma  
RDU Raleigh-Durham  
LAX Los Angeles
```

Reading file data into a dictionary

- Data must have unique key
- Key is one column, value can be string, number, list, or tuple (or anything else!)

To read a file into a dictionary, read the file one line at a time, splitting the line into fields as necessary. Use a unique field for the key. The value can be either some other field, or a group of fields, as stored in a list or tuple. Remember that the value can be any Python object.

Example

read_into_dict_of_tuples.py

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

from pprint import pprint

knight_info = {} ①

with open("../DATA/knights.txt") as knights_in:
    for line in knights_in:
        (name, title, color, quest, comment) = line.rstrip('\n\r').split(":")
        knight_info[name] = title, color, quest, comment ②

pprint(knight_info)
print()

for name, info in knight_info.items():
    print(info[0], name)

print()
print(knight_info['Robin'][2])
```

① create empty dict

② create new dict element with **name** as key and a tuple of the other fields as the value

read_into_dict_of_tuples.py

```
{'Arthur': ('King', 'blue', 'The Grail', 'King of the Britons'),  
 'Bedevere': ('Sir', 'red, no blue!', 'The Grail', 'AARRRRRRRGGGGHH'),  
 'Galahad': ('Sir', 'red', 'The Grail', '"I could handle some more peril"'),  
 'Gawain': ('Sir', 'blue', 'The Grail', 'none'),  
 'Lancelot': ('Sir', 'blue', 'The Grail', '"It\'s too perilous!"),  
 'Robin': ('Sir', 'yellow', 'Not Sure', 'He boldly ran away')}
```

```
King Arthur  
Sir Galahad  
Sir Lancelot  
Sir Robin  
Sir Bedevere  
Sir Gawain
```

```
Not Sure
```

TIP

See also [read_into_dict_of_dicts.py](#) and [read_into_dict_of_named_tuples.py](#) in the [EXAMPLES](#) folder.

Counting with dictionaries

- Use dictionary where key is item to be counted
- Value is number of times item has been seen.

To count items, use a dictionary where the key is the item to be counted, and the value is the number of times it has been seen (i.e., the count).

The `get()` method is useful for this. The first time an item is seen, `get` can return 0; thereafter, it returns the current count. Each time, add 1 to this value.

TIP | [Check out the `Counter` class in the `collections` module](#)

Example

`count_with_dict.py`

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

counts = {} ①
with open("../DATA/breakfast.txt") as breakfast_in:
    for line in breakfast_in:
        breakfast_item = line.rstrip('\n\r')
        if breakfast_item in counts: ②
            counts[breakfast_item] = counts[breakfast_item] + 1 ③
        else:
            counts[breakfast_item] = 1 ④

for item, count in counts.items():
    print(item, count)
```

① create empty dict

② create new dict element with **name** as key and a tuple of the other fields as the value

count_with_dict.py

```
spam 10  
eggs 3  
crumpets 1
```

As a short cut, you could check for the key and increment with a one-liner:

```
counts[breakfast_item] = counts.get(breakfast_item,0) + 1
```

About sets

- Find unique values
- Check for membership
- Find union or intersection
- Like a dictionary where all values are True
- Two kinds of sets
 - set (mutable)
 - frozenset (immutable)

A set is useful when you just want to keep track of a group of values, but there is no particular value associated with them .

The easy way to think of a set is that it's like a dictionary where the value of every element is True. That is, the important thing is whether the key is in the set or not.

There are methods to compute the union, intersection, and difference of sets, along with some more esoteric functionality.

As with dictionary keys, the values in a set must be unique. If you add a key that already exists, it doesn't change the set.

You could use a set to keep track of all the different error codes in a file, for instance.

Creating Sets

- Literal set: {item1, item2, ...}
- Use set() or frozenset()
- Add members with SET.add()

To create a set, use the set() constructor, which can be initialized with any iterable. It returns a set object, to which you can then add elements with the add() method.

Create a literal set with curly braces containing a comma-separated list of the members. This won't be confused with a literal dictionary, because dictionary elements contain a colon separating the key and value.

To create an immutable set, use frozenset(). Once created, you may not add or delete items from a frozenset. This is useful for quick lookup of valid values.

Working with sets

- Common set operations
 - adding an element
 - deleting an element
 - checking for membership
 - computing
 - union
 - intersection
 - symmetric difference (xor)

The most common thing to do with a set is to check for membership. This is accomplished with the **in** operator. New elements are added with the **add()** method, and elements are deleted with the **del** operator.

Intersection (&) of two sets returns a new set with members common to both sets.

Union (|) of two sets returns a new set with all members from both sets.

Xor (^) of two sets returns a new set with members that are one one set or the other, but not both. (AKA symmetric difference)

Difference (-) of two sets returns a new set with members on the right removed from the set on the left.

Example

set_examples.py

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

set1 = {'red', 'blue', 'green', 'purple', 'green'} ①
set2 = {'green', 'blue', 'yellow', 'orange'}

set1.add('taupe') ②

print(set1)
print(set2)
print(set1 & set2) ③
print(set1 | set2) ④
print(set1 ^ set2) ⑤
print(set1 - set2) ⑥
print(set2 - set1)
print()

food = 'spam ham ham spam spam ham spam eggs cheese spam'.split()
food_set = set(food) ⑦
print(food_set)
```

- ① create literal set
- ② add element to set (ignored if already in set)
- ③ intersection of two sets
- ④ union of two sets
- ⑤ XOR (symmetric difference); items in one set but not both
- ⑥ Remove items in right set from left set
- ⑦ Create set from iterable (e.g., list)

set_examples.py

```
{'red', 'taupe', 'green', 'blue', 'purple'}
{'blue', 'green', 'orange', 'yellow'}
{'blue', 'green'}
{'red', 'taupe', 'green', 'blue', 'orange', 'purple', 'yellow'}
{'red', 'taupe', 'orange', 'purple', 'yellow'}
{'red', 'taupe', 'purple'}
{'yellow', 'orange'}

{'spam', 'ham', 'cheese', 'eggs'}
```

Table 13. Set functions and methods

Function	Description
<code>m in</code>	True if s contains member m
<code>m not in</code>	True if s does not contain member m
<code>len(s)</code>	the number of items in s
<code>s.add(m)</code>	Add member m to s (if s already contains m do nothing)
<code>s.clear()</code>	remove all members from s
<code>s.copy()</code>	a (shallow) copy of s
<code>s - s2</code> <code>s.difference(s2)</code>	Return the set of all elements in s that are not in s2
<code>s.difference_update(s2)</code>	Remove all members of s2 from s
<code>s.discard(m)</code>	Remove member m from s if it is a member. If m is not a member, do nothing.
<code>s & s2</code> <code>s.intersection(s2)</code>	Return new set with all unique members of s and s2
<code>s.isdisjoint(s2)</code>	Return True if s and s2 have no members in common
<code>s.issubset(s2)</code>	Return True if s is a subset of s2
<code>s.issuperset(s2)</code>	Return True if s2 is a subset of s
<code>s.pop()</code>	Remove and return an arbitrary set element. Raises <code>KeyError</code> if the set is empty.
<code>s.remove(m)</code>	Remove member m from a set; it must be a member.
<code>s ^ s2</code> <code>s.symmetric_difference(s2)</code>	Return all members in s or s2 but not both.
<code>s.symmetric_difference_update(s2)</code>	Update a set with the symmetric difference of itself and another.
<code>s s2</code> <code>s.union(s2)</code>	Return all members that are in s or s2
<code>s.update(s2)</code>	Update a set with the union of itself and s2

Chapter 6 Exercises

Exercise 6-1 (scores.py)

A class of students has taken a test. Their scores have been stored in **testscores.dat**. Write a program named **scores.py** to read in the data (read it into a dictionary where the keys are the student names and the values are the test scores). Print out the student names, one per line, sorted, and with the numeric score and letter grade. After printing all the scores, print the average score.

```
Grading Scale
95-100
A
89-94
B
83-88
C
75-82
D
< 75
F
```

Exercise 6-2 (shell_users.py)

Using the file named **passwd**, write a program to count the number of users using each shell. To do this, read **passwd** one line at a time. Split each line into its seven (colon-delimited) fields. The shell is the last field. For each entry, add one to the dictionary element whose key is the shell.

When finished reading the password file, loop through the keys of the dictionary, printing out the shell and the count.

Exercise 6-3 (common_fruit.py)

Using sets, compute which fruits are in both **fruit1.txt** and **fruit2.txt**. To do this, read the files into sets (the files contain one fruit per line) and find the intersection of the sets.

What if fruits are in both files, but one is capitalized and the other isn't?

Exercise 6-4 (set_sieve.py)

FOR ADVANCED STUDENTS Rewrite **sieve.py** to use a set rather than a list to keep track of which numbers are non-prime. This turns out to be easier – you don’t have to initialize the set, as you did with the list.

Chapter 7: Functions, Modules and Packages

Objectives

- Define functions
- Learn the four kinds of function parameters
- Create new modules
- Load modules with **import**
- Set module search locations
- Organize modules into packages
- Alias module and package names

Functions

- Defined with **def**
- Accept parameters
- Return a value

Functions are a way of isolating code that is needed in more than one place, refactoring code to make it more modular. They are defined with the **def** statement.

Functions can take various types of parameters, as described on the following page. Parameter types are dynamic.

Functions can return one object of any type, using the **return** statement. If there is no return statement, the function returns **None**.

TIP

Be sure to separate your business logic (data and calculations) from your presentation logic (the user interface).

Example

function_basics.py

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

def say_hello(): ①
    print("Hello, world")
    print()
    ②

say_hello() ③

def get_hello():
    return "Hello, world" ④

h = get_hello() ⑤
print(h)
print()

def sqrt(num): ⑥
    return num **.5

m = sqrt(1234) ⑦
n = sqrt(2)

print("m is {:.3f} n is {:.3f}".format(m, n))
```

- ① Function takes no parameters
- ② If no **return** statement, return None
- ③ Call function (arguments, if any, in ())
- ④ Function returns value
- ⑤ Store return value in h
- ⑥ Function takes exactly one argument
- ⑦ Call function with one argument

function_basics.py

```
Hello, world
```

```
Hello, world
```

```
m is 35.128 n is 1.414
```

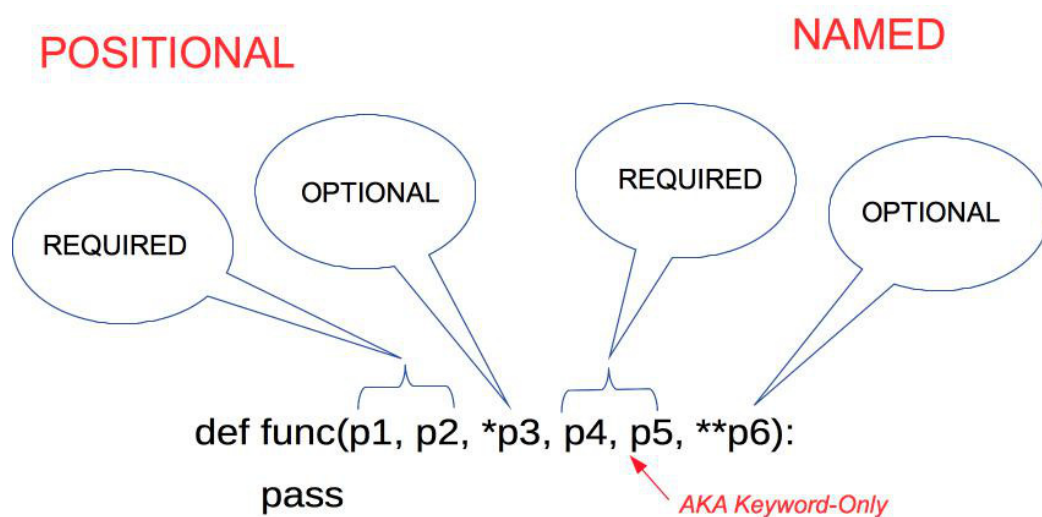
Function parameters

- Positional or named
- Required or optional
- Can have default values

Functions can accept both positional and named parameters. Furthermore, parameters can be required or optional. They must be specified in the order presented here.

The first set of parameters, if any, is a set of comma-separated names. These are all required. Next you can specify a variable preceded by an asterisk—this will accept any optional parameters.

After the optional positional parameters you can specify required named parameters. These must come after the optional parameters. If there are no optional parameters, you can use a plain asterisk as a placeholder. Finally, you can specify a variable preceded by two asterisks to accept optional named parameters.



Example

function_parameters.py

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

def fun_one(): ①
    print("Hello, world")

print("fun_one():", end=' ')
fun_one()
print()

def fun_two(n): ②
    return n ** 2

x = fun_two(5)
print("fun_two(5) is {}\n".format(x))

def fun_three(count=3): ③
    for _ in range(count):
        print("spam", end=' ')
    print()

fun_three()
fun_three(10)
print()

def fun_four(n, *opt): ④
    print("fun_four():")
    print("n is ", n)
    print("opt is", opt)
    print('-' * 20)

fun_four('apple')
fun_four('apple', "blueberry", "peach", "cherry")
```

```
def fun_five(*, spam=0, eggs=0): ⑤
    print("fun_five():")
    print("spam is:", spam)
    print("eggs is:", eggs)
    print()
```

```
fun_five(spam=1, eggs=2)
fun_five(eggs=2, spam=2)
fun_five(spam=1)
fun_five(eggs=2)
fun_five()
```

```
def fun_six(**named_args): ⑥
    print("fun_six():")
    for name in named_args:
        print(name, "==> ", named_args[name])
```

```
fun_six(name="Lancelot", quest="Grail", color="red")
```

- ① no parameters
- ② one required parameter
- ③ one required parameter with default value
- ④ one fixed, plus optional parameters
- ⑤ keyword-only parameters
- ⑥ keyword (named) parameters

function_parameters.py

```
fun_one(): Hello, world

fun_two(5) is 25

spam spam spam
spam spam spam spam spam spam spam spam

fun_four():
n is apple
opt is ()
-----
fun_four():
n is apple
opt is ('blueberry', 'peach', 'cherry')
-----
fun_five():
spam is: 1
eggs is: 2

fun_five():
spam is: 2
eggs is: 2

fun_five():
spam is: 1
eggs is: 0

fun_five():
spam is: 0
eggs is: 2

fun_five():
spam is: 0
eggs is: 0

fun_six():
name ==> Lancelot
quest ==> Grail
color ==> red
```

Default parameters

- Assigned with equals sign
- Used if no values passed to function

Required parameters can have default values. They are assigned to parameters with the equals sign. Parameters without defaults cannot be specified after parameters with defaults.

Example

default_parameters.py

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

def spam(greeting, whom='world'): ①
    print("{} {}".format(greeting, whom))

spam("Hello", "Mom") ②
spam("Hello") ③
print()

def ham(*, file_name, file_format='txt'): ④
    print("Processing {} as {}".format(file_name, file_format))

ham(file_name='eggs') ⑤
ham(file_name='toast', file_format='csv')
```

- ① 'world' is default value for positional parameter **whom**
- ② parameter supplied; default not used
- ③ parameter not supplied; default is used
- ④ 'world' is default value for named parameter **format**
- ⑤ parameter **format** not supplied; default is used

default_parameters.py

```
Hello, Mom  
Hello, world  
  
Processing eggs as txt  
Processing toast as csv
```


Python Function parameter behavior (from PEP 3102)

For each formal parameter, there is a slot which will be used to contain the value of the argument assigned to that parameter.

- Slots which have had values assigned to them are marked as 'filled'. Slots which have no value assigned to them yet are considered 'empty'.
- Initially, all slots are marked as empty.
- Positional arguments are assigned first, followed by keyword arguments.
- For each positional argument:
 - Attempt to bind the argument to the first unfilled parameter slot. If the slot is not a vararg slot, then mark the slot as 'filled'.
 - If the next unfilled slot is a vararg slot, and it does not have a name, then it is an error.
 - Otherwise, if the next unfilled slot is a vararg slot then all remaining non-keyword arguments are placed into the vararg slot.
- For each keyword argument:
 - If there is a parameter with the same name as the keyword, then the argument value is assigned to that parameter slot. However, if the parameter slot is already filled, then that is an error.
 - Otherwise, if there is a 'keyword dictionary' argument, the argument is added to the dictionary using the keyword name as the dictionary key, unless there is already an entry with that key, in which case it is an error.
 - Otherwise, if there is no keyword dictionary, and no matching named parameter, then it is an error.
- Finally:
 - If the vararg slot is not yet filled, assign an empty tuple as its value.
 - For each remaining empty slot: if there is a default value for that slot, then fill the slot with the default value. If there is no default value, then it is an error.
- In accordance with the current Python implementation, any errors encountered will be signaled by raising `TypeError`.

Name resolution (AKA Scope)

- What is "scope"
- Scopes used dynamically
- Four levels of scope
- Assignments always go into the innermost scope (starting with local)

A scope is the area of a Python program where an unqualified (not preceded by a module name) name can be looked up.

Scopes are used dynamically. There are four nested scopes that are searched for names in the following order:

local	local names bound within a function
nonlocal	local names plus local names of outer function(s)
global	the current module's global names
builtin	built-in functions (contents of <i><code>_builtins_</code></i> module)

Within a function, all assignments and declarations create local names. All variables found outside of local scope (that is, outside of the function) are read-only.

Inside functions, local scope references the local names of the current function. Outside functions, local scope is the same as the global scope – the module's namespace. Class definitions also create a local scope.

Nested functions provide another scope. Code in function B which is defined inside function A has read-only access to all of A's variables. This is called **nonlocal** scope.

Example

scope_examples.py

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

x = 42 ①

def function_a():
    y = 5 ②

    def function_b():
        z = 32 ③
        print("function_b(): z is", z) ④
        print("function_b(): y is", y) ⑤
        print("function_b(): x is", x) ⑥
        print("function_b(): type(x) is", type(x)) ⑦

    return function_b

f = function_a() ⑧
f() ⑨
```

- ① global variable
- ② local variable to function_a(), or nonlocal to function_b()
- ③ local variable
- ④ local scope
- ⑤ nested (nonlocal) scope
- ⑥ global scope
- ⑦ builtin scope
- ⑧ calling function_a, which returns function_b
- ⑨ calling function_b

scope_examples.py

```
function_b(): z is 32  
function_b(): y is 5  
function_b(): x is 42  
function_b(): type(x) is <class 'int'>
```

The global statement

- `global` statement allows function to change globals
- `nonlocal` statement allows function to change nonlocals

The **`global`** keyword allows a function to modify a global variable. This is universally acknowledged as a BAD IDEA. Mutating global data can lead to all sorts of hard-to-diagnose bugs, because a function might change a global that affects some other part of the program. It's better to pass data into functions as parameters and return data as needed. Mutable objects, such as lists, sets, and dictionaries can be modified in-place.

The **`nonlocal`** keyword can be used like **`global`** to make nonlocal variables in an outer function writable.



Modules

- Files containing python code
- End with .py
- No real difference from scripts

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable *name*.

To use a module named `spam.py`, say `import spam`

This does not enter the names of the functions defined in `spam` directly into the symbol table; it only adds the module name **`spam`**. Use the module name to access the functions or other attributes.

Python uses modules to contain functions that can be loaded as needed by scripts. A simple module contains one or more functions; more complex modules can contain initialization code as well. Python classes are also implemented as modules.

A module is only loaded once, even there are multiple places in an application that import it.

Modules and packages should be documented with *docstrings*.

Using import

- import statement loads modules
- Three variations
 - import module
 - from module import function-list
 - from module import * use with caution!

There are three variations on the **import** statement:

Variation 1

`import module`

loads the module so its data and functions can be used, but does not put its attributes (names of classes, functions, and variables) into the current namespace.

Variation 2

`from module import function, ...`

imports only the function(s) specified into the current namespace. Other functions are not available (even though they are loaded into memory).

Variation 3

`from module import *`

loads the module, and imports all functions that do not start with an underscore into the current namespace. This should be used with caution, as it can pollute the current namespace and possibly overwrite builtin attributes or attributes from a different module.

NOTE

The first time a module is loaded, the interpreter creates a version compiled for faster loading. This version has platform information embedded in the name, and has the extension `.pyc`. These `.pyc` files are put in a folder named `__pycache__`.

Example

samplelib.py

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

# sample Python module

def spam():
    print("Hello from spam()")

def ham():
    print("Hello from ham()")

def _eggs():
    print("Hello from _eggs()")
```

use_samplelib1.py

```
#!/usr/bin/env python
import samplelib ①

samplelib.spam() ②
samplelib.ham()
```

① import samplelib module (samplelib.py) — creates object named **samplelib** of type "Module"

② call function spam() in module samplelib

use_samplelib1.py

```
Hello from spam()
Hello from ham()
```


use_samplelib2.py

```
#!/usr/bin/env python
from samplelib import spam, ham ①

spam() ②
ham()
```

- ① import functions spam and ham from samplelib module into current namespace — does not create the module object
- ② module name not needed to call function spam()

use_samplelib2.py

```
Hello from spam()
Hello from ham()
```

use_samplelib3.py

```
#!/usr/bin/env python
from samplelib import * ①

spam() ②
ham()
```

- ① import all functions (that do not start with _) from samplelib module into current namespace
- ② module name not needed to call function spam()

use_samplelib3.py

```
Hello from spam()
Hello from ham()
```

use_samplelib4.py

```
#!/usr/bin/env python
from samplelib import spam as pig, ham as hog ①

pig()
hog()
```

① import functions spam and ham, aliased to pig and hog

use_samplelib4.py

```
Hello from spam()
Hello from ham()
```

How *import ** can be dangerous

- Imported names may overwrite existing names
- Be careful to read the documentation

Using `import *` to import all public names from a module has a bit of a risk. While generally harmless, there is the chance that you will unknowingly import a module that overwrites some previously-imported module.

To be 100% certain, always import the entire module, or else import names explicitly.

Example

electrical.py

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

default_amps = 10
default_voltage = 110
default_current = 'AC'

def amps():
    return default_amps

def voltage():
    return default_voltage

def current():
    return default_current
```

navigation.py

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

current_types = 'slow medium fast'.split()

def current():
    return current_types[0]
```

why_import_star_is_bad.py

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

from electrical import * ①
from navigation import * ②

print(current()) ③
print(voltage())
print(amps())
```

- ① import current *explicitly* from electrical
- ② import current *implicitly* from navigation
- ③ calls navigation.current(), not electrical.current()

why_import_star_is_bad.py

```
slow
110
10
```

how_to_avoid_import_star.py

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

import electrical as e ①
import navigation as n ②

print(e.current()) ③
print(n.current()) ④
```

how_to_avoid_import_star.py

```
AC
slow
```

Module search path

- Searches current folder first, then predefined locations
- Add custom locations to PYTHONPATH
- Paths stored in sys.path

When you specify a module to load with the import statement, it first looks in the current directory, and then searches the directories listed in sys.path.

```
>>> import sys
>>> sys.path
```

To add locations, put one or more directories to search in the PYTHONPATH environment variable. Separate multiple paths by semicolons for Windows, or colons for Unix/Linux. This will add them to **sys.path**, after the current folder, but before the predefined locations.

Windows

```
set PYTHONPATH=C:"\Documents and settings\Bob\Python"
```

Linux/OS X

```
export PYTHONPATH="/home/bob/python"
```

You can also append to sys.path in your scripts, but this can result in non-portable scripts, and scripts that will fail if the location of the imported modules changes.

```
import sys
sys.path.extend("/usr/dev/python/libs", "/home/bob/pylib")
import module1
import module2
```

Executing modules as scripts

- `_name_` is current module.
 - set to `__main__` if run as script
 - set to `module_name` if imported
- test with `if name == "__main__"`
- Module can be run directly or imported

It is sometimes convenient to have a module also be a runnable script. This is handy for testing and debugging, and for providing modules that also can be used as standalone utilities.

Since the interpreter defines its own name as `'__main__'`, you can test the current namespace's `name` attribute. If it is `'__main__'`, then you are at the main (top) level of the interpreter, and your file is being run as a script; it was not loaded as a module.

Any code in a module that is not contained in function or method is executed when the module is imported.

This can include data assignments and other startup tasks, for example connecting to a database or opening a file.

Many modules do not need any initialization code.

Example

using_main.py

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

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

# constants (AKA global variables)

# main function
def main(args): ①
    function1()
    function2()

# other functions
def function1():
    print("hello from function1()")

def function2():
    print("hello from function2()")

if __name__ == '__main__':
    main(sys.argv[1:]) ②
```

- ① Program entry point. While **main** is not a reserved word, it is a strong convention
- ② Call `main()` with the command line parameters (omitting the script itself)

Packages

- Package is folder containing modules or packages
- Startup code goes in `__init__.py` (optional)

A package is a group of related modules or subpackages. The grouping is physical – a package is a folder that contains one or more modules. It is a way of giving a hierarchical structure to the module namespace so that all modules do not live in the same folder.

A package may have an initialization script named `__init__.py`. If present, this script is executed when the package or any of its contents are loaded. (In Python 2, `__init__.py` was required).

Modules in packages are accessed by prefixing the module with the package name, using the dot notation used to access module attributes.

Thus, if Module **eggs** is in package **spam**, to call the **scramble()** function in **eggs**, you would say `spam.eggs.scramble()`.

By default, importing a package name by itself has no effect; you must explicitly load the modules in the packages. You should usually import the module using its package name, like `from spam import eggs`, to import the eggs module from the spam package.

Packages can be nested.

Example

```
sound/                Top-level package
  __init__.py         Initialize the sound package (optional)
  formats/            Subpackage for file formats
    __init__.py       Initialize the formats package (optional)
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/            Subpackage for sound effects
    __init__.py       Initialize the formats package (optional)
    echo.py
    surround.py
    reverse.py
    ...
  filters/            Subpackage for filters
    __init__.py       Initialize the formats package (optional)
    equalizer.py
```

```
from sound.formats import aiffread
import sound.effects
import sound.filters.equalizer
```

Configuring import with `__init__.py`

- load modules into package's namespace
- specify modules to load when `*` is used

For convenience, you can put import statements in a package's `__init__.py` to autoload the modules into the package namespace, so that `import PKG` imports all the (or just selected) modules in the package.

`__init__.py` can also be used to setup data or other resources that will be used by multiple modules within a package.

If the variable `_all_` in `__init__.py` is set to a list of module names, then only these modules will be loaded when the import is

```
from PKG import *
```

Given the following package and module layout, the table on the next page describes how `__init__.py` affects imports.

```
my_package
|-----__init__.py
|-----module_a.py
|           function_a()
|-----module_b.py
|           function_b()
|-----module_c.py
|           function_c()
```

Import statement	What it does
If <code>__init__.py</code> is empty	
<code>import my_package</code>	Imports my_package only, but not contents. No modules are imported. This is not useful.
<code>import my_package.module_a</code>	Imports module_a into my_package namespace. Objects in module_a must be prefixed with my_package.module_a
<code>from my_package import module_a</code>	Imports module_a into main namespace. Objects in module_a must be prefixed with module_a
<code>from my_package import module_a, module_b</code>	Imports module_a and module_b into main namespace.
<code>from my_package import *</code>	Does not import anything!
<code>from my_package.module_a import *</code>	Imports all contents of module_a (that do not start with an underscore) into main namespace. Not generally recommended.
If <code>__init__.py</code> contains: <code>all = ['module_a', 'module_b']</code>	
<code>import my_package</code>	Imports my_package only, but not contents. No modules are imported. This is still not useful.
<code>from my_package import module_a</code>	As before, imports module_a into main namespace. Objects in module_a must be prefixed with module_a
<code>from my_package import *</code>	Imports module_a and module_b , but not module_c into main namespace.
If <code>__init__.py</code> contains: <code>all = ['module_a', 'module_b']</code> <code>import module_a</code> <code>import module_b</code>	
<code>import my_package</code>	Imports module_a and module_b into the my_package namespace. Objects in module_a must be prefixed with my_package.module_a . <i>Now this is useful.</i>
<code>from my_package import module_a</code>	Imports module_a into main namespace. Objects in module_a must be prefixed with module_a

Import statement	What it does
<code>from my_package import *</code>	Only imports module_a and module_b into main namespace.
<code>from my_package import module_c</code>	Imports module_c into the main namespace.

Documenting modules and packages

- Use docstrings
- Described in PEP 257
- Generate docs with Sphinx (optional)

In addition to comments, which are for the *maintainer* of your code, you should add docstrings, which provide documentation for the *user* of your code.

If the first statement in a module, function, or class is an unassigned string, it is assigned as the *docstring* of that object. It is stored in the special attribute `_doc_`, and so is available to code.

The docstring can use any form of literal string, but triple double quotes are preferred, for consistency.

See **PEP 257** for a detailed guide on docstring conventions.

Tools such as pydoc, and many IDEs will use the information in docstrings. In addition, the Sphinx tool will gather docstrings from an entire project and format them as a single HTML, PDF, or EPUB document.

Python style

- Code is read more often than it is written!
- Style guides enforce consistency and readability

- Indent 4 spaces (do not use tabs)
- Keep lines \leq 79 characters
- Imports at top of script, and on separate lines
- Surround operators with space
- Comment thoroughly to explain why and how code works when not obvious
- Use docstrings to explain how to use modules, classes, methods, and functions
- Use lower_case_with_underscores for functions, methods, and attributes
- Use UPPER_CASE_WITH_UNDERSCORES for globals
- Use StudlyCaps (mixed-case) for class names
- Use `_leading_underscore` for internal (non-API) attributes

Guido van Rossum, Python's BDFL (Benevolent Dictator For Life), once said that code is read much more often than it is written. This means that once code is written, it may be read by the original developer, users, subsequent developers who inherit your code. Do them a favor and make your code readable. This in turn makes your code more maintainable.

To make your code readable, it is import to write your code in a consistent manner. There are several Python style guides available, including PEP (Python Enhancement Proposal) 8, Style Guide for Python Code, and PEP 257, Docstring Conventions.

If you are part of a development team, it is a good practice to put together a style guide for the team. The team will save time not having to figure out each other's style.

Chapter 7 Exercises

Exercise 7-1 (potus.py, potus_main.py)

Create a module named to provide information from the presidents.txt file. It should provide the following function:

```
get_info(term#) -> dict    provide dictionary of info for a specified president
```

Write a script to use the module.

For the ambitious (potus_amb.py, potus_amb_main.py)

Add the following functions to the module

```
get_oldest() -> string    return the name of oldest president  
get_youngest()-> string   return the name of youngest president
```

'youngest' and 'oldest' refer to age at beginning of first term and age at end of last term.

Chapter 8: Errors and Exception Handling

Objectives

- Understanding syntax errors
- Handling exceptions with try-except-else-finally
- Learning the standard exception objects

Syntax errors

- Generated by the parser
- Cannot be trapped

Syntax errors are generated by the Python parser, and cause execution to stop (your script exits). They display the file name and line number where the error occurred, as well as an indication of where in the line the error occurred.

Because they are generated as soon as they are encountered, syntax errors may not be handled.

Example

```
File "<stdin>", line 1
  for x in bargle
              ^
SyntaxError: invalid syntax
```

TIP

When running in interactive mode, the filename is `<stdin>`.

Exceptions

- Generated when runtime errors occur
- Usually fatal if not handled

Even if code is syntactically correct, errors can occur. A common run-time error is to attempt to open a non-existent file. Such errors are called exceptions, and cause the interpreter to stop with an error message.

Python has a hierarchy of builtin exceptions; handling an exception higher in the tree will handle any children of that exception.

TIP Custom exceptions can be created by sub-classing the `Exception` object.

Example

`exception_unhandled.py`

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

x = 5
y = "cheese"

z = x + y ①
```

① Adding a string to an int raises **`TypeError`**

`exception_unhandled.py`

```
Traceback (most recent call last):
  File "exception_unhandled.py", line 6, in <module>
    z = x + y ①
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Handling exceptions with try

- Use try/except clauses
- Specify expected exception

To handle an exception, put the code which might generate an exception in a try block. After the try block, you must specify a except block with the expected exception. If an exception is raised in the try block, execution stops and the interpreter looks for the exception in the except block. If found, it executes the except block and execution continues; otherwise, the exception is treated as fatal and the interpreter exits.

Example

`exception_simple.py`

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

try: ①
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")

except TypeError as err: ②
    print("Naughty programmer! ", err)

print("After try-except") ③
```

- ① Execute code that might have a problem
- ② Catch the expected error; assign error object to **err**
- ③ Get here whether or not exception occurred

`exception_simple.py`

```
Naughty programmer! unsupported operand type(s) for +: 'int' and 'str'
After try-except
```

Handling multiple exceptions

- Use a tuple of exception names, but with single argument

If your try clause might generate more than one kind of exception, you can specify a tuple of exception types, then the variable which will hold the exception object.

Example

exception_multiple.py

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except (IOError, TypeError) as err: ①
    print("Naughty programmer! ", err)
```

- ① Use a tuple of 2 or more exception types

exception_multiple.py

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
```

Handling generic exceptions

- Use **Exception**
- Specify `except` with no exception list
- Clean up any uncaught exceptions

As a shortcut, you can specify **Exception** or an empty exception list. This will handle any exception that occurs in the `try` block.

Example

`exception_generic.py`

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except Exception as err: ①
    print("Naughty programmer! ", err)
```

① Will catch *any* exception

`exception_generic.py`

```
Naughty programmer!  unsupported operand type(s) for +: 'int' and 'str'
```

Ignoring exceptions

- Use the **pass** statement

Use the **pass** statement to do nothing when an exception occurs

Because the except clause must contain some code, the pass statement fulfills the syntax without doing anything.

Example

exception_ignore.py

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except(TypeError, IOError): ①
    pass
```

① Catch exceptions, and do nothing

exception_ignore.py

```
_no output_
```

This is probably a bad idea...

Using else

- executed if no exceptions were raised
- not required
- can make code easier to read

The last except block can be followed by an else block. The code in the else block is executed only if there were no exceptions raised in the try block. Exceptions in the else block are not handled by the preceding except blocks.

The else lets you make sure that some code related to the try clause (and before the finally clause) is only run if there's no exception, without trapping the exception specified in the except clause.

```
try:
    something_that_can_throw_ioerror()
except IOError as e:
    handle_the_IO_exception()
else:
    # we don't want to catch this IOError if it's raised
    something_else_that_throws_ioerror()
finally:
    something_we_always_need_to_do()
```


Example

exception_else.py

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

numpairs = [(5, 1), (1, 5), (5, 0), (0, 5)]

total = 0

for x, y in numpairs:
    try:
        quotient = x / y
    except Exception as err:
        print("uh-oh, when y = {}, {}".format(y, err))
    else:
        total += quotient ①
print(total)
```

① Only if no exceptions were raised

exception_else.py

```
uh-oh, when y = 0, division by zero
5.2
```

Cleaning up with finally

- Executed whether or not exception occurs
- Code executed whether or not exception raised

A finally block can be used instead of, or in addition to, an except block. The code in a finally block is executed whether or not an exception occurs. The finally block is executed after the try, except, and else blocks.

Example

exception_finally.py

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

try:
    x = 5
    y = 37
    z = x + y
    print("z is", z)
except TypeError as err: ①
    print("Caught exception:", err)
finally:
    print("Don't care whether we had an exception") ②

print()

try:
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")
except TypeError as err:
    print("Caught exception:", err)
finally:
    print("Still don't care whether we had an exception")
```

① Catch **TypeError**

② Print whether **TypeError** is caught or not

exception_finally.py

```
z is 42
Don't care whether we had an exception

Caught exception: unsupported operand type(s) for +: 'int' and 'str'
Still don't care whether we had an exception
```

The Standard Exception Hierarchy (Python 3.7)

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |   |   +-- BrokenPipeError
    |   |   +-- ConnectionAbortedError
    |   |   +-- ConnectionRefusedError
    |   |   +-- ConnectionResetError
    |   +-- FileExistsError
    |   +-- FileNotFoundError
    |   +-- InterruptedError
    |   +-- IsADirectoryError
    |   +-- NotADirectoryError
    |   +-- PermissionError
    |   +-- ProcessLookupError
    |   +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
```

```
|    +-- NotImplementedError
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|        +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|        +-- UnicodeDecodeError
|        +-- UnicodeEncodeError
|        +-- UnicodeTranslateError
```

Chapter 8 Exercises

Exercise 8-1 (`c2f_safe.py`)

Rewrite `c2f.py` to handle the error that occurs if the user enters non-numeric data.

Exercise 8-2 (`c2f_batch_safe.py`)

Rewrite `c2f_batch.py` to handle the `ValueError` that occurs if `sys.argv[1]` is not a valid number.

Chapter 9: Using the Standard Library

Objectives

- Overview of the standard library
- Getting information on the Python interpreter's environment
- Running external programs
- Walking through a directory tree
- Working with path names
- Calculating dates and times
- Fetching data from a URL
- Generating random values

The sys module

- Import the sys module to provide access to the interpreter and its environment
- Get interpreter attributes
- Interact with the operating system

The sys module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

This module provides details of the current Python interpreter; it also provides objects and methods to interact with the operating system.

Even though the sys module is built into the Python interpreter, it must be imported like any other module.

Interpreter Information

- sys provides details of interpreter

To get the folder where Python is installed, use **sys.prefix**.

To get the path to the Python executable, use **sys.executable**.

To get a version string, use **sys.version**.

To get the details of the interpreter as a tuple, use **sys.version_info**.

To get the list of directories that will be searched for modules, examine **sys.path**.

To get a list of currently loaded modules, use **sys.modules**.

To find out what platform (OS/architecture) the script is running on, use **sys.platform**.

STDIO

- `stdin`
- `stdout`
- `stderr`

The `sys` object defines three file objects representing the three streams of STDIO, or "standard I/O".

Unless they have been redirected, `sys.stdin` is the keyboard, and `sys.stdout` and `sys.stderr` are the console screen. You should use `sys.stderr` for error messages.

Example

`stdio.py`

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

import sys
sys.stdout.write("Hello, world\n")
sys.stderr.write("Error message here...\n")
```

`stdio.py 2>spam.txt`

```
Hello, world
```

`type spam.txt` windows

`cat spam.txt` non-windows

```
Error message here...
```

Launching external programs

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

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

`os.system()` launches any external command, as though you had typed it at a command prompt. `popen()` opens a pipe to a command so you can read the output of the command one line at a time. `popen()` is very similar to the `open()` function; it returns an iterable object.

For more control over external processes, use the `subprocess` module (part of the standard library), or check out the `sh` module (not part of the standard library).

Example

external_programs.py

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

os.system("hostname") ①

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

- ① Just run "hostname"
- ② Open command line "netstat -an" as a file-like object
- ③ Iterate over lines in output of "netstat -an"
- ④ Check to see if line contains "ESTAB"

external_programs.py

```
MacBook-Pro-8.attlocal.net
tcp4      0      0 192.168.1.66.57327    52.202.71.214.443    ESTABLISHED
tcp6      564    0 2600:1700:3901:6.57326 2607:f8b0:4000:8.443  ESTABLISHED
tcp4      0      0 192.168.1.66.57321    52.24.6.146.443      ESTABLISHED
tcp4      0      0 192.168.1.66.57320    13.249.98.26.443     ESTABLISHED
tcp6      0      0 2600:1700:3901:6.57319 2600:9000:21f9:1.443  ESTABLISHED
tcp4      31     0 192.168.1.66.57310    3.216.6.184.443      ESTABLISHED
tcp4      0      0 192.168.1.66.57306    54.164.252.155.443   ESTABLISHED
tcp6      0      0 2600:1700:3901:6.57305 2620:108:700f::3.443  ESTABLISHED
tcp4      0      0 192.168.1.66.57304    18.209.37.141.443    ESTABLISHED
tcp4      0      0 192.168.1.66.57303    34.225.190.8.443     ESTABLISHED
tcp4      0      0 192.168.1.66.57302    184.30.19.222.443    ESTABLISHED
tcp4      0      0 192.168.1.66.57300    23.218.88.202.443    ESTABLISHED
tcp4      0      0 192.168.1.66.57299    54.164.10.49.443     ESTABLISHED
tcp4      0      0 192.168.1.66.57298    185.167.164.37.443   ESTABLISHED
```

Paths, directories and filenames

- import `os.path` module
- `path` is mapped to appropriate package for current os
- The `os.path` module provides many functions for working with paths.
- Some of the more common methods:
 - `os.path.exists()`
 - `os.path.dirname()`
 - `os.path.basename`
 - `os.path.split()`

os.path is the primary module for working with filenames and paths. There are many methods for getting and modifying a file or folder's path.

Also provide are methods for getting information about a file.

Example

paths.py

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

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

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

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

- ① Unix relative path
- ② Unix absolute path
- ③ Windows relative path
- ④ Windows UNC path
- ⑤ What platform are we on?
- ⑥ Just the folder name
- ⑦ Just the file (or folder) name
- ⑧ Is it an absolute path?
- ⑨ ~ is current user's home
- ⑩ ~NAME is NAME's home

paths.py

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

paths.py (windows)

```
dirname(win_p1): \\marmoset\sharing\technology\docs\bonsai
dirname(win_p2): \\marmoset\sharing\technology\docs\bonsai
basename(win_p1): foo.doc
basename(win_p2): foo.doc
os.path.split(win_p1) Head: \\marmoset\sharing\technology\docs\bonsai Tail: foo.doc
os.path.split(win_p1) Head: bonsai Tail: foo.doc
os.path.splitunc(win_p1) Head: \\marmoset\sharing Tail:
\technology\docs\bonsai\foo.doc
os.path.splitunc(win_p1) Head: Tail: bonsai\foo.doc
```

Walking directory trees

- Import os module
- Use the os.walk() iterator
- Returns tuple for each directory starting with the specified top directory
- Tuple contains full path to directory, list of subdirectories, and list of files *syntax:

```
for currdir,subdirs,files in os.walk("start-dir"):
    pass
```

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

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

TIP

Remember to not use "dir" or "file" as variables when looping through the iterator, because they will overwrite builtins.

Example

walk.py

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

# count number of files and dirs in a directory tree
# note "files" includes devices, symbolic links, and pipes
import os
import sys

if sys.platform == 'win32': ①
    target = 'C:/Windows'
else:
    target = '/etc'

total_files = 0 ②
total_dirs = 0

for currdir, subdirs, files in os.walk(target): ③
    total_dirs += 1 # increment number of directories seen
    total_files += len(files) # add the number of files in this dir

print("{} contains {} dirs and {} files".format(target, total_dirs, total_files))
④
```

walk.py

```
/etc contains 41 dirs and 351 files
```

walk2.py

```
#!/usr/bin/env python
"""
    find files whose size is greater than or equal to specified number of bytes
"""
import sys
import os

MINIMUM_SIZE = 1000

if len(sys.argv) < 2: ①
    print('Syntax: walk2.py START-DIR')
    sys.exit(1)

for currdir, subdirs, files in os.walk(sys.argv[1]):
    for file in files: ②
        fullpath = os.path.join(currdir, file) ③
        if os.path.isfile(fullpath): ④
            fsize = os.path.getsize(fullpath) ⑤
            if fsize >= MINIMUM_SIZE: ⑥
                print("{:40s} {:8d}".format(fullpath, fsize))
```

walk2.py

./xml_from_presidents.py	2458
./test_pytest_setup_teardown.py	1161
./paramiko_remote_cmd.py	1272
./sa_movie_models.py	1349
./example.zip	56343
./xml_create_knights.py	1396
./threadexample.ui	2808
./pylintrc	14755
./db_mysql_metadata.py	1431
./new_magic.py	1387

...

Grabbing data from the web

- import module urllib
- urlopen() similar to open()
- Iterate through (or read from) URL object
- Use info() method for metadata

Python makes grabbing web pages easy with the urllib module. The urllib.request.urlopen() method returns an HTTP response object (which also acts like a file object).

Iterating through this object returns the lines in the specified web page (the same lines you would see with "view source" in a browser).

Since the URL is opened in binary mode; you can use *response.read()* to download any kind of file which a URL represents – PDF, MP3, JPG, and so forth.

NOTE

Grabbing web pages is even easier with the **requests** modules. See [read_html_requests.py](#) and [read_pdf_requests.py](#) in the EXAMPLES folder.

Example

read_html_urllib.py

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

import urllib.request

u = urllib.request.urlopen("https://www.python.org")

print(u.info()) ①
print()

print(u.read(500).decode()) ②
```

① .info() returns a dictionary of HTTP headers

② The text is returned as a bytes object, so it needs to be decoded to a string

read_html_urllib.py

```

Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur
Via: 1.1 varnish
Content-Length: 48643
Accept-Ranges: bytes
Date: Wed, 16 Oct 2019 12:21:27 GMT
Via: 1.1 varnish
Age: 956
Connection: close
X-Served-By: cache-iad2129-IAD, cache-fty21350-FTY
X-Cache: HIT, HIT
X-Cache-Hits: 5, 1
X-Timer: S1571228487.319118,VS0,VE6
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains

```

```

<!doctype html>
<!--[if lt IE 7]> <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9"> <![endif]-->
<!--[if IE 7]> <html class="no-js ie7 lt-ie8 lt-ie9"> <![endif]-->
<!--[if IE 8]> <html class="no-js ie8 lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr"> <!--<![endif]-->

<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">

```

...

Example

read_pdf_urllib.py

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

import sys
import os
from urllib.request import urlopen
from urllib.error import HTTPError

# url to download a PDF file of a NASA ISS brochure

url = 'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure%202012%20Screenres%203-8-13.pdf' ①

saved_pdf_file = 'nasa_iss.pdf' ②

try:
    URL = urlopen(url) ③
except HTTPError as e: ④
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read() ⑤
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents) ⑥

if sys.platform == 'win32': ⑦
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd) ⑧
```

Sending email

- use `smtplib`
- For attachments, use `email.mime.*`
- Can provide authentication
- Can work with proxies

It is easy to send a simple email message with Python. The `smtplib` module allows you to create and send the message.

To send an attachment, use `smtplib` plus one or more of the submodules of `email.mime`, which are needed to put the message and attachments in proper MIME format.

TIP

When sending attachments, be sure to use the `.as_string()` method on the MIME message object. Otherwise you will be sending binary gibberish to your recipient.

Example

email_simple.py

```
#!/usr/bin/env python
from getpass import getpass ①
import smtplib ②
from email.message import EmailMessage ③
from datetime import datetime

TIMESTAMP = datetime.now().ctime() ④

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']
MESSAGE_SUBJECT = 'Python SMTP example'

MESSAGE_BODY = """
Hello at {}.

Testing email from Python
""".format(TIMESTAMP)

SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:") ⑤

smtpserver = smtplib.SMTP("smtp2go.com", 2525) ⑥
smtpserver.login(SMTP_USER, SMTP_PASSWORD) ⑦

msg = EmailMessage() ⑧
msg.set_content(MESSAGE_BODY) ⑨
msg['Subject'] = MESSAGE_SUBJECT ⑩
msg['from'] = SENDER ⑪
msg['to'] = RECIPIENTS ⑫

try:
    smtpserver.send_message(msg) ⑬
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtpserver.quit() ⑭
```

① module for hiding password

- ② module for sending email
- ③ module for creating message
- ④ get a time string for the current date/time
- ⑤ get password (not echoed to screen)
- ⑥ connect to SMTP server
- ⑦ log into SMTP server
- ⑧ create empty email message
- ⑨ add the message body
- ⑩ add the message subject
- ⑪ add the sender address
- ⑫ add a list of recipients
- ⑬ send the message
- ⑭ disconnect from SMTP server

email_attach.py

```
#!/usr/bin/env python
import smtplib
from datetime import datetime
from imghdr import what ①
from email.message import EmailMessage ②
from getpass import getpass ③

SMTP_SERVER = "smtp2go.com" ④
SMTP_PORT = 2525

SMTP_USER = 'pythonclass'

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']

def main():
    smtp_server = create_smtp_server()
    now = datetime.now()
    msg = create_message(
        SENDER,
        RECIPIENTS,
        'Here is your attachment',
        'Testing email attachments from python class at {}'.format(now),
    )
    add_text_attachment('../DATA/parrot.txt', msg)
    add_image_attachment('../DATA/felix_auto.jpeg', msg)
    send_message(smtp_server, msg)

def create_message(sender, recipients, subject, body):
    msg = EmailMessage() ⑤
    msg.set_content(body) ⑥
    msg['From'] = sender
    msg['To'] = recipients
    msg['Subject'] = subject
    return msg

def add_text_attachment(file_name, message):
```

```
with open(file_name) as file_in: ⑦
    attachment_data = file_in.read()
message.add_attachment(attachment_data) ⑧

def add_image_attachment(file_name, message):
    with open(file_name, 'rb') as file_in: ⑨
        attachment_data = file_in.read()
    image_type = what(None, h=attachment_data) ⑩
    message.add_attachment(attachment_data, maintype='image', subtype=image_type)
    ⑪

def create_smtp_server():
    password = getpass("Enter SMTP server password:") ⑫
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT) ⑬
    smtpserver.login(SMTP_USER, password) ⑭

    return smtpserver

def send_message(server, message):
    try:
        server.send_message(message) ⑮
    finally:
        server.quit()

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

- ① module to determine image type
- ② module for creating email message
- ③ module for reading password privately
- ④ global variables for external information (IRL should be from environment — command line, config file, etc.)
- ⑤ create instance of EmailMessage to hold message
- ⑥ set content (message text) and various headers
- ⑦ read data for text attachment
- ⑧ add text attachment to message
- ⑨ read data for binary attachment
- ⑩ get type of binary data
- ⑪ add binary attachment to message, including type and subtype (e.g., "image/jpg")
- ⑫ get password from user (don't hardcode sensitive data in script)
- ⑬ create SMTP server connection
- ⑭ log into SMTP connection
- ⑮ send message

math functions

- use the math module
- Provides functions and constants

Python provides many math functions. It also provides constants pi and e.

Table 14. Math functions

<code>sqrt(x)</code>	Returns the square root of x
<code>exp(x)</code>	Return e^x
<code>log(x)</code>	Returns the natural log, i.e. $\ln x$
<code>log10(x)</code>	Returns the log to the base 10 of x
<code>sin(x)</code>	Returns the sine of x
<code>cos(x)</code>	Return the cosine of x
<code>tan(x)</code>	Returns the tangent of x
<code>asin(x)</code>	Return the arc sine of x
<code>acos(x)</code>	Return the arc cosine of x
<code>atan(x)</code>	Return the arc tangent of x
<code>fabs(x)</code>	Return the absolute value, i.e. the modulus, of x
<code>ceil(x)</code>	Rounds x (which is a float) up to next highest integer ¹
<code>floor(x)</code>	Rounds x (which is a float) down to next lowest integer
<code>degrees(x)</code>	converts angle x from radians to degrees
<code>radians(x)</code>	converts angle x from degrees to radians

TIP

This table is not comprehensive – see [docs for math module](#) for some more functions.

For more math and engineering functions, see the external modules `numpy` and `scipy`.

Random values

- Use the random module
- Useful methods
 - `random()`
 - `randint(start,stop)`
 - `randrange(start,limit)`
 - `choice(seq)`
 - `sample(seq,count)`
 - `shuffle(seq)`

The random module provides methods based on selected a random number. In addition to `random()`, which returns a fractional number between 0 and 1, there are a number of convenience functions.

`randint()` and `randrange()` return a random integer within a range of numbers; the difference is that `randint()` includes the endpoint of the specified range, and `randrange()` does not.

`choice()` returns one element from any of Python's sequence types; `sample()` is the same, but returns a specified number of elements.

`shuffle()` randomizes a sequence.

Example

random_ex.py

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

import random

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
"lemon", "kiwi", "orange", "lime", "watermelon", "guava",
"papaya", "fig", "pear", "banana", "tamarind", "persimmon",
"elderberry", "peach", "blueberry", "lychee", "grape" ]

for i in range(1, 11):
    print("random():", random.random())
    print("randint(1, 2000):", random.randint(1, 2000))
    print("randrange(1, 5):", random.randrange(1, 5))
    print("choice(fruit):", random.choice(fruits))
    print("sample(fruit, 3):", random.sample(fruits, 3))
    print()
```


random_ex.py

```
random(): 0.864982924168403
randint(1, 2000): 1842
randrange(1, 5): 4
choice(fruit): peach
sample(fruit, 3): ['guava', 'persimmon', 'pear']

random(): 0.8273879082502135
randint(1, 2000): 479
randrange(1, 5): 2
choice(fruit): pomegranate
sample(fruit, 3): ['watermelon', 'lime', 'lychee']

random(): 0.5623322392481583
randint(1, 2000): 128
randrange(1, 5): 1
choice(fruit): watermelon
sample(fruit, 3): ['date', 'tamarind', 'persimmon']

random(): 0.21048265151407475
randint(1, 2000): 920
randrange(1, 5): 1
choice(fruit): papaya
sample(fruit, 3): ['elderberry', 'papaya', 'apricot']

random(): 0.15090404245311984
randint(1, 2000): 1090
randrange(1, 5): 3
choice(fruit): persimmon
sample(fruit, 3): ['persimmon', 'guava', 'pear']
```

Dates and times

- Use the datetime module
- Provides several classes
 - datetime
 - date
 - time
 - timedelta

Python provides the datetime module for manipulating dates and times. Once you have created date or time objects, you can combine them and extract the time units you need.

Example

datetime_ex.py

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

from datetime import datetime, date, timedelta

print("date.today():", date.today()) ①

now = datetime.now() ②
print("now.day:", now.day) ③
print("now.month:", now.month)
print("now.year:", now.year)
print("now.hour:", now.hour)
print("now.minute:", now.minute)
print("now.second:", now.second)

d1 = datetime(2018, 6, 13) ④
d2 = datetime(2018, 8, 24)

d3 = d2 - d1 ⑤

print("raw time delta:", d3)
print("time delta days:", d3.days) ⑥

interval = timedelta(10) ⑦
print("interval:", interval)

d4 = d2 + interval ⑧
d5 = d2 - interval
print("d2 + interval:", d4)
print("d2 - interval:", d5)
print()

t1 = datetime(2016, 8, 24, 10, 4, 34) ⑨
t2 = datetime(2018, 8, 24, 22, 8, 1)
t3 = t2 - t1

print("datetime(2016, 8, 24, 10, 4, 34):", t1)
print("datetime(2018, 8, 24, 22, 8, 1):", t2)
print("time diff (t2 - t1):", t3)
```

datetime_ex.py

```
date.today(): 2019-10-16
now.day: 16
now.month: 10
now.year: 2019
now.hour: 8
now.minute: 21
now.second: 27
raw time delta: 72 days, 0:00:00
time delta days: 72
interval: 10 days, 0:00:00
d2 + interval: 2018-09-03 00:00:00
d2 - interval: 2018-08-14 00:00:00

datetime(2016, 8, 24, 10, 4, 34): 2016-08-24 10:04:34
datetime(2018, 8, 24, 22, 8, 1): 2018-08-24 22:08:01
time diff (t2 - t1): 730 days, 12:03:27
```

Zipped archives

- import zipfile for (PK)zipped files
- Get a list of files
- Extract files

The zipfile module allows you to read and write to zipped archives. In either case you first create a zipfile object; specifying a mode of "w" if you want to create an archive, and a mode of "r" (or nothing) if you want to read an existing zip file.

There are also modules for gzipped, bziped, and compressed archives.

Example

zipfile_ex.py

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

from zipfile import ZipFile, ZIP_DEFLATED
import os.path

# reading & extracting
rzip = ZipFile("../DATA/textfiles.zip") ①
print(rzip.namelist()) ②
ty = rzip.read('tyger.txt').decode() ③
print(ty[:50])
rzip.extract('parrot.txt') ④

# creating a zip file
wzip = ZipFile("example.zip", mode="w", compression=ZIP_DEFLATED) ⑤
for base in "parrot tyger knights alice poe_sonnet spam".split():
    filename = os.path.join("../DATA", base + '.txt')
    print("adding {} as {}".format(filename, base + '.txt'))
    wzip.write(filename, base + '.txt') ⑥
```

- ① Open zip file for reading
- ② Print list of members in zip file
- ③ Read (raw binary) data from member and convert from bytes to string
- ④ Extract member
- ⑤ Create new zip file
- ⑥ Add member to zip file

zipfile_ex.py

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']  
The Tyger
```

```
Tyger! Tyger! burning bright  
adding ../DATA/parrot.txt as parrot.txt  
adding ../DATA/tyger.txt as tyger.txt  
adding ../DATA/knights.txt as knights.txt  
adding ../DATA/alice.txt as alice.txt  
adding ../DATA/poe_sonnet.txt as poe_sonnet.txt  
adding ../DATA/spam.txt as spam.txt
```

Chapter 9 Exercises

Exercise 9-1 (`print_sys_info.py`)

Use the module `os` to print out the pathname separator, the `PATH` variable separator, and the extension separator for your OS.

Exercise 9-2 (`file_size.py`)

Write a script that accepts one or more files on the command line, and prints out the size, one file per line. If any argument is not a file, print out an error message.

TIP | [You will need the `os.path` module.](#)

Chapter 10: Pythonic Programming

Objectives

- Learn what makes code "Pythonic"
- Understand some Python-specific idioms
- Create lambda functions
- Perform advanced slicing operations on sequences
- Distinguish between collections and generators

The Zen of Python

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

— Tim Peters, from PEP 20

Tim Peters is a longtime contributor to Python. He wrote the standard sorting routine, known as "timsort".

The above text is printed out when you execute the code "import this". Generally speaking, if code follows the guidelines in the Zen of Python, then it's Pythonic.

Tuples

- Fixed-size, read-only
- Collection of related items
- Supports some sequence operations
- Think 'struct' or 'record'

A **tuple** is a collection of related data. While on the surface it seems like just a read-only list, it is used when you need to pass multiple values to or from a function, but the values are not all the same type

To create a tuple, use a comma-separated list of objects. Parentheses are not needed around a tuple unless the tuple is nested in a larger data structure.

A tuple in Python might be represented by a struct or a "record" in other languages.

While both tuples and lists can be used for any data:

- Use a list when you have a collection of similar objects.
- Use a tuple when you have a collection of related objects, which may or may not be similar.

TIP

To specify a one-element tuple, use a trailing comma, otherwise it will be interpreted as a single object: `color = 'red',`

Example

```
hostinfo = ( 'gemini','linux','ubuntu','hardy','Bob Smith' )  
  
birthday = ( 'April',5,1978 )
```

Iterable unpacking

- Copy iterable to list of variables
- Can have one wild card
- Frequently used with list of tuples

When you have an iterable such as a tuple or list, you access individual elements by index. However, `spam[0]` and `spam[1]` are not so readable compared to `first_name` and `company`. To copy an iterable to a list of variable names, just assign the iterable to a comma-separated list of names:

```
birthday = ( 'April',5,1978 )  
month, day, year = birthday
```

You may be thinking "why not just assign to the variables in the first place?". For a single tuple or list, this would be true. The power of unpacking comes in the following areas:

- Looping over a sequence of tuples
- Passing tuples (or other iterables) into a function

Example

unpacking_people.py

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

people = [ ①
    ('Melinda', 'Gates', 'Gates Foundation'),
    ('Steve', 'Jobs', 'Apple'),
    ('Larry', 'Wall', 'Perl'),
    ('Paul', 'Allen', 'Microsoft'),
    ('Larry', 'Ellison', 'Oracle'),
    ('Bill', 'Gates', 'Microsoft'),
    ('Mark', 'Zuckerberg', 'Facebook'),
    ('Sergey', 'Brin', 'Google'),
    ('Larry', 'Page', 'Google'),
    ('Linus', 'Torvalds', 'Linux'),
]

for first_name, last_name, org in people: ②
    print("{} {}".format(first_name, last_name))
```

① A list of 3-element tuples

② The for loop unpacks each tuple into the three variables.

unpacking_people.py

```
Melinda Gates
Steve Jobs
Larry Wall
Paul Allen
Larry Ellison
Bill Gates
Mark Zuckerberg
Sergey Brin
Larry Page
Linus Torvalds
```

Unpacking function arguments

- Go from iterable to list of items
- Use * or **

Sometimes you need the other end of iterable unpacking. What do you do if you have a list of three values, and you want to pass them to a method that expects three positional arguments? One approach is to use the individual items by index. A more Pythonic approach is to use * to *unpack* the iterable into individual items:

Use a single asterisk to unpack a list or tuple (or similar iterable); use two asterisks to unpack a dictionary or similar.

In the example, see how the list **HEADINGS** is passed to `.format()`, which expects individual parameters, not *one parameter* containing multiple values.

Example

unpacking_function_args.py

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

people = [ ①
    ('Joe', 'Schmoe', 'Burbank', 'CA'),
    ('Mary', 'Rattburger', 'Madison', 'WI'),
    ('Jose', 'Ramirez', 'Ames', 'IA'),
]

def person_record(first_name, last_name, city, state): ②
    print("{} {} lives in {}, {}".format(first_name, last_name, city, state))

for person in people: ③
    person_record(*person) ④
```

- ① list of 4-element tuples
- ② function that takes 4 parameters
- ③ person is a tuple (one element of people list)
- ④ ***person** unpacks the tuple into four individual parameters

unpacking_function_args.py

```
Joe Schmoe lives in Burbank, CA
Mary Rattburger lives in Madison, WI
Jose Ramirez lives in Ames, IA
```

Example

shoe_sizes.py

```
#!/usr/bin/env python
#
BARLEYCORN = 1 / 3.0
CM_TO_INCH = 2.54
MENS_START_SIZE = 12
WOMENS_START_SIZE = 10.5

FMT = '{:6.1f} {:8.2f} {:8.2f}'
HEADFMT = '{:>6s} {:>8s} {:>8s}'

HEADINGS = ['Size', 'Inches', 'CM']

SIZE_RANGE = []
for i in range(6, 14):
    SIZE_RANGE.extend([i, i + .5])

def main():
    for heading, flag in [("MEN'S", True), ("WOMEN'S", False)]:
        print(heading)
        print((HEADFMT.format(*HEADINGS))) ①
        for size in SIZE_RANGE:
            inches, cm = get_length(size, flag)
            print(FMT.format(size, inches, cm))

        print()

def get_length(size, mens=True):
    if mens:
        start_size = MENS_START_SIZE
    else:
        start_size = WOMENS_START_SIZE

    inches = start_size - ((start_size - size) * BARLEYCORN)
    cm = inches * CM_TO_INCH
    return inches, cm

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


- ① `format` expects individual arguments for each placeholder; the asterisk unpacks HEADINGS into individual strings

shoe_sizes.py

```
MEN'S
Size  Inches      CM
6.0   10.00    25.40
6.5   10.17    25.82
7.0   10.33    26.25
7.5   10.50    26.67
8.0   10.67    27.09
8.5   10.83    27.52
```

...

The sorted() function

- Returns a sorted copy of any collection
- Customize with named keyword parameters

```
key=  
reverse=
```

The sorted() builtin function returns a sorted copy of its argument, which can be any iterable.

You can customize sorted with the **key** parameter.

Example

basic_sorting.py

```
#!/usr/bin/env python  
  
"""Basic sorting example"""  
  
fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",  
         "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear", "banana",  
         "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry", "lychee",  
         "grape"]  
  
sorted_fruit = sorted(fruit) ①  
  
print(sorted_fruit)
```

① sorted() returns a list

basic_sorting.py

```
['Apple', 'BLUEberry', 'FIG', 'Kiwi', 'ORANGE', 'Tamarind', 'Watermelon', 'apricot',  
'banana', 'cherry', 'date', 'elderberry', 'grape', 'guava', 'lemon', 'lime',  
'lychee', 'papaya', 'peach', 'pear', 'persimmon', 'pomegranate']
```

Custom sort keys

- Use **key** parameter
- Specify name of function to use
- Key function takes exactly one parameter
- Useful for case-insensitive sorting, sorting by external data, etc.

You can specify a function with the **key** parameter of the `sorted()` function. This function will be used once for each element of the list being sorted, to provide the comparison value. Thus, you can sort a list of strings case-insensitively, or sort a list of zip codes by the number of Starbucks within the zip code.

The function must take exactly one parameter (which is one element of the sequence being sorted) and return either a single value or a tuple of values. The returned values will be compared in order.

You can use any builtin Python function or method that meets these requirements, or you can write your own function.

TIP

The `lower()` method can be called directly from the builtin object `str`. It takes one string argument and returns a lower case copy.

```
sorted_strings = sorted(unsorted_strings, key=str.lower)
```

Example

custom_sort_keys.py

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

fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon",
         "Kiwi", "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG",
         "pear", "banana", "Tamarind", "persimmon", "elderberry", "peach",
         "BLUEberry", "lychee", "grape"]

def ignore_case(item): ①
    return item.lower() ②

fs1 = sorted(fruit, key=ignore_case) ③
print("Ignoring case:")
print(" ".join(fs1), end="\n\n")

def by_length_then_name(item):
    return (len(item), item.lower()) ④

fs2 = sorted(fruit, key=by_length_then_name)
print("By length, then name:")
print(" ".join(fs2))
print()

nums = [800, 80, 1000, 32, 255, 400, 5, 5000]

n1 = sorted(nums) ⑤
print("Numbers sorted numerically:")
for n in n1:
    print(n, end=' ')
print("\n")

n2 = sorted(nums, key=str) ⑥
print("Numbers sorted as strings:")
for n in n2:
    print(n, end=' ')
print()
```

- ① Parameter is *one* element of iterable to be sorted
- ② Return value to sort on
- ③ Specify function with named parameter **key**
- ④ Key functions can return tuple of values to compare, in order
- ⑤ Numbers sort numerically by default
- ⑥ Sort numbers as strings

custom_sort_keys.py

Ignoring case:

Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon
lime lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon

By length, then name:

FIG date Kiwi lime pear Apple grape guava lemon peach banana cherry lychee ORANGE
papaya apricot Tamarind BLUEberry persimmon elderberry Watermelon pomegranate

Numbers sorted numerically:

5 32 80 255 400 800 1000 5000

Numbers sorted as strings:

1000 255 32 400 5 5000 80 800

Example

sort_holmes.py

```
#!/usr/bin/env python
"""Sort titles, ignoring leading articles"""
import re

books = [
    "A Study in Scarlet",
    "The Sign of the Four",
    "The Hound of the Baskervilles",
    "The Valley of Fear",
    "The Adventures of Sherlock Holmes",
    "The Memoirs of Sherlock Holmes",
    "The Return of Sherlock Holmes",
    "His Last Bow",
    "The Case-Book of Sherlock Holmes",
]

rx_article = re.compile(r'^(the|a|an)\s+', re.I) ①

def strip_articles(title): ②
    stripped_title = rx_article.sub('', title.lower()) ③
    return stripped_title

for book in sorted(books, key=strip_articles): ④
    print(book)
```

① compile regex to match leading articles

② create function which takes element to compare and returns comparison key

③ strip off article and convert title to lower case

④ sort using custom function

sort_holmes.py

```
The Adventures of Sherlock Holmes
The Case-Book of Sherlock Holmes
His Last Bow
The Hound of the Baskervilles
The Memoirs of Sherlock Holmes
The Return of Sherlock Holmes
The Sign of the Four
A Study in Scarlet
The Valley of Fear
```

Lambda functions

- Short cut function definition
- Useful for functions only used in one place
- Frequently passed as parameter to other functions
- Function body is an expression; it cannot contain other code

A **lambda function** is a brief function definition that makes it easy to create a function on the fly. This can be useful for passing functions into other functions, to be called later. Functions passed in this way are referred to as "callbacks". Normal functions can be callbacks as well. The advantage of a lambda function is solely the programmer's convenience. There is no speed or other advantage.

One important use of lambda functions is for providing sort keys; another is to provide event handlers in GUI programming.

The basic syntax for creating a lambda function is

```
lambda parameter-list: expression
```

where parameter-list is a list of function parameters, and expression is an expression involving the parameters. The expression is the return value of the function.

A lambda function could also be defined in the normal manner

```
def function-name(param-list):  
    return expr
```

But it is not possible to use the normal syntax as a function parameter, or as an element in a list.

Example

lambda_examples.py

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

fruits = ['watermelon', 'Apple', 'Mango', 'KIWI', 'apricot', 'LEMON', 'guava']

sfruits = sorted(fruits, key=lambda e: e.lower()) ①

print(" ".join(sfruits))
```

① The lambda function takes one fruit and returns it in lower case

lambda_examples.py

```
Apple apricot guava KIWI LEMON Mango watermelon
```

List comprehensions

- Filters or modifies elements
- Creates new list
- Shortcut for a for loop

A list comprehension is a Python idiom that creates a shortcut for a for loop. It returns a copy of a list with every element transformed via an expression. Functional programmers refer to this as a mapping function.

A loop like this:

```
results = []
for var in sequence:
    results.append(expr)    # where expr involves var
```

can be rewritten as

```
results = [ expr for var in sequence ]
```

A conditional if may be added to filter values:

```
results = [ expr for var in sequence if expr ]
```

Example

listcomp.py

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

fruits = ['watermelon', 'apple', 'mango', 'kiwi', 'apricot', 'lemon', 'guava']

values = [2, 42, 18, 92, "boom", ['a', 'b', 'c']]

ufruits = [fruit.upper() for fruit in fruits] ①

afruits = [fruit for fruit in fruits if fruit.startswith('a')] ②

doubles = [v * 2 for v in values] ③

print("ufruits:", " ".join(ufruits))
print("afruits:", " ".join(afruits))
print("doubles:", end=' ')
for d in doubles:
    print(d, end=' ')
print()
```

- ① Copy each fruit to upper case
- ② Select each fruit if it starts with 'a'
- ③ Copy each number, doubling it

listcomp.py

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
afruits: apple apricot
doubles: 4 84 36 184 boomboom ['a', 'b', 'c', 'a', 'b', 'c']
```

Dictionary comprehensions

- Expression is key/value pair
- Transform iterable to dictionary

A dictionary comprehension has syntax similar to a list comprehension. The expression is a key:value pair, and is added to the resulting dictionary. If a key is used more than once, it overrides any previous keys. This can be handy for building a dictionary from a sequence of values.

Example

dict_comprehension.py

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

animals = ['OWL', 'Badger', 'bushbaby', 'Tiger', 'Wombat', 'GORILLA', 'AARDVARK']

d = {a.lower(): len(a) for a in animals} ①

print(d, '\n')

words = ['unicorn', 'stigmata', 'barley', 'bookkeeper']

d = {w:{c:w.count(c) for c in sorted(w)} for w in words} ②

for word, word_signature in d.items():
    print(word, word_signature)
```

- ① Create a dictionary with key/value pairs derived from an iterable
- ② Use a nested dictionary comprehension to create a dictionary mapping words to dictionaries which map letters to their counts (could be useful for anagrams)

dict_comprehension.py

```
{'owl': 3, 'badger': 6, 'bushbaby': 8, 'tiger': 5, 'wombat': 6, 'gorilla': 7,  
'aardvark': 8}
```

```
unicorn {'c': 1, 'i': 1, 'n': 2, 'o': 1, 'r': 1, 'u': 1}
```

```
stigmata {'a': 2, 'g': 1, 'i': 1, 'm': 1, 's': 1, 't': 2}
```

```
barley {'a': 1, 'b': 1, 'e': 1, 'l': 1, 'r': 1, 'y': 1}
```

```
bookkeeper {'b': 1, 'e': 3, 'k': 2, 'o': 2, 'p': 1, 'r': 1}
```

Set comprehensions

- Expression is added to set
- Transform iterable to set — with modifications

A set comprehension is useful for turning any sequence into a set. Items can be modified or skipped as the set is built.

If you don't need to modify the items, it's probably easier to just pass the sequence to the `set()` constructor.

Example

set_comprehension.py

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

import re

with open("../DATA/mary.txt") as mary_in:
    s = {w.lower() for ln in mary_in for w in re.split(r'\W+', ln) if w} ①
print(s)
```

① Get unique words from file. Only one line is in memory at a time. Skip "empty" words.

set_comprehension.py

```
{'go', 'and', 'was', 'as', 'little', 'fleece', 'to', 'had', 'lamb', 'sure', 'went',
'snow', 'everywhere', 'a', 'that', 'its', 'mary', 'white', 'the'}
```

Iterables

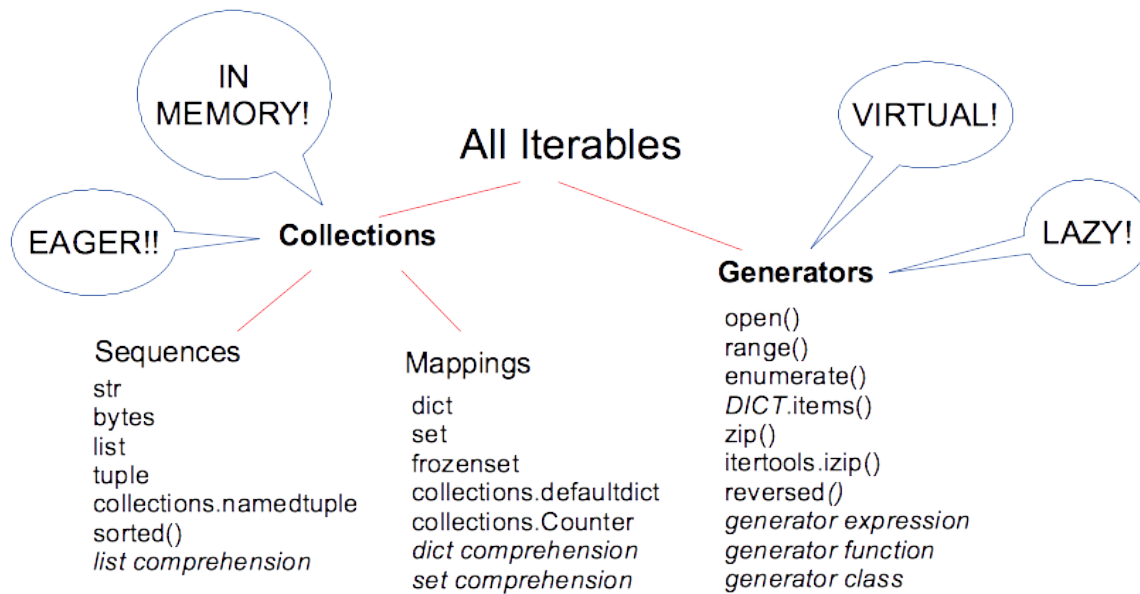
- Expression that can be looped over
- Can be collections *e.g.* list, tuple, str, bytes
- Can be generators *e.g.* range(), file objects, enumerate(), zip(), reversed()

Python has many builtin iterables – a file object, for instance, which allows iterating through the lines in a file.

All builtin collections (list, tuple, str, bytes) are iterables. They keep all their values in memory. Many other builtin iterables are *generators*.

A generator does not keep all its values in memory – it creates them one at a time as needed, and feeds them to the for-in loop. This is a Good Thing, because it saves memory.

Iterables



Generator Expressions

- Like list comprehensions, but create a generator object
- More efficient
- Use parentheses rather than brackets

A generator expression is similar to a list comprehension, but it provides a generator instead of a list. That is, while a list comprehension returns a complete list, the generator expression returns one item at a time.

The main difference in syntax is that the generator expression uses parentheses rather than brackets.

Generator expressions are especially useful with functions like `sum()`, `min()`, and `max()` that reduce an iterable input to a single value:

NOTE | There is an implied **yield** statement at the beginning of the expression.

Example

gen_ex.py

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

# sum the squares of a list of numbers
# using list comprehension, entire list is stored in memory
s1 = sum([x * x for x in range(10)]) ①

# only one square is in memory at a time with generator expression
s2 = sum(x * x for x in range(10)) ②
print(s1, s2)
print()

page = open("../DATA/mary.txt")
m = max(len(line) for line in page) ③
page.close()
print(m)
```

- ① using list comprehension, entire list is stored in memory
- ② with generator expression, only one square is in memory at a time
- ③ only one line in memory at a time. max() iterates over generated values

gen_ex.py

```
285 285

30
```

Generator functions

- Mostly like a normal function
- Use yield rather than return
- Maintains state

A generator is like a normal function, but instead of a return statement, it has a yield statement. Each time the yield statement is reached, it provides the next value in the sequence. When there are no more values, the function calls return, and the loop stops. A generator function maintains state between calls, unlike a normal function.

Example

sieve_generator.py

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

def next_prime(limit):
    flags = set() ①

    for i in range(2, limit):
        if i in flags:
            continue
        for j in range(2 * i, limit + 1, i):
            flags.add(j) ②
        yield i ③

np = next_prime(200) ④
for prime in np: ⑤
    print(prime, end=' ')
```

- ① initialize empty set (to be used for "is-prime" flags)
- ② add non-prime elements to set
- ③ execution stops here until next value is requested by for-in loop
- ④ next_prime() returns a generator object
- ⑤ iterate over **yielded** primes

sieve_generator.py

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107
109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
```

Example

line_trimmer.py

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

def trimmed(file_name):
    with open(file_name) as file_in:
        for line in file_in:
            if line.endswith('\n'):
                line = line.rstrip('\n\r')
            yield line ①

for trimmed_line in trimmed('../DATA/mary.txt'): ②
    print(trimmed_line)
```

- ① 'yield' causes this function to return a generator object
- ② looping over the a generator object returned by trimmed()

line_trimmer.py

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

String formatting

- Numbered placeholders
- Add width, padding
- Access elements of sequences and dictionaries
- Access object attributes

The traditional (i.e., old) way to format strings in Python was with the % operator and a format string containing fields designated with percent signs. The new, improved method of string formatting uses the format() method. It takes a format string and one or more arguments. The format strings contains placeholders which consist of curly braces, which may contain formatting details. This new method has much more flexibility.

By default, the placeholders are numbered from left to right, starting at 0. This corresponds to the order of arguments to format().

Formatting information can be added, preceded by a colon.

```
{:d}          format the argument as an integer +  
{:03d}        format as an integer, 3 columns wide, zero padded +  
{:>25s}      same, but right-justified +  
{:.3f}        format as a float, with 3 decimal places
```

Placeholders can be manually numbered. This is handy when you want to use a format() parameter more than once.

```
"Try one of these: {0}.jpg {0}.png {0}.bmp {0}.pdf".format('penguin')
```

Example

stringformat_ex.py

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

from datetime import date

color = 'blue'
animal = 'iguana'

print('{} {}'.format(color, animal)) ①

fahr = 98.6839832
print('{:.1f}'.format(fahr)) ②

value = 12345
print('{0:d} {0:04x} {0:08o} {0:016b}'.format(value)) ③

data = {'A': 38, 'B': 127, 'C': 9}

for letter, number in sorted(data.items()):
    print("{} {:4d}".format(letter, number)) ④
```

- ① {} placeholders are autonumbered, starting at 0; this corresponds to the parameters to format()
- ② Formatting directives start with ':'; .1f means format floating point with one decimal place
- ③ {} placeholders can be manually numbered to reuse parameters
- ④ :4d means format decimal integer in a field 4 characters wide

stringformat_ex.py

```
blue iguana
98.7
12345 3039 00030071 0011000000111001
A   38
B  127
C    9
```

f-strings

- Shorter syntax for string formatting
- Only available Python 3.6 and later
- Put **f** in front of string

A great new feature, f-strings, was added to Python 3.6. These are strings that contain placeholders, as used with normal string formatting, but the expression to be formatted is also placed in the placeholder. This makes formatting strings more readable, with less typing. As with formatted strings, any expression can be formatted.

Other than putting the value to be formatted directly in the placeholder, the formatting directives are the same as normal Python 3 string formatting.

In normal 3.x formatting:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Bill Gates'
print("{} founded {}".format(name, company))
print("{:10s} {:.2f}".format(x, y))
```

f-strings let you do this:

```
x = 24
y = 32.2345
name = 'Bill Gates'
company = 'Bill Gates'
print(f"{name} founded {company}")
print(f"{x:10s} {y:.2f}")
```


Example

f_strings.py

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

import sys

if sys.version_info.major == 3 and sys.version_info.minor >= 6:

    name = "Tim"
    count = 5
    avg = 3.456
    info = 2093
    result = 38293892

    print(f"Name is [{name:<10s}]") ①
    print(f"Name is [{name:>10s}]") ②
    print(f"count is {count:03d} avg is {avg:.2f}") ③

    print(f"info is {info} {info:d} {info:o} {info:x}") ④

    print(f"${result:,d}") ⑤

    city = 'Orlando'
    temp = 85

    print(f"It is {temp} in {city}") ⑥

else:
    print("Sorry -- f-strings are only supported by Python 3.6+")
```

- ① < means left justify (default for non-numbers), 10 is field width, s formats a string
- ② > means right justify
- ③ .2f means round a float to 2 decimal points
- ④ d is decimal, o is octal, x is hex
- ⑤ , means add commas to numeric value
- ⑥ parameters can be selected by name instead of position

f_strings.py

```
Name is [Tim      ]  
Name is [      Tim]  
count is 005 avg is 3.46  
info is 2093 2093 4055 82d  
$38,293,892  
It is 85 in Orlando
```

Chapter 10 Exercises

Exercise 10-1 (pres_upper.py)

Read the file **presidents.txt**, creating a list of the presidents' last names. Then, use a list comprehension to make a copy of the list of names in upper case. Finally, loop through the list returned by the list comprehension and print out the names one per line.

Exercise 10-2 (pres_by_death.py)

Print out all the presidents first and last names, date of birth, and their political affiliations, sorted by date of birth.

Read the **presidents.txt** file, putting the four fields into a list of tuples.

Loop through the list, sorting by date of birth, and printing the information for each president. Use **sorted()** and a lambda function.

Exercise 10-3 (pres_gen.py)

Write a generator function to provide a sequence of the names of presidents (in "FIRSTNAME MIDDLENAME LASTNAME" format) from the presidents.txt file. They should be provided in the same order they are in the file. You should not read the entire file into memory, but one-at-a-time from the file.

Then iterate over the the generator returned by your function and print the names.

Chapter 11: An Introduction to Python Classes

Objectives

- Understanding the big picture of O-O programming
- Defining a class and its constructor
- Creating object methods
- Adding attributes and properties to a class
- Using inheritance for code reuse
- Adding class data and methods

A class is the representation of an idea, a concept, in the code. An object of a class represents a particular example of the idea in the code. Without classes, a reader of the code would have to guess about the relationships among data items and functions - classes make such relationships explicit and "understood" by compilers. With classes, more of the high-level structure of your program is reflected in the code, not just in the comments.

— Bjarne Stroustrup (creator of C++)

About O-O programming

- Classes are modules that can create objects
- Objects contain data and methods

Python is an object-oriented language. It supports the creation of classes, which are modules that can create objects.

Objects contain both data and the methods that operate on the data. Each object created has its own personal data, called instance data. It can also have data that is shared with all the objects created from its class, called class or static data.

Each class defines an initializer, which initializes and returns an object.

Objects may inherit attributes (data and methods) from other objects.

Methods are not polymorphic; i.e., you can't define multiple versions of a method, with different signatures, and have the corresponding method selected at runtime. However, because Python has dynamic typing, this is seldom needed.

Defining classes

- Use the **class** statement **Syntax**

```
class ClassName(baseclass):  
    pass
```

To create a class, declare the class with the **class** statement. Any base classes may be specified in parentheses, but are not required.

Classes are conventionally named with UpperCamelCase (i.e., all words, including the first, are capitalized). This is also known as CapWords, StudlyCaps, etc. Modules conventionally have lower-case names. Thus, it is usual to have module `rocketengine` containing class `RocketEngine`.

All methods, including the constructor, are passed the object itself. This is conventionally named "self", and while this is not mandatory, most Python programmers expect it.

The basic layout is this:

```
class ClassName(baseclass):  
    classvar = value  
  
    def __init__(self,...):  
        self._attrib = instancevalue;  
        ClassName.attrib = classvalue;  
  
    def method1(self,...):  
        self._attrib = instancevalue  
  
    def method2(self,...):  
        x = self.method1()
```

Example

simple_class.py

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

class Simple(): ①
    def __init__(self, message_text): ②
        self._message_text = message_text ③

    def text(self): ④
        return self._message_text

if __name__ == "__main__":
    msg1 = Simple('hello') ⑤
    print(msg1.text()) ⑥

    msg2 = Simple('hi there') ⑦
    print(msg2.text())
```

- ① default base class is **object**
- ② constructor
- ③ message text stored in instance object
- ④ instance method
- ⑤ instantiate an instance of Simple
- ⑥ call instance method
- ⑦ create 2nd instance of Simple

simple_class.py

```
hello
hi there
```


Constructors

- Constructor is named ***`_init_`***
- AKA initializer
- Passed *self* plus any parameters

A class's constructor (also known as the initializer) is named *init*. It receives the object being created, and any parameters passed into the initializer in the code as part of instantiation.

As with any Python function, the constructor's parameters can be fixed, optional, keyword-only, or keyword.

It is also normal to name data elements (variables) of a class with a leading underscore to indicate (in a non-mandatory way) that the variable is private. Access to private variables should be provided via public access methods (AKA getters) or properties.

Instance methods

- Expect the object as first parameter
- Object conventionally named *self*
- Otherwise like normal Python functions
- Use *self* to access instance attributes or methods
- Use class name to access class data

Instance methods are defined like normal functions, but like constructors, the object that the method is called from is passed in as a parameter. Like the constructor, it should be named *self*.

Example

animal.py

```
#!/usr/bin/env python
class Animal():
    count = 0 ①

    def __init__(self, species, name, sound):
        self._species = species
        self._name = name
        self._sound = sound
        Animal.count += 1

    @property
    def species(self):
        return self._species

    @classmethod
    def kill(cls):
        cls.count -= 1

    @property
    def name(self):
        return self._name

    def make_sound(self):
        print(self._sound)

    @classmethod
    def remove(cls):
        cls.count -= 1 ②

    @classmethod
    def zoo_size(cls): ③
        return cls.count

if __name__ == "__main__":
    leo = Animal("African lion", "Leo", "Roarrrrrrr")
    garfield = Animal("cat", "Garfield", "Meowwww")
    felix = Animal("cat", "Felix", "Meowwww")
```

```
print(leo.name, "is a", leo.species, "--", end=' ')
leo.make_sound()

print(garfield.name, "is a", garfield.species, "--", end=' ')
garfield.make_sound()

print(felix.name, "is a", felix.species, "--", end=' ')
felix.make_sound()
```

- ① class data
- ② update class data from instance
- ③ zoo_size gets class object when called from instance or class

animal.py

```
Leo is a African lion -- Roarrrrrrrr
Garfield is a cat -- Meowwwwww
Felix is a cat -- Meowwwwww
```

Properties

- Properties are managed attributes
- Create with `@property` decorator
- Create getter, setter, deleter, docstring
- Specify getter only for read-only property

An object can have properties, or managed attributes. When a property is evaluated, its corresponding getter method is invoked; when a property is assigned to, its corresponding setter method is invoked.

Properties can be created with the `@property` decorator and its derivatives. `@property` applied to a method causes it to be a "getter" method for a property with the same name as the method.

Using `@name.setter` on a method with the same name as the property creates a setter method, and `@name.deleter` on a method with the same name creates a deleter method.

Example

properties.py

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

class Person():

    def __init__(self, firstname=None, lastname=None):
        self._first_name = None
        self._last_name = None
        self.first_name = firstname ①
        self.last_name = lastname

    @property
    def first_name(self): ②
        return self._first_name

    @first_name.setter ③
    def first_name(self, value): ④
        if value is None or value.isalpha():
            self._first_name = value
        else:
            raise ValueError("First name may only contain letters")

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if value is None or value.isalpha():
            self._last_name = value
        else:
            raise ValueError("Last name may only contain letters")

if __name__ == '__main__':
    person1 = Person('Ferneater', 'Eulalia')

    person2 = Person()
    person2.last_name = 'Pepperpot' ⑤
    person2.first_name = 'Hortense'
```

```
print("{} {}".format(person1.first_name, person1.last_name))
print("{} {}".format(person2.first_name, person2.last_name))

try:
    person3 = Person("R2D2")
except ValueError as err:
    print(err)
else:
    print("{} {}".format(person3.first_name, person3.last_name))
```

- ① calls property
- ② getter property
- ③ decorator comes from getter property
- ④ setter property
- ⑤ access property

properties.py

```
Ferneater Eulalia  
Hortense Pepperpot  
First name may only contain letters
```


Class methods and data

- Defined in the class, but outside of methods
- Defined as attribute of class name (similar to self)
- Define class methods with `@classmethod`
- Class methods get the class object as 1st parameter

Most classes need to store some data that is common to all objects created in the class. This is generally called class data.

Class attributes can be created by using the class name directly, or via class methods.

A class method is created by using the `@classmethod` decorator. Class methods are implicitly passed the class object.

Class methods can be called from the class object or from an instance of the class; in either case the method is passed the class object.

Example

class_methods_and_data.py

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog" ①

    def __init__(self, weapon):
        self.weapon = weapon

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon)) ②

    @classmethod ③
    def get_location(cls): ④
        return cls.LOCATION ⑤

r = Rabbit("a nice cup of tea")
print(Rabbit.get_location()) ⑥
print(r.get_location()) ⑦
```

- ① class data (not duplicated in instances)
- ② instance method
- ③ the **@classmethod** decorator makes a function receive the class object, not the instance object
- ④ `*get_location()` is a *class* method
- ⑤ class methods can access class data via **cls**
- ⑥ call class method from class
- ⑦ call class method from instance

class_methods_and_data.py

```
the Cave of Caerbannog
the Cave of Caerbannog
```

Static Methods

- Define with `@staticmethod`

A static method is a utility method that is included in the API of a class, but does not require either an instance or a class object. Static methods are not passed any implicit parameters.

Many classes do not need any static methods.

Define static methods with the `@staticmethod` decorator.

Example

```
class Spam():  
  
    @staticmethod  
    def format_as_title(s): # no implicit parameters  
        return s.strip().title()
```

Private methods

- Called by other methods in the class
- Not visible to users of the class
- Conventionally named with leading underscore

Private methods are those that are called only within the class. They are not part of the API – they are not visible to users of the class. Private methods may be instance, class, or static methods; if calling a private instance method, pass the instance variable (`self`) explicitly.

It is conventional to name a private method with a leading underscore. This does not protect it from use, but gives programmers a hint that it's for internal use only.

Inheritance

- Specify base classes after class name
- Multiple inheritance OK
- Depth-first, left-to-right search for methods not in derived class

Classes may inherit methods and data from one or more other classes by specifying a parenthesized list of base classes after the class name in the class definition.

If a method or attribute is not found in the derived class, it is first sought in the first base class in the list. If not found, it is sought in the base class of that class, if any, and so on. This is usually called a depth-first search.

The derived class inherits all attributes of the base class. If the base class initializer takes the same arguments as the derived class, then no extra coding is needed. Otherwise, to explicitly call the initializer in the base class, use `super(ThisClass,self).init(args)`.

The simplest derived class would be:

```
class Mammal(Animal):  
    pass
```

A Mammal object will have all the attributes and methods of an Animal object.

Example

mammal.py

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

from animal import Animal

class Mammal(Animal): ①
    def __init__(self, species, name, sound, gestation):
        super(Mammal, self).__init__(species, name, sound)
        self._gestation = gestation

    @property
    def gestation(self): ②
        """Length of gestation period in days"""
        return self._gestation

if __name__ == "__main__":
    mammal1 = Mammal("African lion", "Bob", "Roarrrr", 120)
    print(mammal1.name, "is a", mammal1.species, "--", end=' ')
    mammal1.make_sound()

    print("Number of animals", mammal1.zoo_size())

    mammal2 = Mammal("Fruit bat", "Freddie", "Squeak!!", 180)
    print(mammal2.name, "is a", mammal2.species, "--", end=' ')
    mammal2.make_sound()

    print("Number of animals", mammal2.zoo_size())
    print("Number of animals", Mammal.zoo_size())

    mammal1.kill()
    print("Number of animals", Mammal.zoo_size())

    print("Gestation period of the", mammal1.species, "is", mammal1.gestation,
          "days")
    print("Gestation period of the", mammal2.species, "is", mammal2.gestation,
          "days")
```

- ① inherit from Animal
- ② add property to existing Animal properties

mammal.py

```
Bob is a African lion -- Roarrrr
Number of animals 1
Freddie is a Fruit bat -- Squeak!!
Number of animals 2
Number of animals 2
Number of animals 1
Gestation period of the African lion is 120 days
Gestation period of the Fruit bat is 180 days
```

Untangling the nomenclature

There are many terms to distinguish the various parts of a Python program. This chart is an attempt to help you sort out what is what:

Table 15. Objected-oriented Nomenclature

attribute	A variable or method that is part of a class or object
base class	A class from which other classes inherit
child class	Same as derived class
class	A Python module from which objects may be created
class method	A function that expects the class object as its first parameter. Such a function can be called from either the class itself or an instance of the class. Created with <code>@classmethod</code> decorator.
derived class	A class which inherits from some other class
function	An executable subprogram.
instance method	A function that expects the instance object, conventionally named <code>self</code> , as its first parameter. See "method".
method	A function defined inside a class.
module	A file containing python code, and which is designed to be imported into Python scripts or other modules.
package	A folder containing one or more modules. Packages may be imported. There must be a file named <code>init.py</code> in the package folder.
parent class	Same as base class
property	A managed attribute (variable) of an instance of a class
script	A Python program. A script is an executable file containing Python commands.
static method	A function in a class that does not automatically receive any parameters; typically used for private utility functions. Created with <code>@staticmethod</code> decorator.
superclass	Same as base class

Chapter 11 Exercises

Exercise 11-1 (knight.py, knight_info.py)

Part 1:

Create a module which defines a class named Knight.

The initializer for the class should expect the knight's name as a parameter. Get the information from the file knights.txt to initialize the object.

The object should have these (read-only) properties:

name
title
favorite_color
quest
comment

Example

```
from knight import Knight
k = Knight('Arthur')
print k.favorite_color
```

Part 2:

Create an application to use the Knight class created in part one. For each knight specified on the command line, create a knight object and print out the knight's name, favorite color, quest, and comment. Precede the name with the knight's title.

Example output:

```
kinfo.py Arthur Bedevere
Name: King Arthur
Favorite Color: blue
Quest: The Grail
Comment: King of the Britons

Name: Sir Bedevere
Favorite Color: red, no, blue!
Quest: The Grail
Comment: AARRRRRRRGGGGHH
```

Chapter 12: Developer Tools

Objectives

- Run pylint to check source code
- Debug scripts
- Find speed bottlenecks in code
- Compare algorithms to see which is faster

Program development

- More than just coding
 - Design first
 - Consistent style
 - Comments
 - Debugging
 - Testing
 - Documentation

Comments

- Keep comments up-to-date
- Use complete sentences
- Block comments describe a section of code
- Inline comments describe a line
- Don't state the obvious

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1      # Increment x
```

Only use an inline comment if the reason for the statement is not obvious:

```
x = x + 1      # Add one so range() does the right thing
```

The above was adapted from PEP 8

pylint

- Checks many aspects of code
- Finds mistakes
- Rates your code for standards compliance
- Don't worry if your code has a lower rating!
- Can be highly customized

pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler's Refactoring book)

from the pylint documentation

pylint can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions. It was developed by Python coders at Logilab <http://www.logilab.fr>.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions. To use pylint, just say `pylint filename`, or `pylint directory`. Most Python IDEs have pylint, or the equivalent, built in.

Other tools for analyzing Python code: * `pyflakes` * `pychecker`

Customizing pylint

- Use `pylint --generate-rcfile`
- Redirect to file
- Edit as needed
- Knowledge of regular expressions useful
- Name file `~/.pylintrc` on Linux/Unix/OS X
- Use `-rcfile` file to specify custom file on Windows

To customize pylint, run pylint with only the `-generate-rcfile` option. This will output a well-commented configuration file to STDOUT, so redirect it to a file.

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

Windows

Put the file in a convenient location (name it something like `pylintrc`). Invoke pylint with the `-rcfile` option to specify the location of the file.

pylint will also find a file named `pylintrc` in the current directory, without needing the `-rcfile` option.

Non-Windows systems

On Unix-like systems (Unix, Mac OS, Linux, etc.), `/etc/pylintrc` and `~/.pylintrc` will be automatically loaded, in that order.

See docs.pylint.org for more details.

Using pyreverse

- Source analyzer
- Reverse engineers Python code
- Part of pylint
- Generates UML diagrams

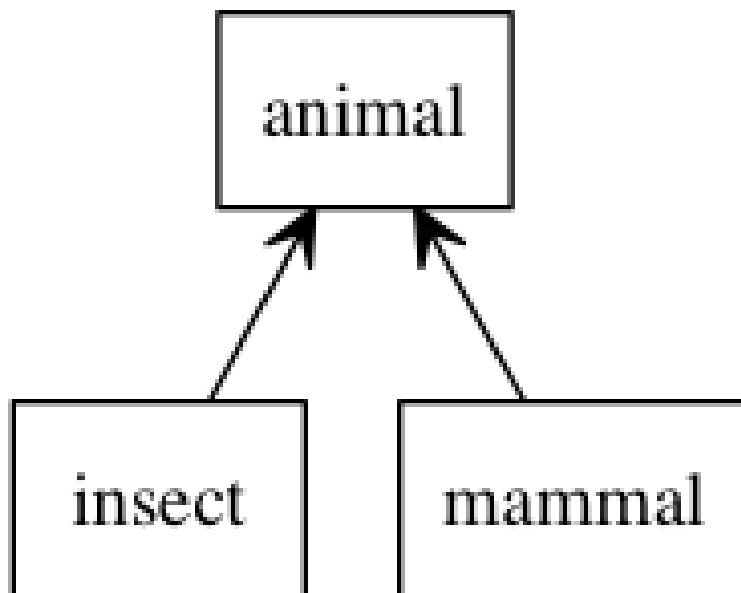
pyreverse is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the pylint package.

There are many options to control what it analyzes and what kind of output it produces.

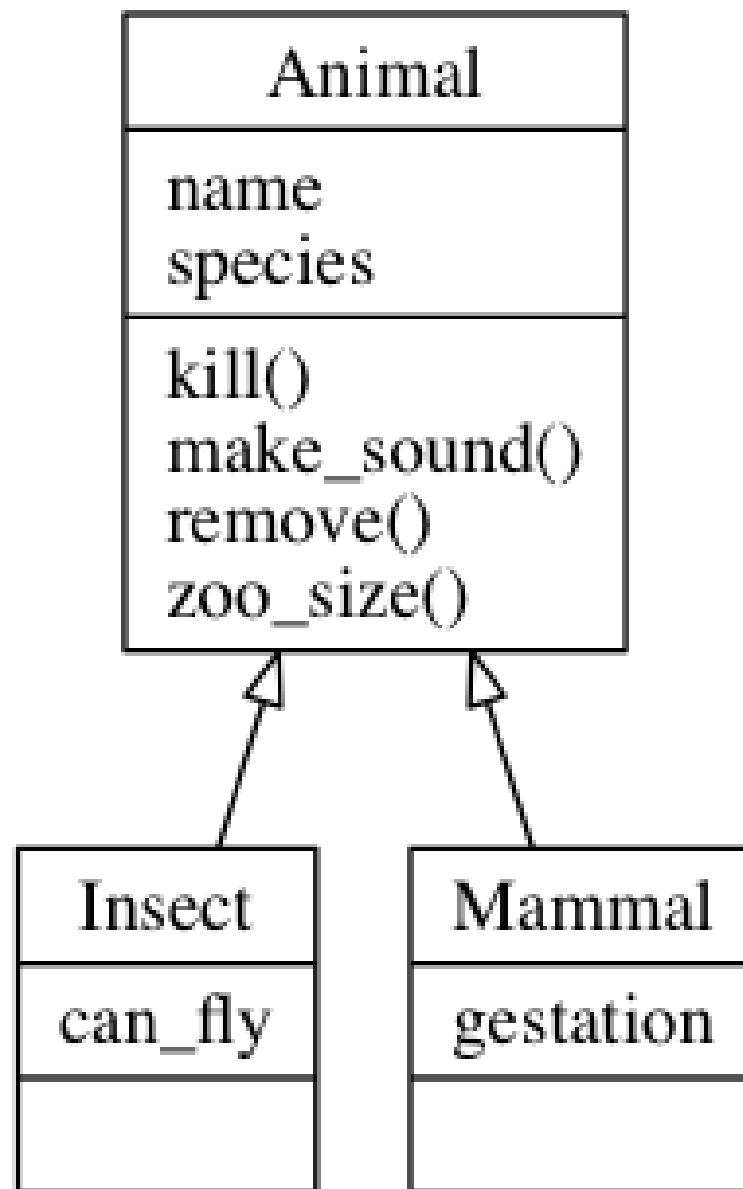
Example

```
pyreverse -o png -p MyProject -A animal.py mammal.py insect.py
```

packages_MyProject.png



classes_MyProject.png



NOTE | [pyreverse](#) requires the (non-Python) Graphviz utility to be installed.

The Python debugger

- Implemented via pdb module
- Supports breakpoints and single stepping
- Based on gdb

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The pdb module provides debugging facilities for Python.

The usual way to use pdb is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

Starting debug mode

- Syntax

```
python -m pdb script
```

or

```
import pdb
pdb.run('function')
```

`pdb` is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Typical usage to run a program under control of the debugger is:

```
>>> import pdb
>>> import some_module
>>> pdb.run('some_module.function_to_text()')
> <string>(0)?()
(Pdb) c      # (c)ontinue
> <string>(1)?()
(Pdb) c      # (c)ontinue
NameError: 'spam'
> <string>(1)?()
(Pdb)
```

To get help, type **h** at the debugger prompt.

Stepping through a program

- **s** single-step, stepping into functions
- **n** single-step, stepping over functions
- **r** return from function
- **c** run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing Enter repeats most commands; if the previous command was list, the debugger lists the next set of lines.

Setting breakpoints

- Syntax

```
b list all breakpoints
b linenumber (, condition)
b file:linenumber (, condition)
b function name (, condition)
```

Breakpoints can be set with the **b** command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The **tbreak** command creates a one-time breakpoint that is deleted after it is hit the first time.

Profiling

- Use the **profile** module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT. You can also specify an output file with the -o option, and the sort order with the -s option. See the docs for more information.

TIP

The **pycallgraph** module (not in the standard library) will create a graphical representation of an application's profile, indicating visually where the application is spending the most time.

Example

```
python -m profile count_with_dict.py
...script output...
    19 function calls in 0.000 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    14    0.000    0.000    0.000    0.000 :0(get)
     1    0.000    0.000    0.000    0.000 :0(items)
     1    0.000    0.000    0.000    0.000 :0(open)
     1    0.000    0.000    0.000    0.000 :0(setprofile)
     1    0.000    0.000    0.000    0.000 count_with_dict.py:3(<module>)
     1    0.000    0.000    0.000    0.000 profile:0(<code object <module> at
0xb74c36e0, file "count_with_dict.py", line 3>)
     0    0.000          0.000          profile:0(profiler)
```

Benchmarking

- Use the `timeit` module
- Create a timer object with specified # of repetitions

Use the `timeit` module to benchmark two or more code snippets. To time code, create a `Timer` object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the `timeit()` method with the number of times to call the test code, or call the `repeat()` method which repeats `timeit()` a specified number of times.

You can also use the **`timeit`** module from the command line. Use the **`-s`** option to specify startup code:

```
python -m timeit -s "startup code..." "code..."
```


Example

bm_range_vs_while.py

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

import timeit

setup_code = 'values = []' ①

test_code_one = '''
for i in range(10000):
    values.append(i)
''' ②
test_code_two = '''
i = 0
while i < 10000:
    values.append(i)
    i += 1
''' ②

t1 = timeit.Timer(test_code_one, setup_code) ③
t2 = timeit.Timer(test_code_two, setup_code) ③

print("test one:")
print(t1.timeit(1000)) ④
print()

print("test two:")
print(t2.timeit(1000)) ④
print()
```

- ① setup code is only executed once
- ② code fragment executed many times
- ③ Timer object creates time-able code
- ④ timeit() runs code fragment N times

bm_range_vs_while.py

```
test one:  
1.354834071  
  
test two:  
2.2168111809999997
```

Chapter 12 Exercises

Exercise 12-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

Exercise 12-2

Use any available debugger to step through any of the scripts you have written so far.

Chapter 13: Excel Spreadsheets

Objectives

- Open Excel spreadsheets and extract data
- Update spreadsheets
- Create new spreadsheets

The openpyxl module

- Provides read/write access to Excel spreadsheets
- Handles .xlsx and .xls formats
- Does not require Excel

The **openpyxl** module allows you to read, write, and create **Excel** spreadsheets. It handles both the new (Excel 2007 and later) and old formats.

openpyxl does not require Excel to be installed.

You can open existing workbooks or create new ones. You can do most anything that you could do manually in a spreadsheet – update or insert data, create formulae, add or change styles, even hide columns.

Whether you open an existing spreadsheet or create a new one, it is an object of type `Workbook`. A workbook contains one or more `Worksheet` objects.

From a `Worksheet` you can access cells directly, or create a range of cells.

The data in each cell can be manipulated through its `value` property. Other properties, such as `font` and `number_format`, control the display of the data.

View the full documentation at <http://openpyxl.readthedocs.org/en/latest/index.html>.

TIP | To save typing, import openpyxl as `px`.

Reading an existing spreadsheet

- Use `load_workbook()` to open file
- Get active worksheet with `WB.active`
- List all worksheets with `get_sheet_by_name()`
- Get named worksheet with `WB['worksheet-name']`

To open and read an existing spreadsheet, use the `load_workbook()` function. This returns a `WorkBook` object.

There are several ways to get a worksheet from a workbook. To list all the sheets by name, use `WB.get_sheet_names()`, and then `WB.get_sheet_by_name()` to get a particular sheet.

A workbook is also an iterable of all the worksheets it contains, and is a dictionary of worksheets by name.

The active property of a workbook is the currently active worksheet. `WB.active` may be used as soon as a workbook is open.

Example

px_load_worksheet.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    wb = px.load_workbook('../DATA/presidents.xlsx')

    # three ways to get to a worksheet:

    # 1
    print(wb.sheetnames, '\n')
    ws = wb['US Presidents']
    print(ws, '\n')

    # 2
    for ws in wb:
        print(ws)
    print()

    # 3
    ws = wb.active
    print(ws, '\n')

    print(ws['A1'].value)
    print(ws['C2'].value, ws['B2'].value)

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


px_load_worksheet.py

```
['US Presidents']  
  
<Worksheet "US Presidents">  
  
<Worksheet "US Presidents">  
  
<Worksheet "US Presidents">  
  
Term  
George Washington
```

Working with ranges

- Range represents a rectangle of cells
- Use slice notation on the worksheet
- Iterate through rows, then columns

To get a range of cells, use slice notation on the worksheet object and standard cell notation, e.g. `WS['A1':'M9']`

The range is a virtual list of rows, and so can be iterated over. Each element of a row is a cell object. Use the `.value` attribute to get or set the cell value.

Example

`px_get_ranges.py`

```
#!/usr/bin/env python
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    print_first_and_last_names(ws)

def print_first_and_last_names(ws):
    """Print first and last names of all presidents"""
    pres_range = ws['B2':'C46'] # cell range
    for row in pres_range: # row object
        print(row[1].value, row[0].value)

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

px_get_ranges.py

```
George Washington  
John Adams  
Thomas Jefferson  
James Madison  
James Monroe  
John Quincy Adams  
Andrew Jackson  
Martin Van Buren  
William Henry Harrison  
John Tyler
```

...

Worksheet info

- Use properties of worksheet

Once a worksheet is opened, you can get information about the worksheet directly from the worksheet object.

Example

`px_worksheet_info.py`

```
#!/usr/bin/env python
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    print(ws.dimensions)
    print(ws.min_column)
    print(ws.min_row)
    print(ws.max_column)
    print(ws.max_row)

    # returns value for specified cell
    print(ws.cell(row=2, column=3).value, ws.cell(row=2, column=2).value, '\n')

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

`px_worksheet_info.py`

```
A1:J46
1
1
10
46
George Washington
```

Modifying a worksheet

- Assign to cells
- Use `WS.cell(row=ROW, column=COLUMN).value`

To modify a worksheet, you can either iterate through rows and columns as described above, or access individual cells with the `WS.cell()` function. It takes a row and column (integers, and 1-based) and returns the cell object. You can assign to the `value` attribute of the cell.

Use `WB.save('name.xlsx')` to save the changes.

Example

px_modify_sheet.py

```
#!/usr/bin/env python
from datetime import date
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    add_age_at_inauguration(ws)

    wb.save('presidents1.xlsx')

def make_date(date_str):
    """Convert date string returned by CELL.value into Python date object"""
    year, month, day = date_str.split('-')
    return date(int(year), int(month), int(day))

def add_age_at_inauguration(ws):
    """Add a new column with age of inauguration"""
    new_col = ws.max_column + 1
    print(new_col)
    ws.cell(row=1, column=new_col).value = 'Age at Inauguration'
    for row in range(2, 47):
        birth_date = make_date(ws.cell(row=row, column=4).value) # treat date as
string
        inaugural_date = make_date(ws.cell(row=row, column=8).value)
        raw_age_took_office = inaugural_date - birth_date
        age_took_office = raw_age_took_office.days / 365.25
        ws.cell(row=row, column=new_col).value = age_took_office

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

Working with formulas

- Assign to cell value as a string
- Be sure to start with '='

To add or update a formula, assign the formula as a string to the cell value.

Example

px_formulas.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx', guess_types=True)
    ws = wb['US Presidents']

    add_age_at_inauguration(ws)

    wb.save('presidents2.xlsx')

def add_age_at_inauguration(ws):
    """Add a new column with age of inauguration"""
    new_col = ws.max_column + 1
    print(new_col)
    ws.cell(row=1, column=new_col).value = 'Age at Inauguration'
    for row in range(2, 47):
        new_cell = ws.cell(row=row, column=new_col)
        new_cell.value = '=(H{0}-D{0})/365.25'.format(row)
        new_cell.number_format = '0.0'

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


Creating a new spreadsheet

- Use the Workbook() function
- One worksheet created by default
- Add worksheets with WB.create_sheet(n)

To create a new spreadsheet file, use the Workbook() function. It creates a new workbook, with a default worksheet named "Sheet1". You can add more worksheets with WB.create_sheet(n). The parameter indicates where to insert the new worksheet; if not specified, it is added at the end. To get or set the name of the worksheet, use its title property.

Example

px_create_worksheet.py

```
#!/usr/bin/env python
import openpyxl as px

fruits = ["pomegranate", "cherry", "apricot", "date", "apple", "lemon", "kiwi",
          "orange", "lime", "watermelon", "guava", "papaya", "fig", "pear",
          "banana",
          "tamarind", "persimmon", "elderberry", "peach", "blueberry", "lychee",
          "grape"]

wb = px.Workbook()

ws = wb.active

ws.title = 'fruits'

for i, fruit in enumerate(fruits, 1):
    ws.cell(row=i, column=1).value = fruit
    ws.cell(row=i, column=2).value = len(fruit)

wb.save('fruits.xlsx')
```

Setting Styles

- Must be set on each cell individually
- Cannot change, once assigned (but can be replaced)
- Copy style to make changes

Each cell has a group of attributes that control its styles and formatting. Most of these have a corresponding class; to change styles, create an instance of the appropriate class and assign it to the attribute.

You can also make a copy of an existing style object, and just change the attributes you need.

Example

px_styles.py

```
#!/usr/bin/env python
import openpyxl as px

def main():
    """program entry point"""
    wb = px.load_workbook('../DATA/presidents.xlsx')
    ws = wb['US Presidents']

    update_last_names(ws)

    wb.save('presidents3.xlsx')

def update_last_names(ws):
    """Make the last name column blue and bold"""
    for row in ws['B2:B46']:
        cell = row[0]
        cell.value = cell.value.upper()
        cell.font = px.styles.Font(color='FF0000FF')

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

Table 16. *openpyxl* Cell Style Attributes

Cell attribute	Class	Parameters	Default values
font	Font		
		name	'Calibri'
		size	11
		bold	False
		italic	False
		vertAlign	None
		underline	'none'
		strike	False
		color	'FF000000'
fill	PatternFill		
		fill_type	None
		start_color	'FFFFFFFF'
		end_color	'FF000000'
border	Border		
		left	Side(border_style=None, color='FF000000')
		right	Side(border_style=None, color='FF000000')
		top	Side(border_style=None, color='FF000000')
		bottom	Side(border_style=None, color='FF000000')
		diagonal	Side(border_style=None, color='FF000000')
		diagonal_direction	
		outline	Side(border_style=None, color='FF000000')

Cell attribute	Class	Parameters	Default values
		vertical	Side(border_style=None, color='FF000000')
		horizontal	Side(border_style=None, color='FF000000')
alignment	Alignment		
		horizontal	'general'
		vertical	'bottom'
		text_rotation	0
		wrap_text	False
		shrink_to_fit	False
		indent	0
number_format	None	N/A	'General'
protection	Protection		
		locked	True
		hidden	False

Chapter 13 Exercises

Exercise 13-1 (`age_of_geeks.py`)

Write a script to compute the average age of the people on the worksheet 'people' in `computer_people.xlsx`. First, you'll have to calculate the age from the birthdate. (Some of the people in the list have died. Ignore this for purposes of this exercise). :q

Exercise 13-2 (`knight_to_spreadsheet.py`, `knight_to_spreadsheet_extra.py`)

Write a script to create a new spreadsheet with data from the `knight.txt` file. The first row of the spreadsheet should have the column headings:

Name, Title, Favorite Color, Quest, Comment

The data should start after that.

Save the workbook as `knight.xlsx`.

NOTE

For extra fun, make the headers bold, the name fields red, and the comments in italics.

Chapter 14: Serializing Data

Objectives

- Have a good understanding of the XML format
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write CSV data
- Read and write YAML data

About XML

- Variant of SGML
- All data contained within tags

An XML document consists of a single element, which contains sub-elements, which can have further sub-elements inside them. Elements are indicated by tags in the text. Tags are always inside angle brackets < >. Elements can either contain content, or they can be empty.

Tags can contain attributes, indicated by attribute="value". Tags can either appear in begin/end pairs, in which the end tag starts with a slash, or as a single tag, in which case the tag ends with a slash. Attributes must be surrounded by double quotes. All tag names and attribute names should be lower case.

Normal Approaches to XML

- SAX
 - One scan through file
 - Good for large files
 - Uses callbacks on XML parsing events
- DOM
 - Builds a document tree
 - Python supports both through many libraries

There are two approaches normally used in working with XML.

The first is called SAX, which stands for Simple API for XML . It processes an XML file as a single stream, and so is appropriate for large XML files.

SAX processing consists of attaching callback functions to SAX events. These events are created when the XML reader encounters all the various components of an XML file – begin tags, end tags, data, and so forth.

The second approach is called the DOM, (Document Object Model), which parses an XML document into a tree that's fully resident in memory.

A top-level Document instance is the root of the tree, and has a single child which is the top-level Element instance; this Element has child nodes representing the content and any sub-elements, which may in turn have further children and so forth. Classes such as Text, Comment, CDATASection, EntityReference, provide access to XML structure and data. Nodes have methods for accessing the parent and child nodes, accessing element and attribute values, insert and delete nodes, and converting the tree back into XML.

The DOM is often useful for modifying XML documents, because you can create a DOM, modify it by adding new nodes and moving sub-trees around, and then produce a new XML document as output.

While the DOM specification doesn't require that the entire tree be resident in memory at one time, many of the Python DOM implementations keep the whole tree in RAM, which can limit the size of the file being processed.

Which module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use `xml.etree.ElementTree`

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, **`xml.etree.ElementTree`** is fast, provides both SAX and DOM style parsing, and unlike many of the other modules, has a Pythonic interface.

If available, use **`lxml.etree`**, which is a superset of `ElementTree` with some nice extra features, such as pretty-printing.

Table 17. XML Modules in the Python 3 Standard Library

Module	Description
<code>xml.parsers.expat</code>	Fast XML parsing using Expat
<code>xml.dom</code>	The Document Object Model API
<code>xml.dom.minidom</code>	Lightweight DOM implementation
<code>xml.dom.pulldom</code>	Support for building partial DOM trees
<code>xml.sax</code>	Support for SAX2 parsers
<code>xml.sax.handler</code>	Base classes for SAX handlers
<code>xml.sax.saxutils</code>	SAX Utilities
<code>xml.sax.xmlreader</code>	Interface for XML parsers
<code>xml.etree.ElementTree</code>	Pythonic interface to XML

Getting Started With ElementTree

- Import `xml.etree.ElementTree` (or `lxml.etree`) as `ET` for convenience
- Parse XML or create empty `ElementTree`

`ElementTree` is part of the Python standard library; `lxml` is included with the Anaconda distribution.

Since putting "`xml.etree.ElementTree`" in front of its methods requires a lot of extra typing, it is typical to alias `xml.etree.ElementTree` to just `ET` when importing it: `import xml.etree.ElementTree as ET`

You can check the version of `ElementTree` via the `VERSION` attribute:

```
import xml.etree.ElementTree as ET
print(ET.VERSION)
```

How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use ElementTree.parse(); this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

Elements

- Element has
 - Tag name
 - Attributes (implemented as a dictionary)
 - Text
 - Tail
 - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the `get()` method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.

TIP | Only the tag property of an Element is required; other properties are optional.

Table 18. Element properties and methods

Property	Description
append(element)	Add a subelement element to end of subelements
attrib	Dictionary of element's attributes
clear()	Remove all subelements
find(path)	Find first subelement matching path
findall(path)	Find all subelements matching path
findtext(path)	Shortcut for find(path).text
get(attr)	Get an attribute; Shortcut for attrib.get()
getiterator()	Returns an iterator over all descendants
getiterator(path)	Returns an iterator over all descendants matching path
insert(pos,element)	Insert subelement element at position pos
items()	Get all attribute values; Shortcut for attrib.items()
keys()	Get all attribute names; Shortcut for attrib.keys()
remove(element)	Remove subelement element
set(attrib,value)	Set an attribute value; shortcut for attr[attrib] = value
tag	The element's tag
tail	Text following the element
text	Text contained within the element

Table 19. *ElementTree* properties and methods

Property	Description
<code>find(path)</code>	Finds the first toplevel element with given tag; shortcut for <code>getroot().find(path)</code> .
<code>findall(path)</code>	Finds all toplevel elements with the given tag; shortcut for <code>getroot().findall(path)</code> .
<code>findtext(path)</code>	Finds element text for first toplevel element with given tag; shortcut for <code>getroot().findtext(path)</code> .
<code>getiterator(path)</code>	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
<code>getroot()</code>	Return the root node of the document
<code>parse(filename)</code> <code>parse(fileobj)</code>	Parse an XML source (filename or file-like object)
<code>write(filename,encoding)</code>	Writes XML document to filename, using encoding (Default <code>us-ascii</code>).

Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call write.

To output an XML string from your elements, call ET.tostring(), passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use .decode() to convert it to a normal Python string.

For an example of creating an XML document from a data file, see **xml_create_knights.py** in the EXAMPLES folder

Example

xml_create_movies.py

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

# from xml.etree import ElementTree as ET
import lxml.etree as ET

movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```

xml_create_movies.py

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```

Parsing An XML Document

- Use `ElementTree.parse()`
- returns an `ElementTree` object
- Use `get*` or `find*` methods to select an element

Use the `parse()` method to parse an existing XML document. It returns an `ElementTree` object, from which you can find the root, or any other element within the document.

To get the root element, use the `getroot()` method.

Example

```
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()
```

Navigating the XML Document

- Use `find()` or `findall()`
- Element is iterable of its children
- `findtext()` retrieves text from element

To find the first child element with a given tag, use `find('tag')`. This will return the first matching element. The `findtext('tag')` method is the same, but returns the text within the tag.

To get all child elements with a given tag, use the `findall('tag')` method, which returns a list of elements.

to see whether a node was found, say

```
if node is None:
```

but to check for existence of child elements, say

```
if len(node) > 0:
```

A node with no children tests as false because it is an empty list, but it is not None.

TIP

The `ElementTree` object also supports the `find()` and `findall()` methods of the `Element` object, searching from the root object.

Example

xml_planets_nav.py

```
#!/usr/bin/env python
'''Use etree navigation to extract planets from solar.xml'''
import lxml.etree as ET

def main():
    '''Program entry point'''
    doc = ET.parse('../DATA/solar.xml') ①

    solar_system = doc.getroot() ②

    print(solar_system)
    print()

    inner = solar_system.find('innerplanets') ③
    print('Inner:')

    for planet in inner: ④
        if planet.tag == 'planet':
            print('\t', planet.get("planetname", "NO NAME"))

    outer = solar_system.find('outerplanets')
    print('Outer:')

    for planet in outer:
        print('\t', planet.get("planetname"))

    plutoids = solar_system.find('dwarfplanets')
    print('Dwarf:')

    for planet in plutoids:
        print('\t', planet.get("planetname"))

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

xml_planets_nav.py

```
<Element solarsystem at 0x1079bf5c8>
```

```
Inner:
```

```
    Mercury
```

```
    Venus
```

```
    Earth
```

```
    Mars
```

```
Outer:
```

```
    Jupiter
```

```
    Saturn
```

```
    Uranus
```

```
    Neptune
```

```
Dwarf:
```

```
    Pluto
```

Example

xml_read_movies.py

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

# import xml.etree.ElementTree as ET
import lxml.etree as ET

movies_doc = ET.parse('movies.xml') ①

movies = movies_doc.getroot() ②

for movie in movies: ③
    print('{} by {}'.format(
        movie.get('name'), ④
        movie.findtext('director'), ⑤
    )
)
```

- ① read and parse the XML file
- ② get the root element (<movies>)
- ③ loop through children of root element
- ④ get 'name' attribute of movie element
- ⑤ get 'director' attribute of movie element

xml_read_movies.py

```
Jaws by Spielberg, Stephen
Vertigo by Alfred Hitchcock
Blazing Saddles by Brooks, Mel
Princess Bride by Reiner, Rob
Avatar by Cameron, James
```

Using XPath

- Use simple XPath patterns Works with find* methods

When a simple tag is specified, the find* methods only search for subelements of the current element. For more flexible searching, the find* methods work with simplified **XPath** patterns. To find all tags named 'spam', for instance, use `./spam`.

```
./movie  
presidents/president/name/last
```

Example

xml_planets_xpath1.py

```
#!/usr/bin/env python  
  
# import xml.etree.ElementTree as ET  
import lxml.etree as ET  
  
doc = ET.parse('../DATA/solar.xml') ①  
  
inner_nodes = doc.findall('innerplanets/planet') ②  
  
outer_nodes = doc.findall('outerplanets/planet') ③  
  
print('Inner:')  
for planet in inner_nodes: ④  
    print('\t', planet.get("planetname")) ⑤  
  
print('Outer:')  
for planet in outer_nodes: ④  
    print('\t', planet.get("planetname")) ⑤
```


- ① parse XML file
- ② find all elements (relative to root element) with tag "planet" under "innerplanets" element
- ③ find all elements with tag "planet" under "outerplanets" element
- ④ loop through search results
- ⑤ print "name" attribute of planet element

xml_planets_xpath1.py

```
Inner:
    Mercury
    Venus
    Earth
    Mars
Outer:
    Jupiter
    Saturn
    Uranus
    Neptune
```

Example

xml_planets_xpath2.py

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

# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')

jupiter = doc.find('../planet[@planetname="Jupiter"]')

if jupiter is not None:
    for moon in jupiter:
        print(moon.text) # grab attribute
```

xml_planets_xpath2.py

```
Metis
Adrastea
Amalthea
Thebe
Io
Europa
Ganymede
Callisto
Themisto
Himalia
Lysithea
Elara
```

Table 20. *ElementTree XPath Summary*

Syntax	Meaning
tag	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
*	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
.	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
//	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “//egg” selects all “egg” elements in the entire tree.
..	Selects the parent element.
[@attrib]	Selects all elements that have the given attribute. For example, “//a[@href]” selects all “a” elements in the tree that has a “href” attribute.
[@attrib=’value’]	Selects all elements for which the given attribute has the given value. For example, “//div[@class=’sidebar’]” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
[tag]	Selects all elements that has a child element named tag. In the current version, only a single tag can be used (i.e. only immediate children are supported).

About JSON

- Lightweight, human-friendly format for data
- Contains dictionaries and lists
- Stands for JavaScript Object Notation
- Looks like Python
- Basic types: Number, String, Boolean, Array, Object
- White space is ignored
- Stricter rules than Python

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001.

A JSON file contains objects and arrays, which correspond exactly to Python dictionaries and lists.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; Booleans are represented by true or false; null (None in Python) is represented by null.

Reading JSON

- `json` module in standard library
- `json.load()` parse from file-like object
- `json.loads()` parse from string
- Both methods return Python dict or list

To read a JSON file, import the `json` module. Use `json.loads()` to parse a string containing valid JSON. Use `json.load()` to read JSON from a file-like object⁰.

Both methods return a Python dictionary containing all the data from the JSON file.

Example

json_read.py

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

import json

with open('../DATA/solar.json') as solar_in: ①
    solar = json.load(solar_in) ②

# json.loads(String)
# json.load(FILE_OBJECT)

# print(solar)

print(solar['innerplanets']) ③
print('*' * 60)
print(solar['innerplanets'][0]['name'])
print('*' * 60)
for planet in solar['innerplanets'] + solar['outerplanets']:
    print(planet['name'])

print("*" * 60)
for group in solar:
    if group.endswith('planets'):
        for planet in solar[group]:
            print(planet['name'])
```

- ① open JSON file for reading
- ② load from file object and convert to Python data structure
- ③ solar is just a Python dictionary

json_read.py

```
[{'name': 'Mercury', 'moons': None}, {'name': 'Venus', 'moons': None}, {'name':  
'Earth', 'moons': ['moon']}, {'name': 'Mars', 'moons': ['Deimos', 'Phobos']}]
```

```
*****
```

```
Mercury
```

```
*****
```

```
Mercury
```

```
Venus
```

```
Earth
```

```
Mars
```

```
Jupiter
```

```
Saturn
```

```
Uranus
```

```
Neptune
```

```
*****
```

```
Mercury
```

```
Venus
```

```
Earth
```

```
Mars
```

```
Jupiter
```

```
Saturn
```

```
Uranus
```

```
Neptune
```

```
Pluto
```

Writing JSON

- Use `json.dumps()` or `json.dump()`

To output JSON to a string, use `json.dumps()`. To output JSON to a file, pass a file-like object to `json.dump()`. In both cases, pass a Python data structure as the data to be output.

Example

`json_write.py`

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

import json

george = [
    {
        'num': 1,
        'lname': 'Washington',
        'fname': 'George',
        'dstart': [1789, 4, 30],
        'dend': [1797, 3, 4],
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'dbirth': [1732, 2, 22],
        'ddeath': [1799, 12, 14],
        'assassinated': False,
        'party': None,
    },
    {
        'spam': 'ham',
        'eggs': [1.2, 2.3, 3.4],
        'toast': {'a': 5, 'm': 9, 'c': 4},
    }
] ①

js = json.dumps(george, indent=4) ②
print(js)

with open('george.json', 'w') as george_out: ③
    json.dump(george, george_out, indent=4) ④
```


- ① Python data structure
- ② dump structure to JSON string
- ③ open file for writing
- ④ dump structure to JSON file using open file object

json_write.py

```
[
    {
        "num": 1,
        "lname": "Washington",
        "fname": "George",
        "dstart": [
            1789,
            4,
            30
        ],
        "dend": [
            1797,
            3,
            4
        ],
        "birthplace": "Westmoreland County",
        "birthstate": "Virginia",
        "dbirth": [
            1732,
            2,
            22
        ],
        "ddeath": [
            1799,
            12,
            14
        ],
        "assassinated": false,
        "party": null
    },
    {
        "spam": "ham",
        "eggs": [
            1.2,
            2.3,
```

```
3.4
],
"toast": {
    "a": 5,
    "m": 9,
    "c": 4
}
}
```

Customizing JSON

- JSON data types limited
- simple cases — dump dict
- create custom encoders

The JSON spec only supports a limited number of datatypes. If you try to dump a data structure contains dates, user-defined classes, or many other types, the json encoder will not be able to handle it.

You can a custom encoder for various data types. To do this, write a function that expects one Python object, and returns some object that JSON can parse, such as a string or dictionary. The function can be called anything. Specify the function with the **default** parameter to `json.dump()`.

The function should check the type of the object. If it is a type that needs specially handling, return a JSON-friendly version, otherwise just return the original object.

Table 21. Python types that JSON can encode

Python	JSON
dict	object
list	array
str	string
int	number (int)
float	number (real)
True	true
False	false
None	null

NOTE

see the file [json_custom singledispatch.py](#) in EXAMPLES for how to use the **singledispatch** module (which is not in the standard library) to handle multiple data types.

Example

json_custom_encoding.py

```
#!/usr/bin/env python
#
import json
from datetime import date

class Parrot(): ①
    def __init__(self, name, color):
        self._name = name
        self._color = color

    @property
    def name(self): ②
        return self._name

    @property
    def color(self):
        return self._color

parrots = [ ③
    Parrot('Polly', 'green'), #
    Parrot('Peggy', 'blue'),
    Parrot('Roger', 'red'),
]

def encode(obj): ④
    if isinstance(obj, date): ⑤
        return obj.ctime() ⑥
    elif isinstance(obj, Parrot): ⑦
        return {'name': obj.name, 'color': obj.color} ⑧
    return obj ⑨

data = { ⑩
    'spam': [1, 2, 3],
    'ham': ('a', 'b', 'c'),
    'toast': date(2014, 8, 1),
    'parrots': parrots,
```

```
}
```

```
print(json.dumps(data, default=encode, indent=4)) ⑪
```

- ① sample user-defined class (not JSON-serializable)
- ② JSON does not understand arbitrary properties
- ③ list of Parrot objects
- ④ custom JSON encoder function
- ⑤ check for date object
- ⑥ convert date to string
- ⑦ check for Parrot object
- ⑧ convert Parrot to dictionary
- ⑨ if not processed, return object for JSON to parse with default parser
- ⑩ dictionary of arbitrary data
- ⑪ convert Python data to JSON data; 'default' parameter specifies function for custom encoding; 'indent' parameter says to indent and add newlines for readability

json_custom_encoding.py

```
{
  "spam": [
    1,
    2,
    3
  ],
  "ham": [
    "a",
    "b",
    "c"
  ],
  "toast": "Fri Aug  1 00:00:00 2014",
  "parrots": [
    {
      "name": "Polly",
      "color": "green"
    },
    {
      "name": "Peggy",
      "color": "blue"
    },
    {
      "name": "Roger",
      "color": "red"
    }
  ]
}
```

Reading and writing YAML

- `yaml` module from PYPI
- Same syntax as JSON
- `yaml.load()`, `dump()` parse from/to file-like object
- `yaml.loads()`, `dumps()` parse from/to string

Reading and writing YAML uses the same syntax as JSON, other than using the **`yaml`** module, which is NOT in the standard library. To install the **`yaml`** module:

```
pip install pyyaml
```

To read a YAML file (or string) into a Python data structure, use `yaml.load(file_object)` or `yaml.loads(string)`.

To write a data structure to a YAML file or string, use `yaml.dump(data, file_object)` or `yaml.dumps(data)`.

You can also write custom YAML processors.

Example

yaml_read_solar.py

```
#!/usr/bin/env python
# (c) 2015 John Strickler
#
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in)

star = solar_data['star']
print("Our star is {}\n".format(star))

for section in PLANET_SECTIONS:
    for planet in solar_data[section]:
        print(planet['name'])
        for moon in planet['moons']:
            print("\t{}".format(moon))
```


yaml_read_solar.py

Our star is Sun

Mercury

None

Venus

None

Earth

Moon

Mars

Deimos

Phobos

Metis

Jupiter

Adrastea

Amalthea

Thebe

Io

Europa

Ganymede

Callisto

Themisto

Himalia

Lysithea

Elara

Saturn

Rhea

Hyperion

Titan

Iapetus

Mimas

Enceladus

Tethys

Dione

Enceladus

Uranus

Ariel

Umbriel

Miranda

Titania

Oberon

...

Example

yaml_create_file.py

```
#!/usr/bin/env python
# (c) 2015 John Strickler
#
import sys
from datetime import date
import yaml

potus = {
    1: {
        'lname': 'Washington',
        'fname': 'George',
        'dob': date(1732, 2, 22),
        'dod': date(1799, 12, 14),
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'term': [ date(1789, 4, 30), date(1797, 3, 4) ],
        'assassinated': False,
        'party': None,
    },
    2: {
        'lname': 'Adams',
        'fname': 'John',
        'dob': date(1735, 10, 30),
        'dod': date(1826, 7, 4),
        'birthplace': 'Braintree, Norfolk',
        'birthstate': 'Massachusetts',
        'term': [date(1797, 3, 4), date(1801, 3, 4)],
        'assassinated': False,
        'party': 'Federalist',
    },
}

with open('potus.yaml', 'w') as potus_out:
    yaml.dump(potus, potus_out)

yaml.dump(potus, sys.stdout)
```

yaml_create_file.py

```
1:
  assassinated: false
  birthplace: Westmoreland County
  birthstate: Virginia
  dob: 1732-02-22
  dod: 1799-12-14
  fname: George
  lname: Washington
  party: null
  term: [1789-04-30, 1797-03-04]
2:
  assassinated: false
  birthplace: Braintree, Norfolk
  birthstate: Massachusetts
  dob: 1735-10-30
  dod: 1826-07-04
  fname: John
  lname: Adams
  party: Federalist
  term: [1797-03-04, 1801-03-04]
```

Reading CSV data

- Use csv module
- Create a reader with any iterable (e.g. file object)
- Understands Excel CSV and tab-delimited files
- Can specify alternate configuration
- Iterate through reader to get rows as lists of columns

To read CSV data, use the `reader()` method in the `csv` module.

To create a reader with the default settings, use the `reader()` constructor. Pass in an iterable – typically, but not necessarily, a file object.

You can also add parameters to control the type of quoting, or the output delimiters.

Example

`csv_read.py`

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

with open('../DATA/knights.csv') as knights_in:
    rdr = csv.reader(knights_in) ①
    for name, title, color, quest, comment, number, ladies in rdr: ②
        print('{:4s} {:9s} {}'.format(
            title, name, quest
        ))
```

① create CSV reader

② Read and unpack records one at a time; each record is a list

csv_read.py

```
King Arthur    The Grail
Sir Lancelot   The Grail
Sir Robin      Not Sure
Sir Bedevere   The Grail
Sir Gawain     The Grail
```

...

Nonstandard CSV

- Variations in how CSV data is written
- Most common alternate is for Excel
- Add parameters to reader/writer

You can customize how the CSV parser and generator work by passing extra parameters to `csv.reader()` or `csv.writer()`. You can change the field and row delimiters, the escape character, and for output, what level of quoting.

You can also create a "dialect", which is a custom set of CSV parameters. The `csv` module includes one extra dialect, **excel**, which handles CSV files generated by Microsoft Excel. To use it, specify the *dialect* parameter:

```
rdr = csv.reader(csvfile, dialect='excel')
```

Table 22. CSV reader()/writer() Parameters

Parameter	Meaning
quotechar	One-character string to use as quoting character (default: '"')
delimiter	One-character string to use as field separator (default: ',')
skipinitialspace	If True, skip white space after field separator (default: False)
lineterminator	The character sequence which terminates rows (default: depends on OS)
quoting	When should quotes be generated when writing CSV <code>csv.QUOTE_MINIMAL</code> – only when needed (default) <code>csv.QUOTE_ALL</code> – quote all fields <code>csv.QUOTE_NONNUMERIC</code> – quote all fields that are not numbers <code>csv.QUOTE_NONE</code> – never put quotes around fields
escapechar	One-character string to escape delimiter when quoting is set to <code>csv.QUOTE_NONE</code>
doublequote	Control quote handling inside fields. When True, two consecutive quotes are read as one, and one quote is written as two. (default: True)

Example

csv_nonstandard.py

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

with open('../DATA/computer_people.txt') as computer_people_in:
    rdr = csv.reader(computer_people_in, delimiter=';') ①

    for first_name, last_name, known_for, birth_date in rdr: ②
        print('{}: {}'.format(last_name, known_for))
```

① create CSV reader from open file object

② iterate over rows of data — csv reader is a generator

csv_nonstandard.py

```
Gates: Gates Foundation
Jobs: Apple
Wall: Perl
Allen: Microsoft
Ellison: Oracle
Gates: Microsoft
Zuckerberg: Facebook
Brin: Google
Page: Google
Torvalds: Linux
```


Using csv.DictReader

- Returns each row as dictionary
- Keys are field names
- Use header or specify

Instead of the normal reader, you can create a dictionary-based reader by using the DictReader class.

If the CSV file has a header, it will parse the header line and use it as the field names. Otherwise, you can specify a list of field names with the **fieldnames** parameter. For each row, you can look up a field by name, rather than position.

Example

csv_dictreader.py

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

field_names = ['term', 'firstname', 'lastname', 'birthplace', 'state', 'party'] ①

with open('../DATA/presidents.csv') as presidents_in:
    rdr = csv.DictReader(presidents_in, fieldnames=field_names) ②
    for row in rdr: ③
        print('{firstname:25s} {lastname:12s} {party}'.format(**row)) ④
```

- ① field names, which will become dictionary keys on each row
- ② create reader, passing in field names (if not specified, uses first row as field names)
- ③ iterate over rows in file
- ④ ****row** converts row to individual parameters

csv_dictreader.py

George	Washington	no party
John	Adams	Federalist
Thomas	Jefferson	Democratic - Republican
James	Madison	Democratic - Republican
James	Monroe	Democratic - Republican
John Quincy	Adams	Democratic - Republican
Andrew	Jackson	Democratic
Martin	Van Buren	Democratic
William Henry	Harrison	Whig
John	Tyler	Whig
James Knox	Polk	Democratic
Zachary	Taylor	Whig
Millard	Fillmore	Whig
Franklin	Pierce	Democratic
James	Buchanan	Democratic
Abraham	Lincoln	Republican
Andrew	Johnson	Republican
Ulysses Simpson	Grant	Republican
Rutherford Birchard	Hayes	Republican
James Abram	Garfield	Republican

...

Writing CSV Data

- Use `csv.writer()`
- Parameter is file-like object (must implement `write()` method)
- Can specify parameters to writer constructor
- Use `writerow()` or `writerows()` to output CSV data

To output data in CSV format, first create a writer using `csv.writer()`. Pass in a file-like object.

For each row to write, call the `writerow()` method of the writer, passing in an iterable with the values for that row.

To modify how data is written out, pass parameters to the writer.

TIP

On Windows, to prevent doublespaced output, add `lineterminator='\n'` when creating a CSV writer.

Example

csv_write.py

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

data = (
    ('February', 28, 'The shortest month, with 28 or 29 days'),
    ('March', 31, 'Goes out like a "lamb"'),
    ('April', 30, 'Its showers bring May flowers'),
)

with open('../TEMP/stuff.csv', 'w') as stuff_in:
    if sys.platform == 'win32':
        wtr = csv.writer(stuff_in, lineterminator='\n') ①
    else:
        wtr = csv.writer(stuff_in) ①
    for row in data:
        wtr.writerow(row) ②
```

- ① create CSV writer from file object that is opened for writing; on windows, need to set output line terminator to '\n'
- ② write one row (of iterables) to output file

Pickle

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Python uses the pickle module for data serialization.

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`. Both functions take a data structure as the first argument. `dumps()` returns the pickled data as a string. `dump()` writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use `pickle.load()`, which takes a file-like object that has been open for writing, or `pickle.loads()` which reads from a string. Both functions return the original data structure that had been pickled.

NOTE

The syntax of the `json` module is based on the `pickle` module.

Example

pickling.py

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

@author: jstrick
Created on Sat Mar 16 00:47:05 2013

"""
import pickle
from pprint import pprint

①
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

data = [ ②
    colors,
    airports,
]

with open('../TEMP/pickled_data.pic', 'wb') as pic_out: ③
    pickle.dump(data, pic_out) ④

with open('../TEMP/pickled_data.pic', 'rb') as pic_in: ⑤
    pickled_data = pickle.load(pic_in) ⑥

pprint(pickled_data) ⑦
```

- ① some data structures
- ② list of data structures
- ③ open pickle file for writing in binary mode
- ④ serialize data structures to pickle file
- ⑤ open pickle file for reading in binary mode
- ⑥ de-serialize pickle file back into data structures
- ⑦ view data structures

pickling.py

```
[[ 'red',  
  'blue',  
  'green',  
  'yellow',  
  'black',  
  'white',  
  'orange',  
  'brown',  
  'purple'],  
 { 'EWR': 'Newark',  
   'IAD': 'Dulles',  
   'LAX': 'Los Angeles',  
   'MGW': 'Morgantown',  
   'ORD': 'Chicago',  
   'RDU': 'Raleigh-Durham' } ]]
```

Chapter 14 Exercises

Exercise 14-1 (xwords.py)

Using ElementTree, create a new XML file containing all the words that start with 'x' from words.txt. The root tag should be named 'words', and each word should be contained in a 'word' tag. The finished file should look like this:

```
<words>
  <word>xanthan</word>
  <word>xanthans</word>
  and so forth
</words>
```

Exercise 14-2 (xpresidents.py)

Use ElementTree to parse presidents.xml. Loop through and print out each president's first and last names and their state of birth.

Exercise 14-3 (jpresidents.py)

Rewrite xpresidents.py to parse presidents.json using the json module.

Exercise 14-4 (cpresidents.py)

Rewrite xpresidents.py to parse presidents.csv using the csv module.

Exercise 14-5 (pickle_potus.py)

Write a script which reads the data from presidents.csv into an dictionary where the key is the term number, and the value is another dictionary of data for one president.

Using the pickle module, Write the entire dictionary out to a file named presidents.pic.

Exercise 14-6 (unpickle_potus.py)

Write a script to open presidents.pic, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

Chapter 15: iPython and Jupyter

Objectives

- Learn the basics of iPython
- Apply magics
- Use Jupyter notebooks

About iPython

- Enhanced interpreter for exploratory computing
- Efficient for data analysis (what-if? Scenarios)
- Great for plotting-tweaking-plotting
- Good for simple code development
- Not so good for enterprise development

iPython is an enhanced interpreter for Python. It provides a large number of "creature comforts" for the user, such as name completion and improved help features.

It's great for ad-hoc queries, what-if scenarios, when you are not developing a full application.

Features of iPython

- Name completion (variables, modules, methods, etc.)
- Autoindent
- Inline plots and other figures
- Dynamic introspection (dir() on steroids)
- Search namespaces with wildcards
- Auto-parentheses ('sin 3' becomes 'sin(3)')
- Auto-quoting ('foo a b' becomes 'foo("a","b")')
- Enhanced help system
- Commands are numbered (and persistent) for recall
- 'Magic' commands for controlling iPython itself
- Aliasing system for interpreter commands
- Easy access to shell commands
- Background execution in separate thread
- Simplified (and lightweight) persistence
- Session logging (can be saved as scripts)
- Detailed tracebacks when errors occur
- Session restoring (playback log to specific state)
- Flexible configuration system
- Easy access to Python debugger
- Simple profiling
- Interactive parallel computing (if supported by hardware)

The many faces of iPython

- Console
- Colorized console (default)
- QtConsole (obsolete)
- Jupyter notebook

iPython can be run in several different modes. The most basic mode is console, which runs from the command prompt, without syntax colorizing.

If your terminal supports it (and most do), iPython can run as a colorized console, using different colors for variables, functions, strings, comments, and so forth.

Both the default console and the colorized console display plots in a separate popup window.

If QT is installed, iPython can run a GUI console, which looks and acts like a regular text console, but allows plots to be generated inline, and has enhanced context-sensitive help. This mode is deprecated, as it is simpler to just run a notebook.

The most flexible and powerful way to run iPython is in notebook mode. This mode starts a dedicated web server and begins a session using your default web browser. Other users can load notebooks from the notebook server, similarly to MatLab. The notebook part of iPython is now called Jupyter.

NOTE | [iPython also supports parallel computing, which will not be covered here.](#)

Starting iPython

- Type ipython at the command line
- Huge number of options

To get started with iPython in console mode, just type ipython at the command line, or double-click the icon in Windows explorer.

In general, it works like the normal interpreter, but with many more features.

There is a huge number of options. To see them all, invoke iPython with the `--help-all` option:

```
ipython --help-all
```

Getting Help

- `? basic help`
- `%quickref quick reference`
- `help standard Python help`
- `thing? help on thing`

iPython provides help in several ways.

Typing `?` at the prompt will display an introduction to iPython and a feature overview.

For a quick reference card, type `%quickref`.

To start Python's normal help system, type `help`.

For help on any Python object, type `object?`. This is similar to saying `help("object")` in the default interpreter, but is "smarter".

For more help on an object, add a second question mark:

Tab Completion

- Press tab to complete
- keywords
- modules
- methods and attributes
- parameters to functions
- file and directory names

Pressing the <TAB> key will invoke autocomplete. If there is only one possible completion, it will be expanded. If there is more than one completion that will match, iPython will display a list of possible completions.

Autocomplete works on keywords, functions, classes, methods, and object attributes, as well as paths from your file system.

Magic Commands

- Start with % (line magic) or %% (cell magic)
- Simplify common tasks
- Use %lsmagic to list all magic commands

One of the enhancements in iPython is the set of "magic" commands. These are meta-commands that help you manipulate the iPython environment.

Normal magics apply to a single line. Cell magics apply to a cell (a group of lines).

For instance, %history will list previous commands.

TIP | [Type %lsmagic for a list of all magics](#)

Benchmarking

- Use %timeit

iPython has a handy magic for benchmarking.

```
In [1]: color_values = { 'red':66, 'green':85, 'blue':77 }
```

```
In [2]: %timeit red_value = color_values['red']  
10000000 loops, best of 3: 54.5 ns per loop
```

```
In [3]: %timeit red_value = color_values.get('red')  
10000000 loops, best of 3: 115 ns per loop
```

%timeit will benchmark whatever code comes after it on the same line. %%timeit will benchmark contents of a notebook cell

External commands

- Precede command with !
- Can assign output to variable

Any shell command can be run by starting it with a !.

Windows

```
In [3]: !dir DATA\*.csv
Volume in drive Z is Shared Folders
Volume Serial Number is 0000-0064

Directory of Z:\Desktop\py2forsci\DATA

02/20/2014  01:53 PM                5,511 airport_boardings.csv
02/20/2014  01:53 PM                2,182 energy_use_quad.csv
02/20/2014  01:53 PM                4,993 parasite_data.csv
               3 File(s)            12,686 bytes
               0 Dir(s)  352,625,324,032 bytes free
```

In [4]:

Non-Windows (Linux, OS X, etc)

```
In [2]: !ls -l DATA/*.csv
-rwxr-xr-x  1 jstrick  staff  5511 Jan 27 19:44 DATA/airport_boardings.csv
-rwxr-xr-x  1 jstrick  staff  2182 Jan 27 19:44 DATA/energy_use_quad.csv
-rwxr-xr-x  1 jstrick  staff  4642 Jan 27 19:44 DATA/parasite_data.csv
```

In [3]:

Enhanced help

- Use object?
- Output of `dir()` is cleaner
- Multidimensional data structures are prettyprinted

iPython makes it easier to get at metadata. Type an object's name followed by a question mark will provide a description of the object plus any available help. It is similar to, but much better than, the `help()` function in the standard interpreter.

Dictionaries and multidimensional data structures are prettyprinted – items are printed on separate lines.

Jupyter notebooks

- Extension of iPython
- Puts the interpreter in a web browser
- Code is grouped into "cells"
- Cells can be edited, repeated, etc.

In 2015, the developers of iPython pulled the notebook feature out of iPython to make a separate product called Jupyter. It is still invoked via the `ipython` command, but now supports many languages other than Python.

A Jupyter notebook is a journal-like python interpreter that lives in a browser window. Code is grouped into cells, which can contain multiple statements. Cells can be edited, repeated, rearranged, and otherwise manipulated.

Plots will be shown inline by default.

A notebook (i.e, a set of cells, can be saved, and reopened). Notebooks can be shared among members of a team via the notebook server which is built into jupyter.

Jupyter Demo

At this point please start the Jupyter notebook server and follow along with a demo of Jupyter notebooks as directed by the instructor

```
cd NOTEBOOKS
jupyter notebook
```

Chapter 16: Introduction to numpy

Objectives

- See the "big picture" of numpy
- Create and manipulate arrays
- Learn different ways to initialize arrays
- Understand the numpy data types available
- Work with shapes, dimensions, and ranks
- Broadcast changes across multiple array dimensions
- Extract multidimensional slices
- Perform matrix operations

Python's scientific stack

- NumPy, SciPy, Matplotlib (and many others)
- Python extensions, written in C/Fortran
- Support for math, numerical, and scientific operations

NumPy is part of what is sometimes called Python's "scientific stack". Along with SciPy, Matplotlib, and other libraries, it provides a broad range of support for scientific and engineering tasks.

There is no one integrated tool for these libraries. You can either create scripts with your choice of IDE, or you can use iPython to manipulate data interactively and ad hoc.

NumPy overview

- Install numpy module from numpy.scipy.org
- Basic object is the array
- Up to 100x faster than normal Python math operations
- Functional-based (fewer loops)
- Dozens of utility functions

The basic object that NumPy provides is the array. Arrays can have as many dimensions as needed. Working with NumPy arrays can be 100 times faster than working with normal Python lists.

Operations are applied to arrays in a functional manner – instead of the programmer explicitly looping through elements of the array, the programmer specifies an expression or function to be applied, and the array object does all of the iteration internally.

There are many utility functions for accessing arrays, for creating arrays with specified values, and for performing standard numerical operations on arrays.

To get started, import the numpy module. It is conventional to import numpy as np. The examples in this chapter will follow that convention.

NOTE

all top-level numpy routines are also available directly through the [scipy package](#).

Creating Arrays

- Create with
 - `array()` function initialized with nested sequences
 - Other utilities (`arange()`, `zeros()`, `ones()`, `empty()`)
- All elements are same type (default float)
- Useful properties: `ndim`, `shape`, `size`, `dtype`
- Can have any number of axes (dimensions)
- Each axis has a length

An array is the most basic object in numpy. It is a table of numbers, indexed by positive integers. All of the elements of an array are of the same type.

An array can have any number of dimensions; these are referred to as axes. The number of axes is called the rank.

Arrays are rectangular, not ragged.

One way to create an array is with the `array()` function, which can be initialized from a sequence of sequences.

The `zeros()` function expects a list of axis lengths, and creates the corresponding array, with all values set to zero. The `ones()` function is the same, but initializes with ones.

The `full()` function expects a tuple of axis lengths, and a value. It creates the array, putting the specified value in every element.

The `empty()` function creates an array initialized with random floats.

When you print an array, NumPy displays it with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

NOTE the `ndarray()` object is initialized with the *shape*, not the *data*.

Example

np_create_arrays.py

```
#!/usr/bin/env python
import numpy as np

a = np.array([[1, 2.1, 3], [4, 5, 6], [7, 8, 9], [20, 30, 40]]) ①
print(a)
print("# dims", a.ndim) ②
print("shape", a.shape) ③
print()

a_zeros = np.zeros((3, 5), dtype=np.uint32) ④
print(a_zeros)
print()

a_ones = np.ones((2, 3, 4, 5)) ⑤
print(a_ones)
print()

# with uninitialized values
a_empty = np.empty((3, 8)) ⑥
print(a_empty)

print(a.dtype) ⑦

nan_array = np.full((5, 10), np.NaN) ⑧
print(nan_array)
```

- ① create array from nested sequences
- ② get number of dimensions
- ③ get shape
- ④ create array of specified shape and datatype, initialized to zeroes
- ⑤ create array of specified shape, initialized to ones
- ⑥ create uninitialized array of specified shape
- ⑦ defaults to float64
- ⑧ create array of NaN values

np_create_arrays.py

```

[[ 1.  2.1  3. ]
 [ 4.  5.  6. ]
 [ 7.  8.  9. ]
 [20. 30. 40. ]]
# dims 2
shape (4, 3)

[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]

[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]]

[[[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]

 [[1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]
  [1. 1. 1. 1. 1.]]]]

[[2.14731625e-314 2.13451557e-314 2.14733501e-314 2.14733504e-314
 2.14733507e-314 2.14733517e-314 2.14732169e-314 2.13393413e-314]

```

```
[2.13471203e-314 2.14733530e-314 2.14733533e-314 2.13446830e-314
 2.14718577e-314 2.14733539e-314 2.14733542e-314 2.14733555e-314]
[2.14733567e-314 2.13446960e-314 2.13586030e-314 2.13458635e-314
 2.13458357e-314 2.13446949e-314 2.14733577e-314 2.14409376e-314]]
float64
[[nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan]
 [nan nan nan nan nan nan nan nan nan]]
```

Creating ranges

- Similar to builtin `range()`
- Returns a one-dimensional numpy array
- Can use floating point values
- Can be reshaped

The `arange()` function takes a size, and returns a one-dimensional numpy array. This array can then be reshaped as needed. The start, stop, and step parameters are similar to those of `range()`.

The `linspace()` function creates a specified number of equally-spaced values.

The resulting arrays can be reshaped into multidimensional arrays.

Example

np_create_ranges.py

```
#!/usr/bin/env python
import numpy as np

r1 = np.arange(50) ①
print(r1)
print("size is", r1.size) ②
print()

r2 = np.arange(5, 101, 5) ③
print(r2)
print("size is", r2.size)
print()

r3 = np.arange(1.0, 5.0, .333333) ④
print(r3)
print("size is", r3.size)
print()

r4 = np.linspace(1.0, 2.0, 10) ⑤
print(r4)
print("size is", r4.size)
print()
```

- ① create range of ints from 0 to 49
- ② size is 50
- ③ create range of ints from 5 to 100 counting by 5
- ④ start, stop, and step may be floats
- ⑤ 10 equal steps between 1.0 and 2.0

np_create_ranges.py

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49]
size is 50

[  5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90
 95 100]
size is 20

[1.          1.3333333 1.6666666 1.9999999 2.3333332 2.6666665 2.9999998
 3.3333331 3.6666664 3.9999997 4.333333  4.6666663 4.9999996]
size is 13

[1.          1.1111111 1.2222222 1.3333333 1.4444444 1.5555555
 1.6666667 1.7777778 1.8888889 2.          ]
size is 10
```

Working with arrays

- Use normal math operators (+, -, /, and *)
- Use NumPy's builtin functions
- By default, apply to every element
- Can apply to single axis
- Operations on between two arrays applies operator to pairs of element

The array object is smart about applying functions and operators. A function applied to an array is applied to every element of the array. An operator applied to two arrays is applied to corresponding elements of the two arrays.

Some builtin array functions:

all, alltrue, any, apply, arange, argmax, argmin, argsort, average, bincount, ceil, clip, conj, conjugate, corrcoef, cov, cross, cumprod, cumsum, diff, dot, floor, inner, inv, lexsort, max, maximum, mean, median, min, minimum, nonzero, outer, prod, re, round, sometrue, sort, std, sum, trace, transpose, var, vdot, vectorize, where

Example

`np_basic_array_ops.py`

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [
        [5, 10, 15],
        [2, 4, 6],
        [3, 6, 9, ],
    ]
) ①

b = np.array(
    [
        [10, 85, 92],
        [77, 16, 14],
    ]
)
```

```

        [19, 52, 23],
    ]
) ②
print("a")
print(a)
print()
print("b")
print(b)
print()

print("a * 10")
print(a * 10) ③
print()

print("a + b")
print(a + b) ④
print()

print("b + 3")
print(b + 3) ⑤
print()

s1 = a.sum() ⑥
s2 = b.sum() ⑥
print("sum of a is {0}; sum of b is {1}".format(s1, s2))
print()

a += 1000 ⑦
print(a)

```

- ① create 2D array
- ② create another 2D array
- ③ multiply every element by 10 (not in place)
- ④ add every element of a to the corresponding element of b
- ⑤ add 3 to every element of b
- ⑥ calculate sum of all elements
- ⑦ add 1000 to every element of a (in place)

np_basic_array_ops.py

```
a
[[ 5 10 15]
 [ 2  4  6]
 [ 3  6  9]]

b
[[10 85 92]
 [77 16 14]
 [19 52 23]]

a * 10
[[ 50 100 150]
 [ 20  40  60]
 [ 30  60  90]]

a + b
[[ 15  95 107]
 [ 79  20  20]
 [ 22  58  32]]

b + 3
[[13 88 95]
 [80 19 17]
 [22 55 26]]

sum of a is 60; sum of b is 388

[[1005 1010 1015]
 [1002 1004 1006]
 [1003 1006 1009]]
```

Shapes

- Number of elements on each axis
- `array.shape` has shape tuple
- Assign to `array.shape` to change
- `array.ravel()` to flatten to one dimension
- `array.transpose()` to flip the shape

Every array has a shape, which is the number of elements on each axis. For instance, an array might have the shape (3,5), which means that there are 3 rows and 5 columns.

The shape is stored as a tuple, in the shape attribute of an array. To change the shape of an array, assign to the shape attribute.

The `ravel()` method will flatten any array into a single dimension. The `transpose()` method will flip shape (x,y) to shape (y,x). It is equivalent to `array.shape = reversed(array.shape)`.

Example

np_shapes.py

```
#!/usr/bin/env python
import numpy as np

a1 = np.arange(15) ①
print("a1 shape", a1.shape) ②
print()

print(a1)
print()

a1.shape = 3, 5 ③
print(a1)
print()

a1.shape = 5, 3 ④
print(a1)
print()

print(a1.flatten()) ⑤
print()

print(a1.transpose()) ⑥
print("-----")

a2 = np.arange(40) ⑦
a2.shape = 2, 5, 4 ⑧

print(a2)
print()
```

- ① create 1D array
- ② get shape
- ③ reshape to 3x5
- ④ reshape to 5x3
- ⑤ print array as 1D
- ⑥ print transposed array
- ⑦ create 1D array
- ⑧ reshape to 2x5x4

np_shapes.py

```
a1 shape (15,)

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]]

[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]

[[ 0  3  6  9 12]
 [ 1  4  7 10 13]
 [ 2  5  8 11 14]]
-----
[[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]]

 [[20 21 22 23]
 [24 25 26 27]
 [28 29 30 31]
 [32 33 34 35]
 [36 37 38 39]]]
```

Slicing and indexing

- Simple indexing similar to lists
- start, stop, step
- start is INclusive, stop is Exclusive
- : used for range for one axis
- ... means "as many : as needed"

NumPy arrays can be indexed and sliced like regular Python lists, but with some convenient extensions. Instead of `[x][y]`, numpy arrays can be indexed with `[x,y]`. Within an axis, ranges can be specified with slice notation (start:stop:step) as usual.

For arrays with more than 2 dimensions, `'...'` can be used to mean "and all the other dimensions".

Example

np_indexing.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
) ①

print(a)
print()

print('a[0] =>', a[0]) ②
print('a[0][0] =>', a[0][0]) ③
print('a[0,0] =>', a[0, 0]) ④
print('a[0,:3] =>', a[0, :3]) ⑤
print('a[0,::2] =>', a[0, ::2]) ⑥
print()
print('a[:,2] =>', a[:,2]) ⑦
print()
print('a[:3, -2:] =>', a[:3, -2:]) ⑧
```

- ① sample data
- ② first row
- ③ first element of first row
- ④ same, but numpy style
- ⑤ first 3 elements of first row
- ⑥ every second element of first row
- ⑦ every second row
- ⑧ every third element of every second row

np_indexing.py

```
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]
 [75 37 11 62 35 76 38  4]]

a[0] => [70 31 21 76 19  5 54 66]
a[0][0] => 70
a[0,0] => 70
a[0,:3] => [70 31 21]
a[0,::2] => [70 21 19 54]

a[:,::2] => [[70 31 21 76 19  5 54 66]
 [11 84  7 10 31 50 11 98]
 [75 37 11 62 35 76 38  4]]

a[:3, -2:] => [[54 66]
 [65 73]
 [11 98]]
```


Indexing with Booleans

- Apply relational expression to array
- Result is array of Booleans
- Booleans can be used to index original array

If a relational expression ($>$, $<$, $>=$, $<=$) is applied to an array, the result is a new array containing booleans reflecting whether the expression was true for each element. That is, for each element of the original array, the resulting array is set to True if the expression is true for that element, and False otherwise.

The resulting Boolean array can then be used as an index, to modify just the elements for which the expression was true.

Example

np_bool_indexing.py

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73],
     [11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90],
     [75, 37, 11, 62, 35, 76, 38, 4]]
) ①

print('a =>', a, '\n')

i = a > 50 ②
print('i (a > 50) =>', i, '\n')

print('a[i] =>', a[i], '\n') ③

print('a[a > 50] =>', a[a > 50], '\n') ④

print('a[i].min(), a[i].max() =>', a[i].min(), a[i].max(), '\n') ⑤

a[i] = 0 ⑥
print('a =>', a, '\n')

print("a[a < 15] += 10")
a[a < 15] += 10 ⑦
print(a, '\n')
```

- ① sample data
- ② create Boolean mask
- ③ print elements of a that are > 50 using mask
- ④ same, but without creating a separate mask
- ⑤ min and max values of result set with values less than 50
- ⑥ set elements with value > 50 to 0
- ⑦ add 10 to elements < 15

np_bool_indexing.py

```
a => [[70 31 21 76 19  5 54 66]
      [23 29 71 12 27 74 65 73]
      [11 84  7 10 31 50 11 98]
      [25 13 43  1 31 52 41 90]
      [75 37 11 62 35 76 38  4]]

i (a > 50) => [[ True False False  True False False  True  True]
               [False False  True False False  True  True  True]
               [False  True False False False False False  True]
               [False False False False False  True False  True]
               [ True False False  True False  True False False]]

a[i] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[a > 50] => [70 76 54 66 71 74 65 73 84 98 52 90 75 62 76]

a[i].min(), a[i].max() => 52 98

a => [[ 0 31 21  0 19  5  0  0]
      [23 29  0 12 27  0  0  0]
      [11  0  7 10 31 50 11  0]
      [25 13 43  1 31  0 41  0]
      [ 0 37 11  0 35  0 38  4]]

a[a < 15] += 10
[[10 31 21 10 19 15 10 10]
 [23 29 10 22 27 10 10 10]
 [21 10 17 20 31 50 21 10]
 [25 23 43 11 31 10 41 10]
 [10 37 21 10 35 10 38 14]]
```

Stacking

- Combining 2 arrays vertically or horizontally
- use `vstack()` or `hstack()`
- Arrays must have compatible shapes

You can combine two or more arrays vertically or horizontally with the `vstack()` or `hstack()` functions. These functions are also handy for adding rows or columns with the results of operations.

Example

`np_stacking.py`

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76, 19, 5, 54, 66],
     [23, 29, 71, 12, 27, 74, 65, 73]]
) ①

b = np.array(
    [[11, 84, 7, 10, 31, 50, 11, 98],
     [25, 13, 43, 1, 31, 52, 41, 90]]
) ②

print('a =>\n', a)
print()
print('b =>\n', b)
print()
print('vstack((a,b)) =>\n', np.vstack((a, b))) ③
print()

print('vstack((a,a[0] + a[1])) =>\n', np.vstack((a, a[0] + a[1]))) ④
print()

print('hstack((a,b)) =>\n', np.hstack((a, b))) ⑤
```

① sample array a

- ② sample array b
- ③ stack arrays vertically (like pancakes)
- ④ add a row with sums of first two rows
- ⑤ stack arrays horizontally (like books on a shelf)

np_stacking.py

```
a =>
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]]

b =>
[[11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,b)) =>
[[70 31 21 76 19  5 54 66]
 [23 29 71 12 27 74 65 73]
 [11 84  7 10 31 50 11 98]
 [25 13 43  1 31 52 41 90]]

vstack((a,a[0] + a[1])) =>
[[ 70  31  21  76  19   5  54  66]
 [ 23  29  71  12  27  74  65  73]
 [ 93  60  92  88  46  79 119 139]]

hstack((a,b)) =>
[[70 31 21 76 19  5 54 66 11 84  7 10 31 50 11 98]
 [23 29 71 12 27 74 65 73 25 13 43  1 31 52 41 90]]
```

Iterating

- Similar to normal Python
- Iterates through first dimension
- Use `array.flat` to iterate through all elements

Iterating through a NumPy array is similar to iterating through any Python list; iteration is across the first dimension. Slicing and indexing can be used.

To iterate across every element, use the `flat` property of the array.

Example

`np_iterating.py`

```
#!/usr/bin/env python
import numpy as np

a = np.array(
    [[70, 31, 21, 76],
     [23, 29, 71, 12]]
) ①

print('a =>\n', a)
print()

print("for row in a: =>")
for row in a: ②
    print("row:", row)
print()

print("for column in a.T:")
for column in a.T: ③
    print("column:", column)
print()

print("for elem in a.flat: =>")
for elem in a.flat: ④
    print("element:", elem)
```

- ① sample array
- ② iterate over rows
- ③ iterate over columns by transposing the array
- ④ iterate over all elements (row-major)

np_iterating.py

```
a =>
[[70 31 21 76]
 [23 29 71 12]]

for row in a: =>
row: [70 31 21 76]
row: [23 29 71 12]

for column in a.T:
column: [70 23]
column: [31 29]
column: [21 71]
column: [76 12]

for elem in a.flat: =>
element: 70
element: 31
element: 21
element: 76
element: 23
element: 29
element: 71
element: 12
```


Array creation shortcuts

- Use "magic" arrays `r_`, `c_`
- Either creates a new numpy array
- Index values determine resulting array
- List of arrays creates a "stacked" array
- List of values creates a 1D array
- Slice notation creates a range of values
- A complex step creates equally-spaced value

Numpy provides several shortcuts for working with arrays.

The `r_` object can be used to magically build arrays via its index expression. It acts like a magic array, and "returns" (evaluates to) a normal numpy ndarray object.

There are two main ways to use `r_()`:

If the index expression contains a list of arrays, then the arrays are "stacked" along the first axis.

If the index contains slice notation, then it creates a one-dimensional array, similar to `numpy.arange()`. It uses start, stop, and step values. However, if step is an imaginary number (a literal that ends with `'j'`), then it specifies the number of points wanted, more like `numpy.linspace()`.

There can be more than one slice, as well as individual values, and ranges. They will all be concatenated into one array.

The first element in the index can be either a string that modifies how the array is created, or a string that makes the result a matrix instead of an ndarray.

If the first element is a string containing one, two, or three integers separated by commas, then the first integer is the axis to stack the arrays along; the second is the minimum number of dimensions to force each entry into; the third allows you to control the shape of the resulting array.

If the first element in the index is `"r"` or `"c"`, then a numpy matrix object is returned. `c_` works exactly like `r_`, but is column-oriented rather than row-oriented.

Example

np_tricks.py

```
#!/usr/bin/env python
import numpy as np

a1 = np.r_[np.array([1, 2, 3]), 0, 0, np.array([4, 5, 6])] ①
print(a1)
print()

a2 = np.r_-1:1:6j, [0] * 3, 5, 6 ②
print(a2)
print()

a = np.array([[0, 1, 2], [3, 4, 5]]) ③
a3 = np.r_['-1', a, a] ④
print(a3)
print()

a4 = np.r_['0,2', [1, 2, 3], [4, 5, 6]] ④
print(a4)
print()

a5 = np.r_['0,2,0', [1, 2, 3], [4, 5, 6]] ⑤
print(a5)
print()

a6 = np.r_['1,2,0', [1, 2, 3], [4, 5, 6]] ⑥
print(a6)
print()

m = np.r_['r', [1, 2, 3], [4, 5, 6]]
print(m)
print(type(m))
```

- ① build array from a sequence of array-like things
- ② faux slice with complex step implements linspace <3>

np_tricks.py

```
[1 2 3 0 0 4 5 6]

[-1.  -0.6 -0.2  0.2  0.6  1.   0.   0.   0.   5.   6. ]

[[0 1 2 0 1 2]
 [3 4 5 3 4 5]]

[[1 2 3]
 [4 5 6]]

[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]

[[1 4]
 [2 5]
 [3 6]]

[[1 2 3 4 5 6]]
<class 'numpy.matrixlib.defmatrix.matrix'>
```

Matrices

- Use normal ndarrays
- Most operations same as ndarray
- Use @ for multiplication

For traditional matrix operations, use a normal ndarray. Most operations are the same as for ndarrays. For matrix (diagonal) multiplication, use the @ (matrix multiplication) operator.

For transposing, use `array.transpose()`.

NOTE

There was formerly a `Matrix` type in NumPy, but it is deprecated since the addition of the @ operator in Python 3.5

Example

np_matrices.py

```
#!/usr/bin/env python
import numpy as np

m1 = np.array(
    [[2, 4, 6],
     [10, 20, 30]]
) ①

m2 = np.array([[1, 15],
               [3, 25],
               [5, 35]]) ②

print('m1 =>\n', m1)
print()

print('m2 =>\n', m2)
print()

print('m1 * 10 =>\n', m1 * 10) ③
print()

print('m1 @ m2 =>\n', m1 @ m2) ④
print()
```

- ① sample 2x3 array
- ② sample 3x2 array
- ③ multiply every element of m1 times 10
- ④ matrix multiply m1 times m2 — diagonal product

np_matrices.py

```
m1 =>
[[ 2  4  6]
 [10 20 30]]

m2 =>
[[ 1 15]
 [ 3 25]
 [ 5 35]]

m1 * 10 =>
[[ 20  40  60]
 [100 200 300]]

m1 @ m2 =>
[[ 44 340]
 [220 1700]]
```

Data Types

- Data type is inferred from initialization data
- Can be specified with `arange()`, `ones()`, `zeros()`, etc.

Numpy defines many different numeric data types. Integers can have different sizes and byte orders, and be either signed or unsigned. The data type is normally inferred from the initialization data. When using `arange()`, `ones()`, etc., to create arrays, the `dtype` parameter can be used to specify the data type.

Example

np_data_types.py

```
#!/usr/bin/env python
import numpy as np

r1 = np.arange(45) ①
r1.shape = (3, 3, 5) ②
print('r1 datatype:', r1.dtype)
print('r1 =>\n', r1, '\n')

r2 = np.arange(45.) ③
r2.shape = (3, 3, 5)
print('r2 datatype:', r2.dtype)
print('r2 =>\n', r2, '\n')

r3 = np.arange(45, dtype=np.int16) ③
r3.shape = (3, 3, 5)
print('r3 datatype:', r3.dtype)
print('r3 =>\n', r3, '\n')
```

- ① create array — `arange()` defaults to int
- ② create array — passing float makes all elements float
- ③ create array — set datatype to short int

np_data_types.py

```
r1 datatype: int64
r1 =>
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]

r2 datatype: float64
r2 =>
[[[ 0.  1.  2.  3.  4.]
  [ 5.  6.  7.  8.  9.]
  [10. 11. 12. 13. 14.]]

 [[15. 16. 17. 18. 19.]
  [20. 21. 22. 23. 24.]
  [25. 26. 27. 28. 29.]]

 [[30. 31. 32. 33. 34.]
  [35. 36. 37. 38. 39.]
  [40. 41. 42. 43. 44.]]]

r3 datatype: int16
r3 =>
[[[ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]]

 [[15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]]

 [[30 31 32 33 34]
  [35 36 37 38 39]
  [40 41 42 43 44]]]
```

Numpy Example List with Doc

- Over 200 functions
- Examples with doc strings

Numpy has over 200 functions. To get more information, check out the NumPy Example List with Doc at http://wiki.scipy.org/Numpy_Example_List_With_Doc. This page has all 217 (as of Spring 2014) of the Numpy functions and operators, with examples, and interleaved with their docstrings. It is a great resource for getting up to speed with Numpy.

Chapter 16 Exercises

Exercise 16-1 (`big_arrays.py`)

Starting with the file `big_arrays.py`, convert the Python list values into a numpy array.

Make a copy of the array named `values_x_3` with all values multiplied by 3.

Print out `values_x_3`

Exercise 16-2 (`create_range.py`)

Using `arange()`, create an array of 35 elements.

Reshape the array to be 5 x 7 and print it out.

Reshape the array to be 7 x 5 and print it out.

Exercise 16-3 (`create_linear_space.py`)

Using `linspace()`, create an array of 500 elements evenly spaced between 100 and 200.

Reshape the array into 5 x 10 x 10.

Multiply every element by .5

Print the result.

Chapter 17: Introduction to SciPy

Objectives

- Understand the motivation for scipy
- See what the scipy module provides
- Import scipy and friends using standard abbreviations
- Learn which functions are aliased from numpy
- Use the scipy documentation commands
- Tour scipy's many subpackages

About scipy

- Part of the "Python Scientific Stack"
- Often used with matplotlib
- Many mathematical and statistical algorithms
- Includes numpy "under the hood"
- Can be used with iPython
- Very large collection of routines and subpackages

scipy is a collection of modules and submodules for doing scientific (mostly numerical) analysis.

The scipy module itself acts as an umbrella module, or repository, for many useful submodules.

Although it can be used alone, numpy is part of scipy, and many useful numpy functions are aliased into the scipy namespace.

In addition, many common functions from scipy's dozens of submodules have been aliased to the scipy namespace.

All of the top-level **numpy** functions are available through scipy as well.

Polynomials

- Create a `poly1d` object
- Represented in either of two ways
 - list of coefficients (1st element is coefficient of highest power)
 - list of roots
- Call polynomial with value to solve for
- `r` attribute represents list of roots

Polynomials can be represented in `scipy` in two ways using `numpy`'s `poly1d()` method, which takes a list of coefficients; the other is to just provide a list of coefficients, where the first element is the coefficient of the highest power.

The `poly1d()` method takes a list of coefficients (or roots) and returns a `poly1d` object.

Treating the polynomial object like a string returns a text representation of the polynomial.

By default, pass an iterable of integers to `poly1d` which represent the coefficients. To specify roots, pass a second parameter to `poly1d` with a `true` value.

The variable used when displaying the polynomial is normally `x`. To use a different variable, add a third parameter with a string.

To solve for a specific value, call the polynomial with that value. The `r` property of the polynomial contains the roots.

You can use the addition, subtraction, division, multiplication, and exponentiation operators between polynomials and scalar values.

NOTE | `poly1d` is automatically imported to `scipy`'s namespace as well

Example

`sp_polynomials.py`

```
#!/usr/bin/env python
import scipy as sp

p1 = sp.poly1d([2, 1, 4]) ①
```

```
print(p1)
print()

print(p1(.75)) ②

print(p1.r) ③
print()

p2 = sp.poly1d([2, 1, -4], True) ④
print(p2)
print()

print(p2(.75)) ⑤
print(p2.r) ⑥
print()

p3 = sp.poly1d([1, 2, 3], False, 'm') ⑦
print(p3)
print()

print(p3(100)) ⑧

print(p3.r) ⑨
print()

p4 = sp.poly1d([1, 2]) ⑩
p5 = sp.poly1d([3, 4])
print()

print(p4)
print()

print(p5)
print()

print(p4 + p5)
print()

print(p4 - p5)
print()

print(p4 ** 3)
print()
```


- ① 2,1,4 are coefficients
- ② evaluate for $x = .75$
- ③ get the roots
- ④ 2,1,-4 are roots
- ⑤ evaluate for $x = .75$
- ⑥ get the roots
- ⑦ 1,2,3 are coefficients, variable is 'm'
- ⑧ evaluate for $m = 100$
- ⑨ get the roots
- ⑩ polynomial arithmetic

sp_polynomials.py

$$2x^2 + 1x + 4$$

5.875
 $[-0.25+1.39194109j \ -0.25-1.39194109j]$

$$1x^3 + 1x^2 - 10x + 8$$

1.484375
 $[-4. \ 2. \ 1.]$

$$1m^2 + 2m + 3$$

10203
 $[-1.+1.41421356j \ -1.-1.41421356j]$

$$1x + 2$$

$$3x + 4$$

$$4x + 6$$

$$-2x - 2$$

$$1x^3 + 6x^2 + 12x + 8$$

Vectorizing functions

- Many functions "just work"
- `sp.vectorize()` allows function to be broadcast.

As with numpy, most scipy functions will automatically be broadcast across any array to which they are applied. For user-defined functions that don't correctly broadcast, scipy provides the `vectorize` function. It takes a function which accepts one or more scalar values (float, integers, etc.) and returns a single scalar value.

Example

sp_vectorize.py

```
#!/usr/bin/env python
import numpy as np
import scipy as sp

def set_default(value, limit, default): ①
    if value > limit:
        value = default

    return value

raw_samples = np.array([5, 18, 36, 1000, 98, 2323]) ②

try:
    print("Without sp.vectorize:")
    norm_samples = set_default(raw_samples, 100, 0) ③
except ValueError as err:
    print("Function failed:", err) ④
else:
    print(norm_samples)
finally:
    print()

set_default_vect = sp.vectorize(set_default) ⑤
try:
    print("With sp.vectorize:")
    norm_samples = set_default_vect(raw_samples, 100, 0) ⑥
except ValueError as err:
    print("Function failed:", err)
else:
    print(norm_samples) ⑦
finally:
    print()
```

- ① define function with more than one parameter
- ② create some sample data
- ③ pass np array to function
- ④ it will fail because it's expecting a function with only one parameter
- ⑤ convert function to vectorized version — creates function that takes one parameter and has the other two "embedded" in it
- ⑥ now call vectorized version with same parameters
- ⑦ success

sp_vectorize.py

Without `sp.vectorize`:

Function failed: The truth value of an array with more than one element is ambiguous. Use `a.any()` or `a.all()`

With `sp.vectorize`:

```
[ 5 18 36  0 98  0]
```

Working with SciPy

- Some functions imported to scipy
- Hundreds more in subpackages
- Most functions have similar interfaces
- Use numpy plus scipy.subpackage routines as needed

scipy's subpackages have hundreds of functions. For convenience, some of them have been imported into the scipy namespace. While numpy is mostly about arrays and matrices, there are some useful data handling functions as well.

Example

sp_functions.py

```
#!/usr/bin/env python
import sys
import numpy as np

if sys.platform == 'darwin': ①
    import matplotlib

    matplotlib.use('QT5Agg')
import matplotlib.pyplot as plt

dt = np.dtype([('Month', 'int8'), ('Day', 'int8'), ('Year', 'int16'), ('Temp',
'float64')]) ②
data = np.loadtxt('../DATA/weather/NYNEWYOR.txt', dtype=dt) ③

print(data['Temp'])

temps = data['Temp'] ④

plt.plot(temps) ⑤
plt.show() ⑥
plt.cla() ⑦

normalized_data = data[data['Temp'] > -50] ⑧
temps = normalized_data['Temp'] ⑨
plt.plot(temps) ⑩
plt.show()
```

- ① fix Mac plotting issue
- ② define custom numpy datatype
- ③ read flat-file data into numpy array
- ④ get data from 'Temp' column
- ⑤ plot days against temps
- ⑥ display plot
- ⑦ clear first axis but not the whole figure
- ⑧ remove readings < -50, which seem to be default N/A values
- ⑨ grab temps again
- ⑩ replot

NOTE[see plot](#)

SciPy Subpackages

- cluster — Clustering algorithms
- constants — Physical and mathematical constants
- fftpack --Fast Fourier Transform routines
- integrate — Integration and ordinary differential equation solvers
- interpolate — Interpolation and smoothing splines
- io — Input and Output
- linalg — Linear algebra
- ndimage — N-dimensional image processing
- odr — Orthogonal distance regression
- optimize — Optimization and root-finding routines
- signal — Signal processing
- sparse — Sparse matrices and associated routines
- spatial — Spatial data structures and algorithms
- special — Special functions
- stats — Statistical distributions and functions
- weave — C/C++ integration

Getting Help

- `sp.info(topic)`
- `sp.source(module-or-method)`

In addition to the normal Python and iPython help systems, scipy provides its own `info()` method, which displays extended help on any scipy method. The `source()` method displays the source code of the specified module or method.

Python and C/C++

- Ctypes allows loading of shared libraries (dll/dylib/so)
- Cython is optimizing static compiler

When there is not a fast numpy/scipy routine that implements a needed algorithm, you can write the algorithm in C/C++.

This may not help with code that already uses existing numpy/scipy functions, but if the code contains nested loops, the speedup can be significant.

Of course, extending scipy with C requires a C development package. This can be either Microsoft Visual C++ or MinGW on Windows, and is typically gcc on other platforms.

The simplest approach is **ctypes**. This is part of the standard library, and allows direct import C/C shared libraries (DLLs) without writing any custom C/C code

cython is a version of Python that automatically pre-compiles selected Python code into C. It looks like Python, but has type declarations similar to those in C.

Tour of SciPy subpackages

- 20 subpackages
- Thousands of routines

SciPy has 20 main subpackages, each of which can have hundreds of routines. They are all designed to work with NumPy arrays, and there is some overlap among them. Some of the routines are also available from the **scipy** package itself, without having to import the subpackage.

We will now visit the SciPy documentation for a brief tour of the SciPy packages, at <https://docs.scipy.org/doc/scipy/reference/>

SciPy in action

- Typical to use Jupyter notebook
- Import subpackages as needed

To see SciPy in action, please view the Jupyter notebook named **scipy_demo.ipynb**.

Chapter 18: Introduction to Pandas

Objectives

- Understand what the pandas module provides
- Load data from CSV and other files
- Access data tables
- Extract rows and columns using conditions
- Calculate statistics for rows or columns

About pandas

- Reads data from file, database, or other sources
- Deals with real-life issues such as invalid data
- Powerful selecting and indexing tools
- Builtin statistical functions
- Munge, clean, analyze, and model data
- Works with numpy and matplotlib

pandas is a package designed to make it easy to get, organize, and analyze large datasets. Its strengths lie in its ability to read from many different data sources, and to deal with real-life issues, such as missing, incomplete, or invalid data.

pandas also contains functions for calculating means, sums and other kinds of analysis.

For selecting desired data, pandas has many ways to select and filter rows and columns.

It is easy to integrate pandas with numpy, matplotlib, and other scientific packages.

While pandas can handle one, two, three, or higher dimensional data, it is generally used with two-dimensional (row/column) data, which can be visualized like a spreadsheet.

pandas provides powerful Split-Apply-Combine operations—groupby enables transformations, aggregations, and easy-access plotting functions. It is easy to emulate R's plyr package via pandas.

NOTE | [pandas gets its name from *panel data* system](#)

Pandas architecture

- `pandas.core.frame`
- Two main structures: Series and DataFrame
- Series – one-dimensional
- DataFrame – two-dimensional

The two main data structures in pandas are the Series and the DataFrame. A series is a one-dimensional indexed list of values, something like an ordered dictionary. A DataFrame is a two-dimensional grid, with both row and column indices (like the rows and columns of a spreadsheet, but more flexible).

You can specify the indices, or pandas will use successive integers. Each row or column of a DataFrame is a Series.

The third structure in pandas is a Panel, which is more or less a collection of DataFrames, and describes three-dimensional (or higher) data.

Series

- Indexed list of values
- Similar to a dictionary, but ordered
- Can get `sum()`, `mean()`, etc.
- Use index to get individual values
- Indices are not positional

A Series is an indexed sequence of values. Each item in the sequence has an index. The default index is a set of increasing integer values, but any set of values can be used.

For example, you can create a series with the values 5, 10, and 15 as follows:

```
s1 = pd.Series([5,10,15])
```

This will create a Series indexed by [0, 1, 2]. To provide index values, add a second list:

```
s2 = pd.Series([5,10,15], ['a','b','c'])
```

This specifies the indices as 'a', 'b', and 'c'.

You can also create a Series from a dictionary. pandas will put the index values in order:

```
s3 = pd.Series({'b':10, 'a':5, 'c':15})
```

There are many methods that can be called on a Series, and Series can be indexed in many flexible ways.

Example

pandas_series.py

```
#!/usr/bin/env python
import numpy as np
import pandas as pd

NUM_VALUES = 10
index = [chr(i) for i in range(97, 97 + NUM_VALUES)] ①
print("index:", index, '\n')

s1 = pd.Series(np.linspace(1, 5, NUM_VALUES), index=index) ②
s2 = pd.Series(np.linspace(1, 5, NUM_VALUES)) ③

print("s1:", s1, "\n")
print("s2:", s2, "\n")

print("selecting elements")
print(s1[['h', 'b']], "\n") ④

print(s1[['a', 'b', 'c']], "\n") ④

print("slice of elements")
print(s1['b':'d'], "\n") ⑤

print("sum(), mean(), min(), max():")
print(s1.sum(), s1.mean(), s1.min(), s1.max(), "\n") ⑥

print("cumsum(), cumprod():")
print(s1.cumsum(), s1.cumprod(), "\n") ⑥

print('a' in s1) ⑦
print('m' in s1) ⑦
print()

s3 = s1 * 10 ⑧
print("s3 (which is s1 * 10)")
print(s3, "\n")

s1['e'] *= 5

print("boolean mask where s3 > 25:")
print(s3 > 25, "\n") ⑨
```

```

print("assign -1 where mask is true")
s3[s3 < 25] = -1 ⑩
print(s3, "\n")

s4 = pd.Series([-0.204708, 0.478943, -0.519439]) ⑪
print("s4.max(), .min(), etc.")
print(s4.max(), s4.min(), s4.max() - s4.min(), '\n') ⑫

s = pd.Series([5, 10, 15], ['a', 'b', 'c']) ⑬
print("creating series with index")
print(s)

```

- ① make list of 'a', 'b', 'c', ...
- ② create series with specified index
- ③ create series with auto-generated index (0, 1, 2, 3, ...)
- ④ select items from series
- ⑤ select slice of elements
- ⑥ get stats on series
- ⑦ test for existence of label
- ⑧ create new series with every element of s1 multiplied by 10
- ⑨ create boolean mask from series
- ⑩ set element to -1 where mask is True
- ⑪ create new series
- ⑫ print stats
- ⑬ create new series with index

pandas_series.py

```
index: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

```
s1: a    1.000000
```

```
    b    1.444444
```

```
    c    1.888889
```

```
    d    2.333333
```

```
    e    2.777778
```

```
    f    3.222222
```

```
    g    3.666667
```

```
    h    4.111111
```

```
    i    4.555556
```

```
    j    5.000000
```

```
dtype: float64
```

```
s2: 0    1.000000
```

```
    1    1.444444
```

```
    2    1.888889
```

```
    3    2.333333
```

```
    4    2.777778
```

```
    5    3.222222
```

```
    6    3.666667
```

```
    7    4.111111
```

```
    8    4.555556
```

```
    9    5.000000
```

```
dtype: float64
```

```
selecting elements
```

```
h    4.111111
```

```
b    1.444444
```

```
dtype: float64
```

```
a    1.000000
```

```
b    1.444444
```

```
c    1.888889
```

```
dtype: float64
```

```
slice of elements
```

```
b    1.444444
```

```
c    1.888889
```

```
d    2.333333
```

```
dtype: float64
```

```
sum(), mean(), min(), max():
```

```
30.0 3.0 1.0 5.0
```

```
cumsum(), cumprod():
```

```
a    1.000000
```

```
b    2.444444
```

```
c    4.333333
```

```
d    6.666667
```

```
e    9.444444
```

```
f   12.666667
```

```
g   16.333333
```

```
h   20.444444
```

```
i   25.000000
```

```
j   30.000000
```

```
dtype: float64 a    1.000000
```

```
b    1.444444
```

```
c    2.728395
```

```
d    6.366255
```

```
e   17.684042
```

```
f   56.981913
```

```
g  208.933682
```

```
h  858.949582
```

```
i 3912.992542
```

```
j 19564.962709
```

```
dtype: float64
```

```
True
```

```
False
```

```
s3 (which is s1 * 10)
```

```
a   10.000000
```

```
b   14.444444
```

```
c   18.888889
```

```
d   23.333333
```

```
e   27.777778
```

```
f   32.222222
```

```
g   36.666667
```

```
h   41.111111
```

```
i   45.555556
```

```
j   50.000000
```

```
dtype: float64
```

```
boolean mask where s3 > 25:
```

```
a    False
```

```
b    False
c    False
d    False
e     True
f     True
g     True
h     True
i     True
j     True
dtype: bool
```

assign -1 where mask is true

```
a    -1.000000
b    -1.000000
c    -1.000000
d    -1.000000
e    27.777778
f    32.222222
g    36.666667
h    41.111111
i    45.555556
j    50.000000
dtype: float64
```

s4.max(), .min(), etc.

```
0.478943 -0.519439 0.998382
```

creating series with index

```
a      5
b     10
c     15
dtype: int64
```

DataFrames

- Two-dimensional grid of values
- Row and column labels (indices)
- Rich set of methods
- Powerful indexing

A DataFrame is the workhorse of Pandas. It represents a two-dimensional grid of values, containing indexed rows and columns, something like a spreadsheet.

There are many ways to create a DataFrame. They can be modified to add or remove rows/columns. Missing or invalid data can be eliminated or normalized.

DataFrames can be initialized from many kinds of data. See the table on the next page for a list of possibilities.

NOTE

The `panda DataFrame` is modeled after R's `data.frame`

Table 23. DataFrame Initializers

Initializer	Description
2D ndarray	A matrix of data, passing optional row and column labels
dict of arrays, lists, or tuples	Each sequence becomes a column in the DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the “dict of arrays” case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the result’s row index if no explicit index is passed.
dict of dicts	Each inner dict becomes a column. Keys are unioned to form the row index as in the “dict of Series” case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the DataFrame’s column labels
List of lists or tuples	Treated as the “2D ndarray” case
Another DataFrame	The DataFrame’s indexes are used unless different ones are passed
NumPy MaskedArray	Like the “2D ndarray” case except masked values become NA/missing in the DataFrame result

NOTE

This utility method is used in some of the example scripts:

printhead.py

```
#!/usr/bin/env python
HEADER_CHAR = '='

def print_header(comment, header_width=50):
    ''' Print comment and separator '''
    header_line = HEADER_CHAR * header_width
    print(header_line)
    print(comment.center(header_width-2).center(header_width, HEADER_CHAR))
    print(header_line)

if __name__ == '__main__':
    print_header("this is a test")
```


Example

pandas_simple_dataframe.py

```
#!/usr/bin/env python
import pandas as pd
from printhead import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon'] ①
indices = ['a', 'b', 'c', 'd', 'e', 'f'] ②

values = [ ③
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('cols')
print(cols, '\n')

print_header('indices')
print(indices, '\n')

print_header('values')
print(values, '\n')

df = pd.DataFrame(values, index=indices, columns=cols) ④
print_header('DataFrame df')
print(df, '\n')

print_header("df['gamma']")
print(df['gamma']) ⑤
```

- ① column names
- ② row names
- ③ sample data
- ④ create dataframe with row and column names
- ⑤ select column 'gamma'

pandas_simple_dataframe.py

```

=====
=                                cols                                =
=====
['alpha', 'beta', 'gamma', 'delta', 'epsilon']

=====
=                                indices                             =
=====
['a', 'b', 'c', 'd', 'e', 'f']

=====
=                                values                             =
=====
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340],
[400, 410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]

=====
=                                DataFrame df                        =
=====
   alpha  beta  gamma  delta  epsilon
a    100   110   120   130    140
b    200   210   220   230    240
c    300   310   320   330    340
d    400   410   420   430    440
e    500   510   520   530    540
f    600   610   620   630    640

=====
=                                df['gamma']                        =
=====
a    120
b    220
c    320
d    420
e    520
f    620
Name: gamma, dtype: int64

```

Data alignment

- pandas will auto-align data by rows and columns
- Non-overlapping data will be set as NaN

When two dataframes are combined, columns and indices are aligned.

The result is the union of matching rows and columns. Where data doesn't exist in one or the other dataframe, it is set to NaN.

A default value can be specified for the overlapping cells when combining dataframes with methods such as `add()` or `mul()`.

Use the `fill_value` parameter to set a default for missing values.

Example

pandas_alignment.py

```
#!/usr/bin/env python
import numpy as np
import pandas as pd
from printhead import print_header ①

dataset1 = np.arange(9.).reshape((3, 3)) ②

df1 = pd.DataFrame( ③
    dataset1,
    columns=['apple', 'banana', 'mango'],
    index=['orange', 'purple', 'blue']
)

dataset2 = np.arange(12.).reshape((4, 3)) ②

df2 = pd.DataFrame( ③
    dataset2,
    columns=['apple', 'banana', 'kiwi'],
    index=['orange', 'purple', 'blue', 'brown']
)

print_header('df1')
print(df1) ④
print()

print_header('df2')
print(df2) ④
print()

print_header('df1 + df2')
print(df1 + df2) ⑤

print_header('df1.add(df2, fill_value=0)')
print(df1.add(df2, fill_value=0)) ⑥
```

- ① provided with lab files to make output easier to read
- ② create a numpy array
- ③ create second Pandas dataframe from dataset, adding row and column numbers; note rows and columns do not quite match
- ④ output dataframe
- ⑤ when dataframes are combined, if no match for row + column label, set value to NaN
- ⑥ same as #5, but where one dataframe has a value, set it to 0

pandas_alignment.py

```
=====
=                                df1                                =
=====
      apple  banana  mango
orange    0.0     1.0    2.0
purple    3.0     4.0    5.0
blue      6.0     7.0    8.0

=====
=                                df2                                =
=====
      apple  banana  kiwi
orange    0.0     1.0    2.0
purple    3.0     4.0    5.0
blue      6.0     7.0    8.0
brown     9.0    10.0   11.0

=====
=                                df1 + df2                            =
=====
      apple  banana  kiwi  mango
blue     12.0    14.0   NaN   NaN
brown     NaN     NaN   NaN   NaN
orange     0.0     2.0   NaN   NaN
purple     6.0     8.0   NaN   NaN

=====
=                                df1.add(df2, fill_value=0)            =
=====
      apple  banana  kiwi  mango
blue     12.0    14.0    8.0    8.0
brown     9.0    10.0   11.0   NaN
orange     0.0     2.0    2.0    2.0
purple     6.0     8.0    5.0    5.0
```

Index objects

- Used to index Series or DataFrames
- `index = pandas.core.frame.Index(sequence)`
- Can be named

An index object is a kind of ordered set that is used to access rows or columns in a dataset. As shown earlier, indexes can be specified as lists or other sequences when creating a Series or DataFrame.

You can create an index object and then create a Series or a DataFrame using the index object. Index objects can be named, either something obvious like 'rows' or 'columns', or more appropriate to the specific type of data being indexed.

Remember that index objects act like sets, so the main operations on them are unions, intersections, or differences.

TIP You can replace an existing index on a DataFrame with the `set_index()` method.

Example

`pandas_index_objects.py`

```
#!/usr/bin/env python
import pandas as pd
from printhead import print_header

index1 = pd.Index(['a', 'b', 'c'], name='letters') ①
index2 = pd.Index(['b', 'a', 'c'])
index3 = pd.Index(['b', 'c', 'd'])
index4 = pd.Index(['red', 'blue', 'green'], name='colors')

print_header("index1, index2, index3", 70) ②
print(index1)
print(index2)
print(index3)
print()

print_header("index2 & index3", 70)
# these are the same
```

```

print(index2 & index3) ③
print(index2.intersection(index3)) ③
print()

print_header("index2 | index3", 70)
# these are the same
print(index2 | index3) ④
print(index2.union(index3))
print()

print_header("index1.difference(index3)", 70)
print(index1.difference(index3)) ⑤
print()

print_header("Series([10,20,30], index=index1)", 70)
series1 = pd.Series([10, 20, 30], index=index1) ⑥
print(series1)
print()

print_header("DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1, columns=index4)",
70)
dataframe1 = pd.DataFrame([(1, 2, 3), (4, 5, 6), (7, 8, 9)], index=index1,
columns=index4)
print(dataframe1)
print()

print_header("DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4, columns=index1)",
70)
dataframe2 = pd.DataFrame([(1, 2, 3), (4, 5, 6), (7, 8, 9)], index=index4,
columns=index1)
print(dataframe2)
print()

```

- ① create some indexes
- ② display indexes
- ③ get intersection of indexes
- ④ get union of indexes
- ⑤ get difference of indexes
- ⑥ use index with series (can also be used with dataframe)

pandas_index_objects.py

```

=====
=                                index1, index2, index3                                =
=====
Index(['a', 'b', 'c'], dtype='object', name='letters')
Index(['b', 'a', 'c'], dtype='object')
Index(['b', 'c', 'd'], dtype='object')

=====
=                                index2 & index3                                =
=====
Index(['b', 'c'], dtype='object')
Index(['b', 'c'], dtype='object')

=====
=                                index2 | index3                                =
=====
Index(['a', 'b', 'c', 'd'], dtype='object')
Index(['a', 'b', 'c', 'd'], dtype='object')

=====
=                                index1.difference(index3)                        =
=====
Index(['a'], dtype='object')

=====
=                                Series([10,20,30], index=index1)                  =
=====
letters
a    10
b    20
c    30
dtype: int64

=====
= DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index1, columns=index4) =
=====
colors  red  blue  green
letters
a        1    2    3
b        4    5    6
c        7    8    9

```

```
=====
= DataFrame([(1,2,3),(4,5,6),(7,8,9)], index=index4, columns=index1) =
=====
letters  a  b  c
colors
red      1  2  3
blue     4  5  6
green    7  8  9
```

Basic Indexing

- Similar to normal Python or numpy
- Slices select rows

One of the real strengths of pandas is the ability to easily select desired rows and columns. This can be done with simple subscripting, like normal Python, or extended subscripting, similar to numpy. In addition, pandas has special methods and attributes for selecting data.

For selecting columns, use the column name as the subscript value. This selects the entire column. To select multiple columns, use a sequence (list, tuple, etc.) of column names.

For selecting rows, use slice notation. This may not map to similar tasks in normal python. That is, `dataframe[x:y]` selects rows x through y, but `dataframe[x]` selects column x.

Example

pandas_selecting.py

```
#!/usr/bin/env python
import pandas as pd
from printhead import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon'] ①
index = ['a', 'b', 'c', 'd', 'e', 'f'] ②

values = [ ③
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]

df = pd.DataFrame(values, index=index, columns=cols) ④
print_header('DataFrame df')
print(df, '\n')

print_header("df['alpha']")
print(df['alpha'], '\n') ⑤

print_header("df.beta")
print(df.beta, '\n') ⑥

print_header("df['b':'e']")
print(df['b':'e'], '\n') ⑦

print_header("df[['alpha','epsilon','beta']]")
print(df[['alpha', 'epsilon', 'beta']]) ⑧
print()

print_header("df[['alpha','epsilon','beta']]['b':'e']")
print(df[['alpha', 'epsilon', 'beta']]['b':'e']) ⑨
print()
```

- ① column labels
- ② row labels
- ③ sample data
- ④ create dataframe with data, row labels, and column labels
- ⑤ select column 'alpha'
- ⑥ same, but alternate syntax (only works if column name is letters, digits, and underscores)
- ⑦ select rows 'b' through 'e' using slice of row labels
- ⑧ select columns — note index is an iterable
- ⑨ select columns AND slice rows

pandas_selecting.py

```
=====
=                      DataFrame df                      =
=====
   alpha  beta  gamma  delta  epsilon
a    100   110   120   130     140
b    200   210   220   230     240
c    300   310   320   330     340
d    400   410   420   430     440
e    500   510   520   530     540
f    600   610   620   630     640

=====
=                      df['alpha']                      =
=====
a     100
b     200
c     300
d     400
e     500
f     600
Name: alpha, dtype: int64

=====
=                      df.beta                          =
=====
a     110
b     210
c     310
```

```

d    410
e    510
f    610
Name: beta, dtype: int64

```

```

=====
=                df['b':'e']                =
=====

```

	alpha	beta	gamma	delta	epsilon
b	200	210	220	230	240
c	300	310	320	330	340
d	400	410	420	430	440
e	500	510	520	530	540

```

=====
=      df[['alpha','epsilon','beta']]      =
=====

```

	alpha	epsilon	beta
a	100	140	110
b	200	240	210
c	300	340	310
d	400	440	410
e	500	540	510
f	600	640	610

```

=====
=      df[['alpha','epsilon','beta']]['b':'e']      =
=====

```

	alpha	epsilon	beta
b	200	240	210
c	300	340	310
d	400	440	410
e	500	540	510

Broadcasting

- Operation is applied across rows and columns
- Can be restricted to selected rows/columns
- Sometimes called vectorization
- Use `apply()` for more complex operations

If you multiply a dataframe by some number, the operation is broadcast, or vectorized, across all values. This is true for all basic math operations.

The operation can be restricted to selected columns.

For more complex operations, the `apply()` method will apply a function that selects elements. You can use the name of an existing function, or supply a lambda (anonymous) function.

Example

pandas_broadcasting.py

```
#!/usr/bin/env python
import pandas as pd
from printhead import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon'] ①
index = pd.date_range('2013-01-01 00:00:00', periods=6, freq='D') ②

print(index, "\n")

values = [ ③
    [100, 110, 120, 930, 140],
    [250, 210, 120, 130, 840],
    [300, 310, 520, 430, 340],
    [275, 410, 420, 330, 777],
    [300, 510, 120, 730, 540],
    [150, 610, 320, 690, 640],
]

df = pd.DataFrame(values, index, cols) ④
print_header("Basic DataFrame:")
print(df)
print()

print_header("Triple each value")
print(df * 3)
print() ⑤

print_header("Multiply column gamma by 1.5")
df['gamma'] *= 1.5 ⑥
print(df)
print()
```


- ① column labels
- ② date range to be used as row indexes
- ③ sample data
- ④ create dataframe from data
- ⑤ multiply every value by 3
- ⑥ multiply values in column 'gamma' by 1.

pandas_broadcasting.py

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
=====
=                Basic DataFrame:                =
=====
```

	alpha	beta	gamma	delta	epsilon
2013-01-01	100	110	120	930	140
2013-01-02	250	210	120	130	840
2013-01-03	300	310	520	430	340
2013-01-04	275	410	420	330	777
2013-01-05	300	510	120	730	540
2013-01-06	150	610	320	690	640

```
=====
=                Triple each value                =
=====
```

	alpha	beta	gamma	delta	epsilon
2013-01-01	300	330	360	2790	420
2013-01-02	750	630	360	390	2520
2013-01-03	900	930	1560	1290	1020
2013-01-04	825	1230	1260	990	2331
2013-01-05	900	1530	360	2190	1620
2013-01-06	450	1830	960	2070	1920

```
=====
=                Multiply column gamma by 1.5                =
=====
```

	alpha	beta	gamma	delta	epsilon
2013-01-01	100	110	180.0	930	140
2013-01-02	250	210	180.0	130	840
2013-01-03	300	310	780.0	430	340
2013-01-04	275	410	630.0	330	777
2013-01-05	300	510	180.0	730	540
2013-01-06	150	610	480.0	690	640

Removing entries

- Remove rows or columns
- Use drop() method

To remove columns or rows, use the drop() method, with the appropriate labels. Use axis=1 to drop columns, or axis=0 to drop rows.

Example

pandas_drop.py

```
#!/usr/bin/env python
import pandas as pd
from printhead import print_header

cols = ['alpha', 'beta', 'gamma', 'delta', 'epsilon']
index = ['a', 'b', 'c', 'd', 'e', 'f']
values = [
    [100, 110, 120, 130, 140],
    [200, 210, 220, 230, 240],
    [300, 310, 320, 330, 340],
    [400, 410, 420, 430, 440],
    [500, 510, 520, 530, 540],
    [600, 610, 620, 630, 640],
]
print_header('values:')
print(values, '\n\n')

df = pd.DataFrame(values, index=index, columns=cols) ①
print_header('DataFrame df')
print(df, '\n')

df2 = df.drop(['beta', 'delta'], axis=1) ②
print_header("After dropping beta and delta:")
print(df2, '\n')

print_header("After dropping rows b, c, and e")
df3 = df.drop(['b', 'c', 'e']) ③
print(df3)
```

- ① create dataframe
- ② drop columns beta and delta (axes: 0=rows, 1=columns)
- ③ drop rows b, c, and e

pandas_drop.py

```
=====
=                               values:                               =
=====
[[100, 110, 120, 130, 140], [200, 210, 220, 230, 240], [300, 310, 320, 330, 340],
 [400, 410, 420, 430, 440], [500, 510, 520, 530, 540], [600, 610, 620, 630, 640]]

=====
=                               DataFrame df                               =
=====
   alpha  beta  gamma  delta  epsilon
a    100   110   120   130    140
b    200   210   220   230    240
c    300   310   320   330    340
d    400   410   420   430    440
e    500   510   520   530    540
f    600   610   620   630    640

=====
=       After dropping beta and delta:       =
=====
   alpha  gamma  epsilon
a    100   120    140
b    200   220    240
c    300   320    340
d    400   420    440
e    500   520    540
f    600   620    640

=====
=       After dropping rows b, c, and e       =
=====
   alpha  beta  gamma  delta  epsilon
a    100   110   120   130    140
d    400   410   420   430    440
f    600   610   620   630    640
```

Time Series

- Use `time_series()`
- Specify start/end time/date, number of periods, time units
- Useful as index to other data
- `freq=time_unit`
- `periods=number_of_periods`

pandas provides a function `time_series()` to generate a list of timestamps. You can specify the start/end times as dates or dates/times, and the type of time units. Alternatively, you can specify a start date/time and the number of periods to create.

The frequency strings can have multiples – 5H means every 5 hours, 3S means every 3 seconds, etc.

Table 24. Units for `time_series()` `freq` flag

Unit	Represents
M	Month
D	Day
H	Hour
T	Minute
S	Second

Example

pandas_time_slices.py

```
#!/usr/bin/env python
import pandas as pd
import numpy as np

hourly = pd.date_range('1/1/2013 00:00:00', '1/3/2013 23:59:59', freq='H') ①
print("Number of periods: ", len(hourly))

seconds = pd.date_range('1/1/2013 12:00:00', freq='S', periods=(60 * 60 * 18)) ②
print("Number of periods: ", len(seconds))
print("Last second: ", seconds[-1])

monthly = pd.date_range('1/1/2013', '12/31/2013', freq='M') ③
print("Number of periods: {0} Seventh element: {1}".format(
    len(monthly),
    monthly[6]
))

NUM_DATA_POINTS = 1441 ④

dates = pd.date_range('4/1/2013 00:00:00', periods=NUM_DATA_POINTS, freq='T') ⑤

data = np.random.random(NUM_DATA_POINTS) ⑥

series = pd.Series(data, index=dates) ⑦

time_slice = series['4/1/2013 10:00:00':'4/1/2013 10:30:00'] ⑧
print(time_slice) # 31 values
```

- ① make time series — every hour for 3 days
- ② make time series — every second for 18 hours
- ③ every month for 1 year
- ④ number of minutes in a day
- ⑤ create range from starting point with specified number of points—one day's worth of minutes
- ⑥ a day's worth of data
- ⑦ series indexed by minutes
- ⑧ select the half hour of data from 10:00 to 10:30

pandas_time_slices.py

```
Number of periods: 72
Number of periods: 64800
Last second: 2013-01-02 05:59:59
Number of periods: 12 Seventh element: 2013-07-31 00:00:00
2013-04-01 10:00:00    0.492323
2013-04-01 10:01:00    0.527705
2013-04-01 10:02:00    0.647620
2013-04-01 10:03:00    0.825908
2013-04-01 10:04:00    0.959560
2013-04-01 10:05:00    0.913720
2013-04-01 10:06:00    0.375076
2013-04-01 10:07:00    0.675413
2013-04-01 10:08:00    0.130059
2013-04-01 10:09:00    0.415462
2013-04-01 10:10:00    0.113834
2013-04-01 10:11:00    0.525120
2013-04-01 10:12:00    0.805297
2013-04-01 10:13:00    0.854677
2013-04-01 10:14:00    0.605145
2013-04-01 10:15:00    0.996919
2013-04-01 10:16:00    0.044626
2013-04-01 10:17:00    0.422211
2013-04-01 10:18:00    0.256331
2013-04-01 10:19:00    0.654759
2013-04-01 10:20:00    0.893795
2013-04-01 10:21:00    0.176522
2013-04-01 10:22:00    0.364400
2013-04-01 10:23:00    0.825236
2013-04-01 10:24:00    0.820759
2013-04-01 10:25:00    0.528879
2013-04-01 10:26:00    0.013953
2013-04-01 10:27:00    0.878799
2013-04-01 10:28:00    0.497146
2013-04-01 10:29:00    0.046714
2013-04-01 10:30:00    0.941946
Freq: T, dtype: float64
```


Table 25. Methods and attributes for fetching data

Method	Description
DF.columns	Get or set column labels
DF.shape()	S.shape()
Get or set shape (length of each axis)	DF.head(n)
DF.tail(n)	Return n items (default 5) from beginning or end
DF.values	S.values
Get the actual values from a data structure	DF.loc[row_indexer, col_indexer]
Multi-axis indexing by label (not by position)	DF.iloc[row_indexer, col_indexer]
Multi-axis indexing by integer position (not by labels)	DF.ix[row_indexer, col_indexer]

Table 26. Methods for Computations/Descriptive Stats

Method	Returns
abs()	absolute values
corr()	pairwise correlations
count()	number of values
cov()	Pairwise covariance
cumsum()	cumulative sums
cumprod()	cumulative products
cummin(), cummax()	cumulative minimum, maximum
kurt()	unbiased kurtosis
median()	median
min(), max()	minimum, maximum values
prod()	products
quantile()	values at given quantile
skew()	unbiased skewness
std()	standard deviation
var()	variance

NOTE

these methods return Series or DataFrames, as appropriate, and can be computed over rows (axis=0) or columns (axis=1). They generally skip NA/null values.

Reading Data

- Supports many data formats
- Reads headings to create column indexes
- Auto-creates indexes as needed
- Can use specified column as row index

pandas support many different input formats. It will reading file headings and use them to create column indexes. By default, it will use integers for row indices, but you can specify a column to use as the index.

The read methods have many options for controlling and parsing input. For instance, if large integers in the file contain commas, the thousands options let you set the separator as comma (in the US), so it will ignore them.

NOTE

See [Jupyter notebook Pandas_Input_Demo](#) for examples of reading most types of input.

Table 27. Pandas I/O methods

Methods	Reads the following into a DataFrame
<code>read_table()</code>	Generic delimited file
<code>read_csv()</code>	CSV file
<code>read_fwf()</code>	File with fixed-width fields
<code>read_clipboard()</code>	Data from OS clipboard (passed to <code>read_table()</code>)
<code>read_excel()</code>	Excel table
<code>read_html()</code>	HTML table
<code>read_hdf()</code>	HDF5 Store (using PyTables)
<code>read_sql_query()</code>	Result of SQL query

Example

pandas_read_csv.py

```
#!/usr/bin/env python
import pandas as pd
from printhead import print_header

# data from
# http://www.rita.dot.gov/bts/sites/rita.dot.gov.bts/files/publications/
# national_transportation_statistics/html/table_01_44.html

airports_df = pd.read_csv('../DATA/airport_boardings.csv', thousands=',',
index_col=1) ①

print_header("HEAD OF DATAFRAME")

print(airports_df.head(), "\n")

print_header("SELECTED COLUMNS WITH FILTERED ROWS")
columns_wanted = ['2001 Total', 'Airport']
sort_col = '2001 Total'
max_val = 20000000
selector = airports_df['2001 Total'] > max_val
selected = airports_df[selector][columns_wanted]
print(selected)

print_header("COLUMN TOTALS")
print(airports_df[['2001 Total', '2010 Total']].sum(), "\n")

# print_header("'CODE' COLUMN SET AS INDEX")
# airports_df.set_index('Code')
# print(airports_df)

print_header("FIRST FIVE ROWS")
print(airports_df.iloc[:5])
```

- ① Read CSV file into dataframe; parse numbers containing commas, use first column as row index.

pandas_read_csv.py

```

=====
=                HEAD OF DATAFRAME                =
=====
                                Airport                ...
Percent change 2010-2011
Code                                                    ...
ATL  Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...
-22.6
ORD      Chicago, IL (Chicago O'Hare International)    ...
-25.5
DFW      Dallas, TX (Dallas/Fort Worth International)  ...
-23.7
DEN              Denver, CO (Denver International)    ...
-23.1
LAX      Los Angeles, CA (Los Angeles International)  ...
-19.6

[5 rows x 9 columns]

=====
=  SELECTED COLUMNS WITH FILTERED ROWS  =
=====
      2001 Total                                Airport
Code
ATL    36384264  Atlanta, GA (Hartsfield-Jackson Atlanta Intern...
ORD    28626694      Chicago, IL (Chicago O'Hare International)
DFW    25198256      Dallas, TX (Dallas/Fort Worth International)
LAX    22873307      Los Angeles, CA (Los Angeles International)
=====
=                COLUMN TOTALS                =
=====
2001 Total    492424399
2010 Total    558240706
dtype: int64

=====
=                FIRST FIVE ROWS                =
=====
                                Airport                ...
Percent change 2010-2011
Code                                                    ...
ATL  Atlanta, GA (Hartsfield-Jackson Atlanta Intern...  ...

```

```
-22.6
ORD      Chicago, IL (Chicago O'Hare International)      ...
-25.5
DFW      Dallas, TX (Dallas/Fort Worth International)    ...
-23.7
DEN      Denver, CO (Denver International)              ...
-23.1
LAX      Los Angeles, CA (Los Angeles International)     ...
-19.6

[5 rows x 9 columns]
```

Chapter 18 Exercises

Exercise 18-1 (simple_dataframe.py)

Create a DataFrame with columns named 'Test1', 'Test2', up to 'Test6'. Use default row indexes. Fill the DataFrame with random values.

- Print only columns 'Test3' and 'Test5'.
- Print the dataframe with every value multiplied by 3.6

Exercise 18-2 (parasites.py)

The file `parasite_data.csv`, in the DATA folder, has some results from analysis on some intestinal parasites (not that it matters for this exercise...). Read `parasite_data.csv` into a DataFrame. Print out all rows where the Shannon Diversity is ≥ 1.0 .

Chapter 19: Introduction to Matplotlib

Objectives

- Understand what matplotlib can do
- Create many kinds of plots
- Label axes, plots, and design callouts

About matplotlib

- matplotlib is a package for making 2D plots
- Emulates MATLAB®, but not a drop-in replacement
- matplotlib's philosophy: create simple plots simply
- Plots are publication quality
- Plots can be rendered in GUI applications

This chapter's discussion of matplotlib will use the iPython notebook named **MatplotlibExamples.ipynb**. Please start the iPython notebook server and load this notebook, as directed by the instructor.

matplotlib architecture

- pylab/pyplot front end plotting functions
- API create/manage figures, text, plots
- backends device-independent renderers

matplotlib consists of roughly three parts: pylab/pyplot, the API, and the backends.

pyplot is a set of functions which allow the user to quickly create plots. Pyplot functions are named after similar functions in MATLAB.

The API is a large set of classes that do all the work of creating and manipulating plots, lines, text, figures, and other graphic elements. The API can be called directly for more complex requirements.

pylab combines pyplot with numpy. This makes pylab emulate MATLAB more closely, and thus is good for interactive use, e.g., with iPython. On the other hand, pyplot alone is very convenient for scripting. The main advantage of pylab is that it imports methods from both pyplot and pylab.

There are many backends which render the in-memory representation, created by the API, to a video display or hardcopy format. For example, backends include PS for PostScript, SVG for scalable vector graphics, and PDF.

The normal import is

```
import matplotlib.pyplot as plt
```

Matplotlib Terminology

- Figure
- Axis
- Subplot

A Figure is one "picture". It has a border ("frame"), and other attributes. A Figure can be saved to a file.

A Plot is one set of values graphed onto the Figure. A Figure can contain more than one Plot.

Axes and Subplot are similar; the difference is how they get placed on the figure. Subplots allow multiple plots to be placed in a rectangular grid. Axes allow multiple plots to be placed at any location, including within other plots, or overlapping.

matplotlib uses default objects for all of these, which are sufficient for simple plots. You can explicitly create any or all of these objects to fine-tune a graph. Most of the time, for simple plots, you can accept the defaults and get great-looking figures.

Matplotlib Keeps State

- Primary method is `matplotlib.pyplot()`
- The current figure can have more than one plot
- Calling `show()` displays the current figure

matplotlib.pyplot is the workhorse of figure drawing. It is usually aliased to "plt".

While Matplotlib is object oriented, and you can manually create figures, axes, subplots, etc., `pyplot()` will create a figure object for you automatically, and commands called from `pyplot()` (usually through the **plt** alias) will work on that object.

Calling **plt.plot()** plots one set of data on the current figure. Calling it again adds another plot to the same figure.

`plt.show()` displays the figure, although iPython may display each separate plot, depending on the current settings.

You can pass one or two datasets to `plot()`. If there are two datasets, they need to be the same length, and represent the x and y data.

What Else Can You Do?

- Multiple plots
- Control ticks on any axis
- Scatter plots
- Polar axes
- 3D Plots
- Quiver plots
- Pie Charts

There are many other types of drawings that `matplotlib` can create. Also, there are many more style details that can be tweaked. See <http://matplotlib.org/gallery.html> for dozens of sample plots and their source.

There are many extensions (AKA toolkits) for Matplotlib, including Seaborn, Cartopy, at Natgrid.

Chapter 19 Exercises

Exercise 19-1 (energy_use_plot.py)

Using the file `energy_use_quad.csv` in the `DATA` folder, use `matplotlib` to plot the data for "Transportation", "Industrial", and "Residential and Commercial". Don't plot the "as a percent...".

You can do this in `iPython`, or as a standalone script. If you create a standalone script, save the figure to a file, so you can view it.

Use `pandas` to read the data. The columns are, in Python terms:

```
['Desc', "1960", "1965", "1970", "1975", "1980", "1985", "1990", "1991", "1992", "1993", "1994",  
 "1995", "1996", "1997", "1998", "1999", "2000", "2001", "2002", "2003", "2004", "2005", "2006",  
 "2007", "2008", "2009", "2010", "2011"]
```

TIP

See the script `pandas_energy.py` in the `EXAMPLES` folder to see how to load the data.

Appendix A: Where do I go from here?

Resources for learning Python

These are from Jessica Garson, who, among other things, teaches Python classes at NYU. (Used with permission).

Run the script `where_do_i_go.py` to display a web page with live links.

[Resources for Learning Python](https://dev.to/jessicagarson/resources-for-learning-python-hd6) [https://dev.to/jessicagarson/resources-for-learning-python-hd6]

Just getting started

Here are some resources that can help you get started learning how to code.

- [Code Newbie Podcast](https://www.codenewbie.org/podcast) [https://www.codenewbie.org/podcast]
- [Dive into Python3](http://www.diveintopython3.net) [http://www.diveintopython3.net]
- [Learn Python the Hard Way](https://learnpythonthehardway.org/python3) [https://learnpythonthehardway.org/python3]
- [Learn Python the Hard Way](https://learnpythonthehardway.org/python3) [https://learnpythonthehardway.org/python3]
- [Automate the Boring Stuff with Python](https://automatetheboringstuff.com) [https://automatetheboringstuff.com]
- [Automate the Boring Stuff with Python](https://automatetheboringstuff.com) [https://automatetheboringstuff.com]

So you want to be a data scientist?

- [Data Wrangling with Python](https://www.amazon.com/Data-Wrangling-Python-Tools-Easier/dp/1491948817) [https://www.amazon.com/Data-Wrangling-Python-Tools-Easier/dp/1491948817]
- [Data Analysis in Python](http://www.data-analysis-in-python.org/index.html) [http://www.data-analysis-in-python.org/index.html]
- [Titanic: Machine Learning from Disaster](https://www.kaggle.com/c/titanic/discussion/5105) [https://www.kaggle.com/c/titanic/discussion/5105]
- [Deep Learning with Python](https://www.manning.com/books/deep-learning-with-python) [https://www.manning.com/books/deep-learning-with-python]
- [How to do X with Python](https://chrisalbon.com/) [https://chrisalbon.com/]
- [Machine Learning: A Probabilistic Prospective](https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020) [https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020]

So you want to write code for the web?

- [Learn flask, some great resources are listed here](https://www.fullstackpython.com/flask.html) [https://www.fullstackpython.com/flask.html]

- [Django Polls Tutorial](https://docs.djangoproject.com/en/2.0/intro/tutorial01/) [https://docs.djangoproject.com/en/2.0/intro/tutorial01/]
- [Hello Web App](https://www.amazon.com/Hello-Web-App-Learn-Build-ebook/dp/B00U5MMZ2E/ref=sr_1_1?ie=UTF8&qid=1510599119&sr=8-1&keywords=hello+web+app) [https://www.amazon.com/Hello-Web-App-Learn-Build-ebook/dp/B00U5MMZ2E/ref=sr_1_1?ie=UTF8&qid=1510599119&sr=8-1&keywords=hello+web+app]
- [Hello Web App Intermediate](https://www.amazon.com/Hello-Web-App-Intermediate-Concepts/dp/0986365920) [https://www.amazon.com/Hello-Web-App-Intermediate-Concepts/dp/0986365920]
- [Test-Driven-Development for Web Programming](https://www.obeythetestinggoat.com/pages/book.html#toc) [https://www.obeythetestinggoat.com/pages/book.html#toc]
- [2 Scoops of Django](https://www.amazon.com/Two-Scoops-Django-1-11-Practices-ebook/dp/B076D5FKFX/ref=sr_1_1?s=books&ie=UTF8&qid=1510598897&sr=1-1&keywords=2+scoops+of+django) [https://www.amazon.com/Two-Scoops-Django-1-11-Practices-ebook/dp/B076D5FKFX/ref=sr_1_1?s=books&ie=UTF8&qid=1510598897&sr=1-1&keywords=2+scoops+of+django]
- [HTML and CSS: Design and Build Websites](https://www.amazon.com/HTML-CSS-Design-Build-Websites/dp/1118008189/ref=sr_1_1?ie=UTF8&qid=1510599157&sr=8-1&keywords=css+and+html) [https://www.amazon.com/HTML-CSS-Design-Build-Websites/dp/1118008189/ref=sr_1_1?ie=UTF8&qid=1510599157&sr=8-1&keywords=css+and+html]
- [JavaScript and JQuery](https://www.amazon.com/JavaScript-JQuery-Interactive-Front-End-Development/dp/1118531647) [https://www.amazon.com/JavaScript-JQuery-Interactive-Front-End-Development/dp/1118531647]

Not sure yet, that's okay!

Here are some resources for self guided learning. I recommend trying to be very good at Python and the rest should figure itself out in time.

- [Python 3 Crash Course](https://www.amazon.com/Python-Crash-Course-Hands-Project-Based/dp/1593276036) [https://www.amazon.com/Python-Crash-Course-Hands-Project-Based/dp/1593276036]
- [Base CS Podcast](https://www.codenewbie.org/basescs) [https://www.codenewbie.org/basescs]
- [Writing Idiomatic Python](https://www.amazon.com/Writing-Idiomatic-Python-Jeff-Knupp-ebook/dp/B00B5VXMRG) [https://www.amazon.com/Writing-Idiomatic-Python-Jeff-Knupp-ebook/dp/B00B5VXMRG]
- [Fluent Python](https://www.amazon.com/dp/1491946008?axitk=o7.Y1C9z7oJp87fs3ev30Q&pd_rd_i=1491946008&hsa_cr_id=1406361870001) [https://www.amazon.com/dp/1491946008?axitk=o7.Y1C9z7oJp87fs3ev30Q&pd_rd_i=1491946008&hsa_cr_id=1406361870001]
- [Pro Python](https://www.amazon.com/Pro-Python-Marty-Alchin/dp/1484203356/ref=sr_1_1?s=books&ie=UTF8&qid=1510598874&sr=1-1&keywords=pro+python) [https://www.amazon.com/Pro-Python-Marty-Alchin/dp/1484203356/ref=sr_1_1?s=books&ie=UTF8&qid=1510598874&sr=1-1&keywords=pro+python]
- [Refactoring](https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672/ref=sr_1_1?ie=UTF8&qid=1510598784&sr=8-1&keywords=refactoring+martin+fowler) [https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672/ref=sr_1_1?ie=UTF8&qid=1510598784&sr=8-1&keywords=refactoring+martin+fowler]
- [Clean Code](https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?s=books&ie=UTF8&qid=1510598926&sr=1-1&keywords=clean+code) [https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?s=books&ie=UTF8&qid=1510598926&sr=1-1&keywords=clean+code]
- [Write music with Python, since that's my favorite way to learn a new language](https://github.com/reckoner165/soundmodular) [https://github.com/reckoner165/soundmodular]

Appendix B: Python Bibliography

Title	Author	Publisher
Data Science		
Building machine learning systems with Python	Willi Richert, Luis Pedro Coelho	Packt Publishing
High Performance Python	Mischa Gorlelick and Ian Ozsvald	O'Reilly Media
Introduction to Machine Learning with Python	Sarah Guido	O'Reilly & Assoc.
iPython Interactive Computing and Visualization Cookbook	Cyril Rossant	Packt Publishing
Learning iPython for Interactive Computing and Visualization	Cyril Rossant	Packt Publishing
Learning Pandas	Michael Heydt	Packt Publishing
Learning scikit-learn: Machine Learning in Python	Raúl Garreta, Guillermo Moncecchi	Packt Publishing
Mastering Machine Learning with Scikit-learn	Gavin Hackeling	Packt Publishing
Matplotlib for Python Developers	Sandro Tosi	Packt Publishing
Numpy Beginner's Guide	Ivan Idris	Packt Publishing
Numpy Cookbook	Ivan Idris	Packt Publishing
Practical Data Science Cookbook	Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta	Packt Publishing
Python Text Processing with NLTK 2.0 Cookbook	Jacob Perkins	Packt Publishing
Scikit-learn cookbook	Trent Hauck	Packt Publishing
Python Data Visualization Cookbook	Igor Milovanovic	Packt Publishing
Python for Data Analysis	Wes McKinney	O'Reilly & Assoc.
Design Patterns		

Title	Author	Publisher
Design Patterns: Elements of Reusable Object-Oriented Software	Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides	Addison-Wesley Professional
Head First Design Patterns	Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra	O'Reilly Media
Learning Python Design Patterns	Gennadiy Zlobin	Packt Publishing
Mastering Python Design Patterns	Sakis Kasampalis	Packt Publishing
General Python development		
Expert Python Programming	Tarek Ziadé	Packt Publishing
Fluent Python	Luciano Ramalho	O'Reilly & Assoc.
Learning Python, 2nd Ed.	Mark Lutz, David Asher	O'Reilly & Assoc.
Mastering Object-oriented Python	Stephen F. Lott	Packt Publishing
Programming Python, 2nd Ed.	Mark Lutz	O'Reilly & Assoc.
Python 3 Object Oriented Programming	Dusty Phillips	Packt Publishing
Python Cookbook, 3rd. Ed.	David Beazley, Brian K. Jones	O'Reilly & Assoc.
Python Essential Reference, 4th. Ed.	David M. Beazley	Addison-Wesley Professional
Python in a Nutshell	Alex Martelli	O'Reilly & Assoc.
Python Programming on Win32	Mark Hammond, Andy Robinson	O'Reilly & Assoc.
The Python Standard Library By Example	Doug Hellmann	Addison-Wesley Professional
Misc		
Python Geospatial Development	Erik Westra	Packt Publishing

Title	Author	Publisher
Python High Performance Programming	Gabriele Lanaro	Packt Publishing
Networking		
Python Network Programming Cookbook	Dr. M. O. Faruque Sarker	Packt Publishing
Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers	T J O'Connor	Syngress
Web Scraping with Python	Ryan Mitchell	O'Reilly & Assoc.
Testing		
Python Testing Cookbook	Greg L. Turnquist	Packt Publishing
Learning Python Testing	Daniel Arbuckle	Packt Publishing
Learning Selenium Testing Tools, 3rd Ed.	Raghavendra Prasad MG	Packt Publishing
Web Development		
Building Web Applications with Flask	Italo Maia	Packt Publishing
Django 1.0 Website Development	Ayman Hourieh	Packt Publishing
Django 1.1 Testing and Development	Karen M. Tracey	Packt Publishing
Django By Example	Antonio Melé	Packt Publishing
Django Design Patterns and Best Practices	Arun Ravindran	Packt Publishing
Django Essentials	Samuel Dauzon	Packt Publishing
Django Project Blueprints	Asad Jibran Ahmed	Packt Publishing
Flask Blueprints	Joel Perras	Packt Publishing
Flask by Example	Gareth Dwyer	Packt Publishing
Flask Framework Cookbook	Shalabh Aggarwal	Packt Publishing

Title	Author	Publisher
Flask Web Development	Miguel Grinberg	O'Reilly & Assoc.
Full Stack Python (e-book only)	Matt Makai	Gumroad (or free download)
Full Stack Python Guide to Deployments (e-book only)	Matt Makai	Gumroad (or free download)
High Performance Django	Peter Baumgartner, Yann Malet	Lincoln Loop
Instant Flask Web Development	Ron DuPlain	Packt Publishing
Learning Flask Framework	Matt Copperwaite, Charles O Leifer	Packt Publishing
Mastering Flask	Jack Stouffer	Packt Publishing
Two Scoops of Django: Best Practices for Django 1.11	Daniel Roy Greenfeld, Audrey Roy Greenfeld	Two Scoops Press
Web Development with Django Cookbook	Aidas Bendoraitis	Packt Publishing

Index

@

- @property, 281
- @staticmethod, 287
- __init__.py, 182
- __pycache__, 171

A

- Anaconda, 335
- Array types, 78

B

- Boolean operators, 69
- benchmarking, 308
- break statement, 72
- builtin functions, 27
 - table, 28

C

- CSV, 369
 - nonstandard, 371
- class
 - constructor, 277
 - definition, 275
 - instance methods, 278
 - private method, 288
 - static method, 287
- class data, 285
- class method, 285
- collection vs generator, 259
- command line parameters, 56
- comments, 297
- conditional expression, 66
- continue statement, 72
- csv
 - DictReader, 373
- csv.reader(), 369
- csv.writer(), 375

D

- DOM, 333-334
- Douglas Crockford, 352
- date, 230
- dates and times, 230
- datetime, 230
- debugger
 - setting breakpoints, 305
 - starting, 303
 - stepping through a program, 304
- dictionary comprehension, 256

E

- Element, 336-337
- ElementTree, 335
 - find(), 344
 - findall(), 344
- Excel, 314
- Excel 2007, 314
- Explicit execution, 17
- editors and IDEs, 22
- email.mime, 220
- enumerate(), 98
- exceptions, 191
 - else, 196
 - finally, 198
 - generic, 194
 - ignoring, 195
 - list, 202
 - multiple, 193

F

- file(), 127
- flow control, 62
- for loop, 87
- format, 51
- formatting, 51
- function parameters, 159

- default, 163
 - named, 159
 - optional, 159
 - positional, 159
 - required, 159
 - functions, 156
- G*
- Graphviz, 301
 - generator expression, 261
 - getroot(), 343
 - global, 169
 - grabbing data from the web, 217
- H*
- HTTP download, 217
 - hello, world, 23
- I*
- if statement, 64
 - if-else, 66
 - if/elif/else, 64
 - implicit execution
 - Windows, 18
 - non-Windows, 19
 - import *, 176
 - import {star}, 175
 - in, 101
 - in operator, 259
 - indentation, 63
 - indexing, 84
 - inheritance, 289
 - instance methods, 278
 - interpreter attributes, 206
 - iterable, 91
 - iterating through a sequence, 87
- J*
- JSON, 352
 - custom encoding, 359
 - types, 352
 - json module, 353
 - json.dumps(), 356
 - json.loads(), 353
- K*
- keywords, 27
- L*
- lambda function, 252
 - launching external programs, 208
 - legacy string formatting, 55
 - len(), 95
 - list comprehension, 254
 - list methods
 - table, 81
 - lists, 80
 - literal Unicode characters, 34
 - lxml
 - Element, 337
 - SubElement, 337
 - lxml.etree, 334
- M*
- Math operators and expressions, 45
 - math functions, 226
 - math operators
 - table, 47
 - matplotlib.pyplot, 497
 - max(), 95
 - min(), 95
 - modules, 170
 - documenting, 185
 - executing as scripts, 178
 - importing, 171
 - search path, 177
- N*
- None, 26
 - nested sequences, 92
 - nonlocal, 169
 - numeric literals, 43

numpy, 434

O

object-oriented programming, 274

openpyxl, 314

os.path, 210

os.system(), 208

os.walk(), 214

P

PATH variable

- Linux, 14

- OS X, 14

- Unix, 14

- Windows, 14

- non-Windows, 14

PEP 20, 238

PEP 8, 186

PYTHONPATH, 177

PyCharm, 22

Python IDEs, 22

Python Interpreter, 15

Python scripts, 17

packages, 180

- configuring, 182

plt.plot(), 497

plt.show(), 497

popen(), 208

print() function, 49

private method, 288

profiler, 306

properties, 281

pycallgraph, 307

pychecker, 298

pyflakes, 298

pylint, 298

- customizing, 299

pyreverse, 300

python debugger, 302

python style, 186

R

Reading from the keyboard, 57

Relational Operators, 67

random, 227

random.choice(), 227

random.randint(), 227

random.randrange(), 227

random.sample(), 227

random.shuffle(), 227

raw strings, 33

read(), 127

readline(), 127

readlines(), 127

reversed(), 95

S

SAX, 333-334

Sequences, 78

SubElement, 337

scope

- builtin, 166

- global, 166

- nonlocal, 166

self, 278

sending email, 220

sequence functions, 95

set comprehension, 258

sh, 208

singledispatch, 359

slicing, 84

smtplib, 220

sorted(), 95, 247

sorting

- custom key, 249

standard I/O, 207

standard exception hierarchy, 202

starting python, 12

static method, 287

stderr, 207

stdin, 207

- stdio, 207
- stdout, 207
- string formatting, 51
- string literals, 31
- string methods, 37
- string operators, 37
- strings, 30
- subprocess, 208
- sum(), 95
- syntax errors, 190
- sys module, 206
- sys.executable, 206
- sys.modules, 206
- sys.path, 177, 206
- sys.platform, 206
- sys.prefix, 206
- sys.version, 206

T

- Tim Peters, 238
- time, 230
- timedelta, 230
- timeit, 308
- timsort, 238
- triple-delimited strings, 32
- triple-quoted strings, 32
- tuple, 239
- tuple unpacking, 91
- type conversions, 48

U

- unicode, 30
- unpacking function parameters, 242
- urllib module, 217
- urlopen(), 217
- using pydoc, 21
- using try/except, 192

V

- variable scope, 166
- variable typing, 29

- variables, 26

W

- walking directory trees, 214
- while loop, 71
- whitespace, 63
- write(), 127
- writelines(), 127

X

- XML, 333-334
 - root element, 340
- XPath, 348
- xml.etree.ElementTree, 334-335

Y

- yield, 261

Z

- Zen of Python, 238
- zip(), 95
- zipfile, 233
- zipped archives, 233



Trivera Technologies LLC - Worldwide | Educate. Collaborate. Accelerate!
Global Developer Education, Courseware & Consulting Services

JAVA | JEE | OOAD | UML | XML | Web Services | SOA | Struts | JSF | Hibernate | Spring | Admin

IBM WebSphere | Rational | Oracle WebLogic | JBoss | TomCat | Apache | Linux | Perl

609.953.1515 direct | Training@triveratech.com | www.triveratech.com