

Introduction to Python Programming

TTPS4802-GKJ

Student Guide



Experience is
Everything

Trivera Technologies

Instructor-Led Training, Coaching & Skills Development

Solutions: *Experience is Everything!*

Welcome to Trivera Technologies, the premier provider for collaborative and innovative IT education! As a top-tier TechEd services firm, we are devoted to arming IT professionals across all career stages with the latest skills and best practices. Our expertly designed programs don't just teach leading-edge technical skills, they also instill strong problem-solving capabilities and bolster confidence in our learners. Our methodology is rooted in the principle that real-world application builds confidence, and so, our programs are structured to ensure you're equipped to use your newly acquired skills in your job from day one.

Our extensive portfolio includes hundreds of courses spanning a wide variety of cutting edge topics, tools, best practices and in-demand skills, ready to apply on the job. Our diverse catalog ensures we have something to cater to every unique learning need. Our rich public enrollment schedule offers flexibility and accessibility, ensuring you can learn at a pace that suits you. Our team is composed of experienced instructors who are industry practitioners at heart, ensuring that real-world insights are integrated into every course and project. We can train individuals, empower teams, and even drive a complete business transformation for enterprise-level organizations.

Services We Offer Include...

- Onsite, Online and Blended Learning Solutions
- Hundreds of Public / Open Enrollment Courses
- Tailored Skills-Focused Training Programs
- User-Friendly, Hands-on Learning Experience Platform for real time and long term success
- AI Driven Training Plans
- Unique, Secure CodeCoach.AI Superhero Tutor Access
- Engaging Expert Coaching and Mentoring Services
- Skills Assessments & Gap Training
- Skills-Immersion, Re-Skilling and Up-Skilling Solutions and SkillJourneys for Teams of All Sizes
- New Hire / Cohort Camps / Boot Camps
- Enterprise-Wide, Business Transformation Solutions
- Turn-Key Partner and Reseller Programs
- Free Blogs, Free Training Series, Free Webinars
- Courseware Development & Licensing
- Regular Corporate Discounts & Special Offers
- Pricing and Satisfaction Guarantee



Subscribe for Free Training, Tech Skills Playlists & Webinars!
www.youtube.com/@triveratech

Follow Us for News & Offers!

www.linkedin.com/company/triveratech
www.X.com/TriveraTech
www.facebook.com/triveratech

Some of the Skills We Cover Include:

(Please visit www.triveratech.com for our complete catalog)

AI & Machine Learning: Artificial Intelligence / AI • Generative AI • GPT / Bard / Copilot / Gemini • OpenAI • AzureAI • Applied AI • Automation • Prompt Engineering • AI for Business Users / Stakeholders • AI for Technical Users / Developers • AIOps • Machine Learning Algorithms • NLP • Neural Networks • Vision Systems Automation • Hadoop • Kafka • Spark • R • Scala • Python

Big Data, Data Science & Analytics: Data Science • Ecosystem • Analytics • Data Literacy • Visualization • Modernization • Reporting • Python Data Science • R

Python: Basics • Non-Developers • Advanced • Security Networking / SysAdmin • Python Data Science & Analytics Python in AI & Machine Learning • Python FullStack & Web Test Automation

Business Intelligence Tools: Tableau • PowerBI • Looker Snowflake • Informatica

DevOps: DevOps • Jira • Git, GitHub, GitLab, TortoiseGit Jenkins • Docker • Kafka • NoSQL • Containers • DevSecOps

Application Development, Programming & Coding: Intro to Coding • Java / JEE • Full Stack • Java Secure Coding • Spring / Core / Boot / Data / Web / MVC / Cloud • C++ Programming / All Levels • Microsoft C# / .Net • ASP.Net • .Net Secure Coding • Microservices • Web Services • RESTful Agile Development • Cloud Development

Modern Web Development & Design: HTML5 / CSS3 JavaScript • JavaScript Libraries • React / Redux • Angular Full Stack • MEANStack / MERNStack • Node.js • UX / UI • Responsive Design • Python • Perl • PHP • XML • SQL

Mobile Development: Android • Kotlin • iOS • Swift • Mobile Testing • Secure Mobile Development

Security: Secure Software Design • Secure Coding • Secure Web Development • Database Security • OWASP • STIG • Cryptography • Secure Python

Software Testing, TDD & Test Automation: Test Driven Development • Unit Testing • JUnit / NUnit • Selenium • Cucumber • QA • Test Automation

Databases: Database Design • Database Security • RDBMS • SQL • PL/SQL • Oracle • MySQL • SQLServer • MongoDB • NoSQL • POSTGresSQL • MariaDB • Cassandra

O/S: Windows • Linux / UNIX • iOS • Administration

Oracle Tool & Databases: 19c / 23c • DBA I • DBA II • SQL • PL/SQL • Multithreading • New Features • OEM • ODI • Performance Tuning • Cloud • Backup & Recovery

Software Architecture, Design & Engineering: Architecture • Analysis • Requirements • Estimation • Use Cases • OO • Data Modeling & Design • Software Design

Table of Contents

About Introduction to Python for JPMC: TTPS4800.....	1
Course Outline	2
Student files	3
Examples.....	4
Appendices	5
Classroom etiquette	6
Chapter 1: The Python Environment.....	7
Starting Python	8
If the interpreter is not in your PATH	9
Using the interactive interpreter.....	10
Trying out a few commands	11
Running Python scripts.....	12
Using the <code>help()</code> function.....	13
Python Editors and IDEs.....	15
For more information	16
Chapter 2: Setting up Visual Studio.....	18
About Visual Studio Code.....	19
Getting started.....	20
Configuration	21
Opening a project	23
Creating a Python script.....	23
Running the script	24
Chapter 3: Variables and values	26
Using variables.....	27
Keywords and Builtin functions.....	28
Variable typing.....	30
Strings	31
Single-delimited string literals.....	32
Triple-delimited string literals	33
Raw string literals	34
Unicode characters	35
String operators and methods	38
Numeric literals	43
Math operators and expressions	45
Converting among types	48
Chapter 4: Basic input and output	51

Writing to the screen	52
Format strings	54
Using the <code>.format()</code> method	58
Legacy String Formatting	59
Command line parameters	62
Reading from the keyboard	64
Chapter 5: Flow Control	66
About flow control	67
if and elif	68
What's with the white space?	70
The Conditional Expression	71
Relational Operators	72
Boolean operators	74
while loops	76
Loop control	77
Chapter 6: Array Types	81
About Array Types	82
Lists	84
Indexing and slicing	88
Iterating through a sequence	92
Tuples	95
Iterable Unpacking	97
Nested sequences	100
Operators and keywords for sequences	102
Functions for all sequences	104
Iterators	106
List comprehensions	111
Generator Expressions	114
Chapter 7: Working with Files	120
Text file I/O	121
Opening a text file	122
Reading a text file	124
Processing files from the command line	128
Writing to a text file	130
Modifying text files	131
Chapter 8: Dictionaries and Sets	134
About dictionaries	135
Creating dictionaries	136

Getting dictionary values	139
Iterating over a dictionary	142
Reading file data into a dictionary	144
Counting with dictionaries	146
About sets	148
Creating Sets	149
Working with sets	150
Chapter 9: Functions	155
Defining a function	156
Function parameters	158
Returning values	165
Variable scope	166
Chapter 10: Creating and using modules	172
What is a module?	173
Creating Modules	174
The import statement	177
Import module	178
Import module with alias	179
Import names from module	180
Import all names from module	181
Import names with aliases	182
Where did <code>_pycache_</code> come from?	183
Module search path	184
Executing modules as scripts	186
Packages	188
Example	189
<code>__init__</code> module	190
When the batteries aren't included	191
Chapter 11: Errors and Logging	193
Exceptions	194
Handling exceptions with try	195
Handling multiple exceptions	196
Handling generic exceptions	197
Ignoring exceptions	198
Using else	199
Cleaning up with finally	201
Simple Logging	205
Formatting log entries	207

Logging exception information	211
For more information	213
Chapter 12: Introduction to Python Classes	215
About object-oriented programming	216
Defining classes	217
Constructors	225
Instance methods	226
Properties	227
Class methods and data	230
Static Methods	232
Private methods	233
Inheritance	234
Untangling the nomenclature	237
Chapter 13: Sorting	240
Sorting Overview	241
The sorted() function	242
Custom sort keys	243
Lambda functions	248
Sorting nested data	251
Sorting dictionaries	254
Sorting lists in place	256
Sorting in reverse	257
Chapter 14: Regular Expressions	260
Regular expressions	261
RE syntax overview	262
Finding matches	264
RE objects	267
Compilation flags	270
Working with newlines	274
Groups	277
Special groups	280
Replacing text	282
Replacing with backrefs	284
Replacing with a callback	287
Splitting a string	289
Chapter 15: Using the Standard Library	292
The sys module	293
Interpreter Information	294

STDIO.....	295
Launching external programs.....	296
Paths, directories and filenames.....	298
Walking directory trees.....	301
Grabbing data from the web.....	304
Sending email	307
math functions.....	313
Random values	314
Dates and times.....	317
Zipped archives	320
Appendix A: Where do I go from here?	323
Resources for learning Python	323
Appendix B: Python Bibliography	325
Appendix C: An Overview of Python	328
What is Python?.....	329
The Birth of Python	330
About Interpreted Languages.....	332
Advantages of Python.....	333
Disadvantages of Python.....	334
How to get Python.....	335
Which version of Python?.....	336
The end of Python 2	337
Getting Help	338
Appendix D: String Formatting	339
Overview	339
Parameter Selectors	340
f-strings.....	342
Data types.....	343
Field Widths	346
Alignment	349
Fill characters	352
Signed numbers	354
Parameter Attributes	357
Formatting Dates.....	359
Run-time formatting	363
Miscellaneous tips and tricks.....	365
Index	367

About Introduction to Python for JPMC: TTPS4800

Course Outline

Half-day 1

- Chapter 1** The Python Environment
- Chapter 2** Setting up Visual Studio
- Chapter 3** Variables and values
- Chapter 4** Basic input and output

Half-day 2

- Chapter 5** Flow Control
- Chapter 6** Array Types

Half-day 3

- Chapter 7** Working with Files
- Chapter 8** Dictionaries and Sets
- Chapter 9** Functions

Half-day 4

- Chapter 10** Creating and using modules
- Chapter 11** Errors and Logging
- Chapter 12** Introduction to Python Classes

If time permits...

- Chapter 13** Sorting
- Chapter 14** Regular Expressions
- Chapter 15** Using the Standard Library



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

Student files

You will need to load some student files onto your computer. The files are in a compressed archive. When you extract them onto your computer, they will all be extracted into a directory named **bpyjpmcintro**. See the setup guides for details.

What's in the files?

bpyjpmcintro contains all files necessary for the class

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

bpyjpmcintro/ANSWERS/ contains sample answers to the labs.

bpyjpmcintro/DATA/ contains data used in examples and answers

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

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

The following folders *may* be present:

bpyjpmcintro/BIG_DATA/ contains large data files used in examples and answers

bpyjpmcintro/NOTEBOOKS/ Jupyter notebooks for use in class

bpyjpmcintro/LOGS/ initially empty; used by some examples to write log files



The student files do not contain Python itself. It will need to be installed separately. This may already have been done.

Examples

Most of the examples from the course manual are provided in EXAMPLES subdirectory.

It will look like this:

Example

cmd_line_args.py

```
import sys    # Import the sys module

print(sys.argv) # Print all parameters, including script itself

name = sys.argv[1] # Get the first actual parameter
print("name is", name)
```

cmd_line_args.py apple mango 123

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

Appendices

Appendix A [Where do I go from here?](#)

Appendix B [Python Bibliography](#)

Appendix C [An Overview of Python](#)

Appendix D [String Formatting](#)

Classroom etiquette

Remote learning

- Mic off when you're not speaking. If multiple mics are on, it makes it difficult to hear
- The instructor doesn't know you need help unless you let them know via voice or chat.
- It's ok to ask for help a lot.
 - Ask questions. Ask questions. Ask questions.
 - **INTERACT** with the instructor and other students.
- Log off the remote S/W at the end of the day

In-person learning

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

Please turn off cell phone ringers and other noisemakers.

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

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



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

Chapter 1: The Python Environment

Objectives

- Using the interactive interpreter
- Running scripts
- Getting help
- Learning about 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

- Open Anaconda prompt, command prompt, or terminal window
 - Type `python`
- `python` *should* be in the search path
- If `python` not found
 - Install Python
 - Add folder with Python executable to `PATH`

To start the Python interpreter, open an Anaconda Prompt, a command window (Windows) or a terminal prompt (Mac/Linux). Type `python`. If you get an error message, one of three things has happened:

- Python is not installed on your computer
- The folder containing the interpreter (`python`) is not in your `PATH` variable
- `python` is aliased to `python3`.

If you're using the **Anaconda** distribution of Python, use the **Anaconda Prompt**, which is a command prompt or terminal window with the `PATH` variable already set for the Python interpreter.

On some older systems you may need to type `python3` to start the interpreter.

If the interpreter is not in your PATH

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

Use the Anaconda Prompt

If you have installed the Anaconda Distribution, you can open the Anaconda Prompt from the Anaconda menu under the Start Menu.

Type the full path

On any platform, start **python** by typing the full path to the interpreter (e.g., `C:\python35\python` or `/usr/local/bin/python`)

Add the directory to PATH temporarily

Windows (from a command prompt)

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

Linux/Mac (from a terminal window)

```
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.

Linux/Mac

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 interactive interpreter

- Prompt is **>>>**
- Type any Python statement
- Windows or Gnu-style command line editing
- **Ctrl-D** to exit

The interactive prompt is **>>>**. You can type in any Python command or expression 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 Mac and Linux 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.



The interpreter does autocomplete when you press the TAB key

Table 1. Mac/Linux command line editing

Key binding	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)
3.333333333333335
>>>
```

You don't really need `print()`. If you type an expression (variable or combination of variables, values, and operators), the interpreter will display its value.

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

When you press kbd[ENTER], the interpreter evaluates and prints out whatever you typed in.



If you have **IPython** installed, start `ipython` for a better interactive interpreter (**IPython** is included with Anaconda).

Running Python scripts

- Use Python interpreter
- Same for all platforms

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

```
python myscript.py
```

This will work on any platform.



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 the Python executable to your path.

Using the `help()` function

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 function, module, or package object.

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

len(obj, /)
    Return the number of items in a container.
```

From the Anaconda prompt, Windows command prompt, or Mac/Linux terminal window

Use `pydoc name` to display the documentation for *name*, which can be the name of a function, module, package, or a method or attribute of an object.

```
$ 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.
```



On Windows, open an Anaconda prompt (if available) to make sure that `pydoc` is in the search path.



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

From iPython

iPython makes it easy to get help. Just put a question mark before or after an object, and it will display help.

```
In [1]: len?
```

Signature: len(obj, /)

Docstring: Return the number of items in a container.

Type: builtin_function_or_method

Python Editors and IDEs

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

An IDE (Integrated Development Environment) is the programmer's most important tool other than the language itself. A good IDE can make you more productive, and produce better code.

Visual Studio Code, PyCharm Community Edition, Spyder, and Eclipse are the most full-featured free Python IDEs available. They all have versions for Windows, Mac, and Linux.

There are a couple of pages on the Python Wiki that discuss IDEs:

<http://wiki.python.org/moin/PythonEditors>
<http://wiki.python.org/moin/IntegratedDevelopmentEnvironments>

Some developers use advanced editors such as **Sublime**, **Atom**, or **Notepad++**, but these do not have the extended features of the above IDEs.

For more information

- <https://www.ipython.org>
- <https://www.anaconda.com>
- <https://realpython.com/python-repl/>
- <https://code.visualstudio.com/>
- <https://www.jetbrains.com/pycharm/>

Chapter 1 Exercises

Exercise 1-1 (hello.py)

Use any editor (Notepad, vi, Nano, etc) to write a "Hello, world" python script named **hello.py**.

Run the script from the command line.

Chapter 2: Setting up Visual Studio

Objectives

- Use VS Code to develop Python applications

About Visual Studio Code

Visual Studio Code (AKA "Code" or "VS Code") is a full-featured IDE for programming. It is free, lightweight, and works across common platforms and with many languages.

Visual Studio Code Features

- Autocomplete (AKA IntelliSense™)
- Autoindent
- Syntax checking/highlighting
- Debugging
- git integration
- Code navigation
- Command palette (easily find any command)
- Smart search-and-replace
- Project management
- Split views
- Zen mode (hide UI details)
- Code snippets (macros)
- Variable explorer
- Integrated Python console
- Interpreter configuration
- Unit testing tools
- Keyboard shortcuts
- Many powerful extensions
- Works with many languages
- Free

Getting started

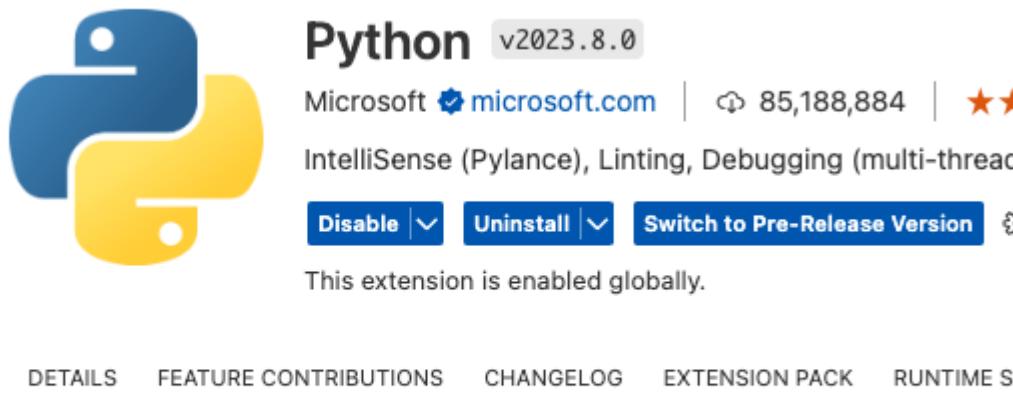
Installing

If VS Code is not already installed, download and install from <https://code.visualstudio.com>.

Adding the Python extension

This class requires the Microsoft extension for working with Python. If it is not installed, please follow these steps:

1. Start VS Code
2. Go to **Extensions** in sidebar on left
3. Search for "python"
4. Select and install the Python extension from Microsoft



Python extension for Visual Studio Code

Configuration

Many aspects of VS Code can be configured. A little configuration makes VS Code more convenient for Python development.

To configure settings, go to **File > Preferences > Settings**.



On Macs, start with the **Code** menu rather than the **File** menu.

Auto save

To turn on Auto Save, which saves your file as you type:

1. Search for "auto save"
2. change **Files: Auto Save** to **afterDelay**.

Launch folder

For convenience, set VS Code to run a script from the folder that contains it. This lets scripts open files relative to the script's location.

1. Search for "execute in"
2. Check the box for **Python > Terminal: Execute in File Dir**

Minimap

If you do not want to use the minimap (the narrow guide strip along the right margin), you can turn it off.

1. Search for "minimap enabled"
2. Uncheck **Editor > Minimap: Enabled**

Editor font size

To change the font size of the code editor window:

1. Search for "editor font size"
2. Set **Editor: Font Size** to desired value

Terminal font size

To change the font size of the terminal window (bottom pane where scripts are executed):

1. Search for "terminal font size"

2. Set **Terminal > Integrated: Font Size** to desired value

Themes

To change the overall theme (colors, font, etc):

1. Go to **File > Preferences > Theme > Color Theme**
2. Choose a new theme



Visual Studio Code may be already setup and configured.

Opening a project

To open a folder, choose **File > Open Folder....** When you open a folder, it automatically becomes a VS Code *workspace*. This means that you can configure settings specific to this folder and any folders under it.

Generally speaking, this is great for managing individual projects.

Creating a Python script

Use the **File** menu

Go to **File > New File....**

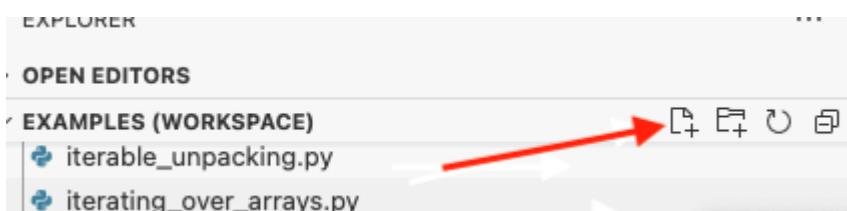
Type a name in the blank, including the `.py` extension. You'll get a file dialog for where to save it. The default location will be the same folder as the currently open file, so be sure to make sure you save it in the desired folder.



You can also just select "Python File" and Code will create a new editor window named `untitled-n`, where *n* is a unique number. You can then use **File > Save** to save the file with a permanent name.

Use the "new file" icon

Click on the "new file" icon next to the folder/workplace name in the Explorer panel.



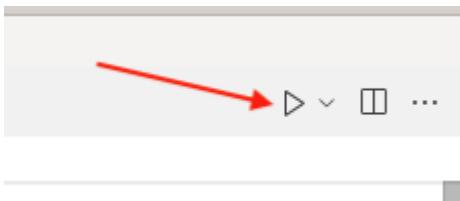
Click on "select a language" and choose Python. As above, it will create a new file named `untitled-n`. Use **File > Save** to save the file.

This will create a file in the same folder as the currently open file.

Running the script

Click the "run" icon

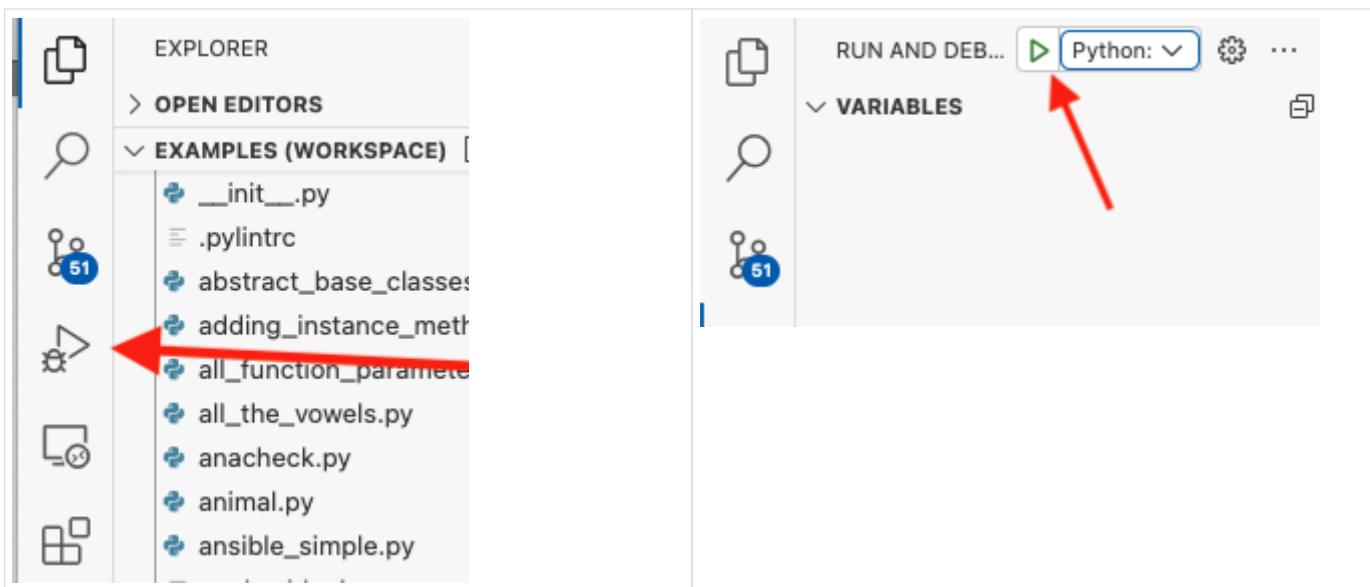
The easiest way to run a script is to click the "Run Python File" icon next to the editor tabs



Once you have run the file, you can re-run it by pressing up-arrow in the terminal window. This makes it easy to add arguments.

Use Run/Debug panel

You can open the Run/Debug panel, and then click on the green arrow to run the script. This uses the default launch configuration, which can be customized.



You can create custom launch configurations to fine-tune how you want to launch each script.

Use **Run** menu

The **Run** menu is essentially a shortcut for the tools on the Run/Debug panel. You can choose to run with or without debugging. If you have no breakpoints set, there is not much difference.

Chapter 2 Exercises

Exercise 2-1 (hellocode.py)

In VS Code, create a new script named `hellocode.py`. Put one or more `print()` calls in it and run it with VS Code

Now run it from the command line as well (not in the VS Code integrated terminal).

Chapter 3: Variables and values

Objectives

- Using variables
- Understanding dynamic typing
- Working with text
- Working with numbers

Using variables

- Variables created when assigned to
- Hold any type of data
- Names case sensitive
- Names can be any length

Variables in Python are created by assigning a value to a name. 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](#) suggests:

`all_lower_case_words_spelled_out_with_underscores`

Example

```
quantity = 5
python_founder = "Guido van Rossum"
final_result = get_result()
program_status = None
```

Keywords and Builtin functions

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

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

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
```

Builtin functions

Any of Python's builtin functions and classes, such as `len()` or `int()` *may* be used as identifiers, but that will overwrite the builtin's functionality, so you shouldn't do that.



Try to avoid `dir`, `file`, `id`, `len`, `max`, `min`, and `sum` as variable names, even though they are tempting, as these are all builtin function names.

Table 2. Python 3.11 Built-in functions

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

*These functions are actually 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`.

Variable names refer to *objects*. Every object has a type and a value. *Everything in Python is an object*.

Certain operations are only valid with appropriate types.



Python does not automatically convert strings to numbers or numbers to strings. There is no concept of "casting" in Python.

Example

`variable_typing.py`

```
a = "123"  
b = 456  
result = a + b # raises error due to incompatible types
```

`variable_typing.py`

```
Traceback (most recent call last):  
  File "/Users/jstrick/curr/courses/python/common/examples/variable_typing.py", line 3,  
    in <module>  
      result = a + b # raises error due to incompatible types  
      ~~~^~~~  
TypeError: can only concatenate str (not "int") to str
```

Strings

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

Python strings (type `str`) are sequences of characters. Strings have full support for Unicode. They can be initialized with several types of string literals. Escape characters, such as `\t` and `\n`, are used for non-printable characters, and are indicated with a backslash. (`\`).

Strings are *immutable* — they can't be modified in place.

While there are multiple delimiters for strings, there is only one string type.

Single-delimited string literals

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

Single-delimited (AKA "single-quoted") 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* (\t) and *newline* (\n) in a string.

Single-delimited strings may not be spread over multiple physical lines. They may not contain a literal new line unless it is escaped.

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

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

- Similar to single-delimited
- Used for multi-line strings
- Can have embedded quote characters
- Used for docstrings

Triple-delimited (AKA "triple quoted") 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 quote characters as well as for documentation and boiler-plate text.

Example

string_literals.py

```
s1 = "spam\n"    # all 4 are the same
s2 = 'spam\n'
s3 = """spam\n"""
s4 = '''spam\n'''

print("Guido's the ex-BDFL of Python")
print()
print('Guido is the ex-"BDFL" of Python')
print()
print("""Guido's the ex-"BDFL" of Python""")
```

string_literals.py

Guido's the ex-BDFL of Python

Guido is the ex-"BDFL" of Python

Guido's the ex-"BDFL" of Python

Raw string literals

- Start with `r`
- Do not interpret backslashes

If a string literal starts with `r` before the opening quotes, 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

`raw_strings.py`

```
regex = r"\w+\s+\w+"
file_path = r"c:\temp"
message = r"please put a newline character (\n) after each line"

print(regex)
print(file_path)
print(message)
```

`raw_strings.py`

```
\w+\s+\w+
c:\temp
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 *codepoint* value in hex
- Use `\uXXXXXXXXX` if codepoint > 0xFFFF

Unicode characters may be embedded in literal strings. Use the Unicode value for the character in the form `\uXXXX`, where `XXXX` is Unicode value in hexadecimal.

For code points above 0xFFFF, use `\UXXXXXXXXX` (note capital "U").

You can also specify the Unicode unique character name using the syntax `\N{name}`. Raw strings do not interpret the `\u`, `\U`, or `\N{}`.



See <http://www.unicode.org/charts> for lists of Unicode characters and their values (codepoints).

Example

unicode.py

```

print('26\u00B0') # Use \uXXXX where XXXX is the Unicode value in hex
print(r'26\u00B0\n') # unicode is not evaluated in raw strings

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

print("\u0928\u092e\u0938\u094d\u0924\u0947\u0020\u0926\u0941\u0928\u093f\u092f\u093e!")
# hindi
print("\u4f60\u597d\u4e16\u754c!") # chinese
print("\u0417\u0434\u0440\u0430\u0432\u0435\u0439\u0020\u0441\u0432\u044f\u0442!") # bulgarian
print("\u00a1\u0048\u006f\u006c\u0061\u0020\u004d\u0075\u006e\u0064\u006f\u0021") # spanish
print("!\u0645\u0631\u062d\u0628\u0627\u0020\u0628\u0627\u0644\u0639\u0627\u0644\u0645") # arabic
print()

data = ['\U0001F95A', '\U0001F414'] # answers the age-old question (at least for Python)
print("unsorted:", data)
print("sorted:", sorted(data))

```

unicode.py

```

26°
26\u00B0\n
we spent €1.23M for an original Cézanne
Romance in F# Major

ହୋମ୍ ମନ୍ଦିର!
ମନ୍ଦିର!
ଶ୍ରୀମତୀ କାମଲାମଣି
ଶ୍ରୀମତୀ କାମଲାମଣି

Hola Mundo!
¡Hola Mundo!

unsorted: [' ', ' ']
sorted: [' ', ' ']

```

Table 3. Escape Sequences

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

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.

String operators

Use the `in` operator to test whether one string is a subset of another: `s1 in s2`

Use `+` (plus sign) to concatenate two strings: `s1 + s2`

String methods

Methods are called from string objects (variables) using *dot notation*: `STR.method()`.

Because strings are immutable, most string functions return a modified copy of the string, or information about the string.

String methods may be chained. You can call a string method on the string returned by another method.

If you need a substring method, that is provided by the `slice` operator, as discussed in the **Array Types** chapter.

String methods may be called on literal strings as well

Builtin functions

Some builtin functions may be passed string objects: `len(STR)`.

Example

strings.py

```
a = "My hovercraft is full of EELS"

print("original:", a)
print("upper:", a.upper())
print("lower:", a.lower())
print("swapcase:", a.swapcase()) # Swap upper and lower case
print("title:", a.title()) # All words are capitalized
print("e count (normal):", a.count('e'))
print("e count (lower-case):", a.lower().count('e')) # Methods can be chained. The next
method is called on the object returned by the previous method.
print("found EELS at:", a.find('EELS'))
print("found WOLVERINES at:", a.find('WOLVERINES')) # Returns -1 if substring not found

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

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 case-less comparisons.
<code>S.center(width[, fillchar])</code>	Return S centered in a string of length <i>width</i> . 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 <i>sub</i> . Optional arguments <i>start</i> and <i>end</i> 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 `UnicodeEncodeError`'s.
<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 find() but raise ValueError 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.
<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.

Method	Description
<code>S.isprintable()</code>	Return True if all characters in S are considered printable in repr() 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 title-cased string and there is at least one character in S, i.e. upper- and title-case 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 join() 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.
<code>S.partition(sep)</code>	Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
<code>S.removeprefix(prefix)</code>	Return copy of S with prefix removed from beginning. If the prefix is not found, return S.
<code>S.removesuffix(suffix)</code>	Return copy of S with suffix removed from end. If the suffix is not found, return S.
<code>S.replace(old, new[, count])</code>	Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.
<code>S.rfind(sub[, start[, end]])</code>	Return the highest 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. Return -1 on failure.
<code>S.rindex(sub[, start[, end]])</code>	Like rfind() but raise ValueError when the substring is not found.
<code>S.rjust(width[, fillchar])</code>	Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space).
<code>S.rpartition(sep)</code>	Search for the separator sep in S, starting at the end of S, 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 the separator.

Method	Description
<code>S.rsplit(sep=None, maxsplit=-1)</code>	Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.
<code>S.rstrip([chars])</code>	Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead.
<code>S.split(sep=None, maxsplit=1)</code>	Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, 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 S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
<code>S.startswith(prefix[, start[, end]])</code>	Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.
<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

- Two main numeric types
 - Integers
 - Floats
- Integer literals can be decimal, octal, or hexadecimal
- Floats can be traditional or scientific notation

Python provides the `int` and `float` types for storing numbers, and for numeric calculations. Both types may be intermixed in expressions.

Integers

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

Floats

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



Python also supports the `bool` and `complex` numeric types, but they are not used in everyday data or math calculations.

Example

numeric.py

```
a = 5
b = 10
c = 20.22
d = 0o123      # Octal
e = 0xdeadbeef # Hex
f = 0b10011101 # Binary

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)
```

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 may look at some libraries with extended math functionality.

Most of the operators should look familiar.

Division

Division (/) always returns a float result.

Assignment-with-operation

Python supports *assignment-with-operation*. For instance, `x += 5` adds 5 to variable `x`. This works for nearly any operator, using the following 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. If both operands are ints, the result is an integer; otherwise, the result is a whole number float.

Order of operations

Please **E**x~~cuse~~ **M**y **D**ear **A**unt **S**ally!

The order of operations is parentheses, exponents, multiplication or division, addition or subtraction, and left-to-right within the same type. It's best practice to use parentheses for readability.

Example

math_operators.py

```
a = 23
b = 7

print(a + b, a - b, a * b) # normal operations

print(a / b, a // b, a / -b, a // -b) # division and floored division

print(a ** b) # exponentiation

print(a % b) # modulus (remainder)

x = 22
x += 10 # Same as x = x + 10
print(f"x: {x}")
```

math_operators.py

```
30 16 161
3.2857142857142856 3 -3.2857142857142856 -4
3404825447
2
x: 32
```



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
<code>x + y</code>	sum of x and y
<code>x - y</code>	difference of x and y
<code>x * y</code>	product of x and y
<code>x / y</code>	quotient of x and y
<code>x // y</code>	(floored) quotient of x and y
<code>x % y</code>	remainder of x / y
<code>-x</code>	x negated
<code>+x</code>	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> <code>x ** y</code>	x raised to the power y

Converting among types

- No automatic conversion between numbers and strings
- Builtin type constructors

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 is no automatic conversion between types.

However, there are built-in *constructors* to do these conversions. Use `int(s)` to convert string `s` to an integer. Use `str(x)` to convert anything to a string, and so forth.

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.



these "conversion functions" are really builtin classes. When you "convert" a value, you are creating a new instance of the specified class, initialized with the old value.

Table 6. Builtin type constructors

constructor	converts...
<code>str()</code>	any object
<code>bool()</code>	any object
<code>int()</code> <code>float()</code> <code>bool()</code> <code>complex()</code>	any number, or any string that looks like the appropriate number.
<code>list()</code> <code>tuple()</code> <code>set()</code> <code>frozenset()</code>	any iterable
<code>dict()</code>	any iterable of valid key/value pairs
<code>bytes()</code>	any string, or any iterable of ints which are all in the range 0-255.

Example

converting_types.py

```
a = "123"
b = 456
result1 = a + str(b) # convert int to str and concatenate
print(f"result1: {result1}")

result2 = int(a) + b # convert str to int and add
print(f"result2: {result2}")
```

converting_types.py

```
result1: 123456
result2: 579
```

Chapter 3 Exercises

Exercise 3-1 (string_fun.py)

Start with the file `string_fun.py`, which is provided in the top folder of the student files. The variable `name` is set to "john jacob jingleheimer schmidt". Using that variable,

- Print `name` as-is
- Print `name` in upper case
- Print `name` in title case
- Print number of occurrences of 'j' in `name`
- Print length of `name`
- Print position (offset) of "jacob" in `name`; in other words, what character location (starting at 0) is the 'j' of 'jacob'

Chapter 4: Basic input and output

Objectives

- Writing output to the screen
- Formatting output
- Getting command line parameters
- Reading keyboard input

Writing to the screen

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

To output text to the screen, use the `print()` function. It takes a list of one or more arguments, converts each to a string, 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

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

print("Hello,", end=' ') # Print space instead of newline at the end
print("world")
print("#-----")

print("Hello,", end=' ')
print("world", end='!') # Print bang instead of newline at end
print("#-----")

x = "Hello"
y = "world"

print(x, y) # Item separator is space instead of comma
print("#-----")

print(x, y, sep=', ') # Item separator is comma + space
print("#-----")

print(x, y, sep='') # Item separator is empty string
print("#-----")
```

print_examples.py

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

Format strings

- Add `f` in front of literal strings
- Added in version 3.6
- Format: `{expression:formatting codes}`

The best way to format text in Python is *format strings*, better known as *f-strings*. An f-string is a literal string that starts with an `f` in front of the opening quotes. It can contain both literal text and *expressions*, which can be any normal Python expression.

Each expression is put inside curly braces (`{}`). This way, the value to be formatted is right where it will be in the resulting string. The expression can be any variable, as well as an expression or function call, such as `x + y` or `spam()`.

```
name = "Anna Karenina"  
city = "Moscow"  
  
s = f"{name} lives in {city}"
```

By default, the expression is converted to a string. You can add a colon and formatting codes to fine-tune how it is formatted.

```
result = 22 / 7  
print(f"result is {result:.2f}")
```

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

See [String Formatting](#) for details on formatting.



Format strings are fast and flexible, and a big improvement over `STR.format()`, which is described in the next section. They are evaluated at run time, so they are not constants. See file `f_strings_fun.py` for other examples.



For more details, check out the PyDoc topic FORMATTING, or [section 6.1.3.1](#) of The Python Standard Library documentation, the **Format Specification Mini-Language**.

Example

f_strings.py

```
city = 'Orlando'  
temp = 85  
count = 5  
avg = 3.4563892382  
  
print(f"It is {temp}\u00b0 in {city}") # variables inserted into string  
  
# .2f means round a float to 2 decimal points  
print(f"count is {count:03d} avg is {avg:.2f}")  
  
print(f"2 + 2 is {2 + 2}") # any expression is OK
```

f_strings.py

```
It is 85° in Orlando  
count is 005 avg is 3.46  
2 + 2 is 4
```

Example

f_strings_fun.py

```
# fun with strings
name = "Guido"

print(f"name: {name}")
# < left justify (default for non-numbers), 10 is field width, s formats a string
print(f"name: [{name:<10s}]")
# > right justify
print(f"name: [{name:>10s}]")
# >. right justify and pad with dots
print(f"name: [{name:.>10s}]")
# ^ center
print(f"name: [{name:^10s}]")
# ^ center and pad with dashes
print(f"name: [{name:-^10s}]")
print()

# fun with integers
value = 2093
print(f"value: {value}")
print(f"value: [{value:10d}]") # pad with spaces to width 10
print(f"value: [{value:010d}]") # pad with zeroes to width 10
print(f"value: {value:d} {value:b} {value:x} {value:o}") # d is decimal, b is binary, o is octal, x is hex
print(f"value: {value} {value:#b} {value:#x} {value:#o}") # add prefixes
print()

result = 38293892
print(f"result: ${result:,d}") # , adds commas to numeric value
print()

# fun with floats
amount = .325482039
print(f"amount: {amount}")
print(f"amount: {amount:.2f}") # round to 2 decimal places
print(f"amount: {amount:.2%}") # convert to percent
print()

# fun with functions
print(f"length of 'name': {len(name)}") # function call OK
```

f_strings_fun.py

```
name: Guido
name: [Guido      ]
name: [      Guido]
name: [.....Guido]
name: [  Guido   ]
name: [--Guido--]

value: 2093
value: [      2093]
value: [0000002093]
value: 2093 100000101101 82d 4055
value: 2093 0b100000101101 0x82d 0o4055

result: $38,293,892

amount: 0.325482039
amount: 0.33
amount: 32.55%

length of 'name': 5
```

Using the `.format()` method

- Expressions passed via `.format()`
- Same rules as `string.format()`

Prior to version 3.6, the best tool for formatted output was the `.format()` method on strings. This is very similar to f-strings, except that the expressions to be formatted are passed to the `.format()` method. Curly braces are used as before, but they are left empty. The first argument to `.format()` goes in the first pair of braces, etc.

Formatting codes still go after a colon.

There are many more ways of using `format()`; these are just some of the basics.



`.format()` is very useful for reusing the same format in multiple `print()` statements.

Example

`string_formatting.py`

```
city = 'Orlando'
temp = 85
count = 5
avg = 3.4563892382

print("It is {} \u00b0 in {}".format(temp, city)) # variables inserted into string

# .2f means round a float to 2 decimal points
print("count is {:03d} avg is {:.2f}".format(count, avg))

print("2 + 2 is {}".format(2 + 2)) # any expression is OK
```

`string_formatting.py`

```
It is 85° in Orlando
count is 005 avg is 3.46
2 + 2 is 4
```

Legacy String Formatting

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

Prior to Python 2.6, when the `.format()` method was added to strings, the % symbol was used as a format operator. Like the more modern versions, this operator returns a string based on filling in a format string with values.

```
%flagW.Ptype
```

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

The placeholders are similar to those used in the C `printf()` function. They are specified with a percent symbol (%), rather than braces.

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

Table 7. Legacy formatting types

placeholder	data type
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 8. Legacy formatting flags

flag	description
-	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

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

print("Name is [%-10s]" % name)    # Dash means left justify string
print("Name is [%10s]" % name)    # Right justify (default)
print("count is %03d avg is %.2f" % (count, avg)) # Argument to % is either a single
variable or a tuple

print("info is %d %o %x" % (info, info, info))      # Arguments must be repeated to be
used more than once
print("info is %d %o %x" % ((info,) * 3))      # Obscure way of doing the same thing Note:
(x,) is singleton tuple

print("info is %d %#0o %#x" % (info, info, info))    # # means add 0x, 0o, etc.
```

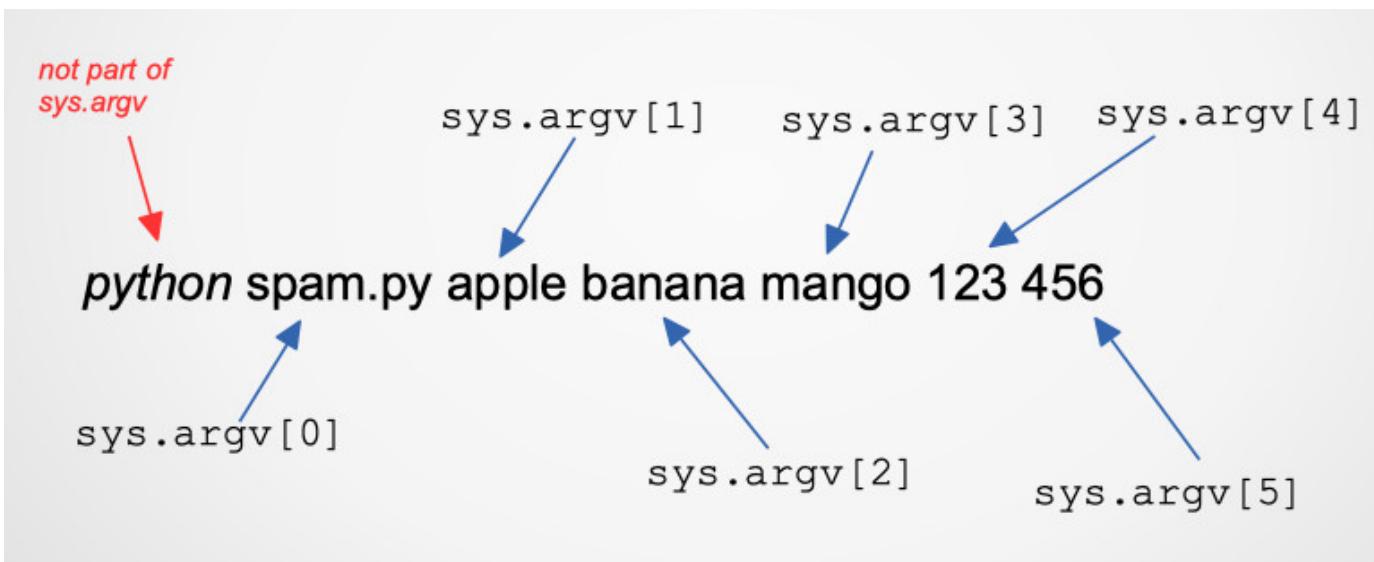
string_formatting_legacy.py

```
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

- `argv` list in `sys` module
- `sys` must be imported
- Element 0 is 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 actual argument to your script.



Example

sys_argv.py

```
import sys

print(f"sys.argv: {sys.argv}\n")

animal = sys.argv[1] # First command line parameter
print(f"animal: {animal}")
```

sys_argv.py wombat

```
sys.argv: ['/Users/jstrick/curr/courses/python/common/examples/sys_argv.py', 'wombat']
animal: wombat
```



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 that will be displayed on the console. `input()` returns the text that was entered as a string.

Use `int()` or `float()` to convert the input to an integer or a float as needed.



When 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.

Example

`keyboard_input.py`

```
user_name = input("What is your name: ")
quest = input("What is your quest? ")
print(f"{user_name} seeks {quest}")

raw_num = input("Enter number: ") # input is always a string
num = float(raw_num) # convert to numbers as needed

print(f"2 times {num} is {num * 2}")
```

`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 5.0 is 10.0
```

Chapter 4 Exercises

Exercise 4-1 (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 your IDE)

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 4-2 (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 your IDE)

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

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

Table 9. Temperature Scales

	0	100
Fahrenheit	Really cold	Really hot
Celsius	Cold	Fatal
Kelvin	Fatal	Fatal

Chapter 5: 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

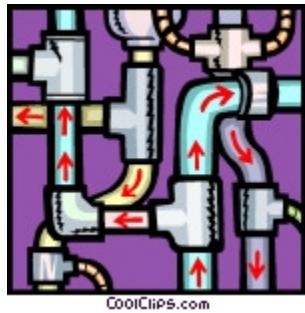
- Controls order of execution
- Conditionals and loops
- Uses Boolean logic

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`.



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.



if and elif

- Basic conditional statement is `if`
- `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 builtin objects `True` and `False` are objects of type `bool`, predefined to have values of 1 and 0, respectively.

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.

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



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

Don't say

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



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

```
if x:
```

Table 10. True and False

False	True
<code>False</code>	<code>True</code>
<code>None</code>	<i>Everything else</i>
<code>0</code>	
<code>""</code> (empty string)	
<code>[]</code> (empty list)	
<code>{}</code> (empty dictionary)	
<code>()</code> (empty tuple)	
<i>Any other empty container</i> [*]	

^{*} Any object that has a length, and the length is 0

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.

After a line introducing a block structure (`if` statement, `for/while` loop, function definition, or class definition), the next line must be indented. All statements indented the same amount 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 current block.

The standard indentation for Python scripts is 4 spaces.



While tempting, indenting with tabs can cause formatting problems. Most Python-aware editors insert 4 spaces when you press the `TAB` key

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
```

The Conditional Expression

- Used for simple if-then-else conditions
- Can be used inline

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.

```
value = expr1 if condition else expr2
```

This is a shortcut for

```
if condition:  
    value = expr1  
else:  
    value = expr2
```

Sometimes this is useful for inline code, as for passing arguments to a callable (function or class constructor).

Note: In C-like languages, this concept is implemented as the *ternary operator*, A?B:C, where if A is true, use value b, else use value C. The Python equivalent is B if A else C.

Example

```
print(long_message if DEBUGGING else short_message)  
  
audience = 'j' if is_juvenile(curr_book_rec) else 'a'  
  
file_mode = 'a' if APPEND_MODE else 'w'
```

Relational Operators

- Compare two objects
- Overloaded for different types of data
- Numbers cannot be compared to 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**.

`==` `!=` `<` `>` `>=` `<=`



Strings and numbers cannot be compared using any of the greater-than or less-than operators. No string is equal to any number.



A seventh operator, `is`, returns **True** only if two names refer to the same object.

Example

if_else.py

```
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:           # elif is short for "else if", and always requires an
expression to check
    print("Good afternoon")
elif hour < 23:
    print("Good evening")
else:
    print("You're up late")
```

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 11. 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. Unlike many languages, reading a file in Python uses a `for` loop.

Loop control

- `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`

```
print("Welcome to ticket sales\n")

while True: # Loop "forever"
    raw_quantity = input("Enter quantity to purchase (or q to quit): ")
    if raw_quantity == '':
        continue # Skip rest of loop; start back at top
    if raw_quantity.lower() == 'q':
        print("goodbye!")
        break # Exit loop

    quantity = int(raw_quantity) # could validate via try/except
    print(f"Sending {quantity} ticket(s)")
```

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):
```

```
Enter quantity to purchase (or q to quit): 2
sending 2 ticket(s)
```

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

Chapter 5 Exercises

Exercise 5-1 (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".



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

Exercise 5-2

Part A (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.

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



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).

Part B (guessx.py)

Redo **guess.py** to get the maximum number from the command line *or* prompt the user to input the maximum, or both (if no value on command lines then prompt).

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 6: 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, known as "containers". 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. A **list** is an array of objects. A **tuple** is an array of objects.

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

```
fruits = ['apple', 'cherry', 'orange', 'kiwi', 'banana', 'pear', 'fig'] # list
name = "Eric Idle" # str
knight = 'King', 'Arthur', 'Britain' # tuple
data = b"abcde" # bytes

print(fruits[3]) # print 4th element of fruits
print(name[2]) # print 3rd letter of name
print(knight[1]) # print 2nd element of knight
print(data[4]) # print 5th element of data
```

typical_arrays.py

```
kiwi
i
Arthur
101
```

Lists

- Array of objects
- Create with `list()` or `[]`
- 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` object with the `list()` class or a pair of square brackets. A list can be initialized with a comma-separated list of values or any *iterable container*, such as a `tuple` or `string`.

Objects like `list` which contain multiple values are referred to as *container types*, or just *containers*, and sometimes as *collections*.

Table 12. List Methods and Keywords (note L represents a list)

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

Example

using_lists.py

```
cities = ['Portland', 'Pittsburg', 'Peoria']
print(f"cities: {cities}\n")

cities.append('Miami')
cities.append('Montgomery')
print(f"cities: {cities}\n")

cities.insert(0, 'Boston')
cities.insert(5, "Buffalo")
print(f"cities: {cities}\n")

more_cities = ["Detroit", "Des Moines"]
cities.extend(more_cities)
print(f"cities: {cities}\n")

del cities[3]
print(f"cities: {cities}\n")

cities.remove('Buffalo')
print(f"cities: {cities}\n")

city = cities.pop()
print(f"city: {city}")
print(f"cities: {cities}\n")

city = cities.pop(3)
print(f"city: {city}")
print(f"cities: {cities}\n")
```

using_lists.py

```
cities: ['Portland', 'Pittsburg', 'Peoria']

cities: ['Portland', 'Pittsburg', 'Peoria', 'Miami', 'Montgomery']

cities: ['Boston', 'Portland', 'Pittsburg', 'Peoria', 'Miami', 'Buffalo', 'Montgomery']

cities: ['Boston', 'Portland', 'Pittsburg', 'Peoria', 'Miami', 'Buffalo', 'Montgomery',
'Detroit', 'Des Moines']

cities: ['Boston', 'Portland', 'Pittsburg', 'Miami', 'Buffalo', 'Montgomery', 'Detroit',
'Des Moines']

cities: ['Boston', 'Portland', 'Pittsburg', 'Miami', 'Montgomery', 'Detroit', 'Des
Moines']

city: Des Moines
cities: ['Boston', 'Portland', 'Pittsburg', 'Miami', 'Montgomery', 'Detroit']

city: Miami
cities: ['Boston', 'Portland', 'Pittsburg', 'Montgomery', 'Detroit']
```

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. A negative index counts from the end — `x[-1]` is the last element of `x`.

```
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-at:stop-before:count-by]
```

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

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

and **start** is incremented by **step**



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

Example

indexing_and_slicing.py

```
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]) # First element
print("pythons[5]", pythons[5]) # Sixth element
print("pythons[0:3]", pythons[0:3]) # First 3 elements
print("pythons[2:]", pythons[2:]) # Third element through the end
print("pythons[:2]", pythons[:2]) # First 2 elements
print("pythons[1:-1]", pythons[1:-1]) # Second through next-to-last element
print("pythons[0::2]", pythons[0::2]) # Every other element, starting with first
print("pythons[1::2]", pythons[1::2]) # Every other element, starting with second

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]) # Last element
print("phrase[21:25]", phrase[21:25])
print("phrase[21:]", phrase[21:])
print("phrase[:10]", phrase[:10])
print("phrase[::2]", phrase[::2])
```

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

```
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.



The loop variable retains the last value it was set to in the loop

Example

`iterating_over_arrays.py`

```
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: # Iterate over elements of list
    print(p)
print()

for r in my_tuple: # Iterate over elements of tuple
    print(r)
print()

for ch in my_str: # Iterate over characters of string
    print(ch, end=' ')
print()
```

`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
```

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 container 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 = ()
```



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

Example

creating_tuples.py

```
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)
```

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 iterable
- 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.

Example

iterable_unpacking.py

```
values = ['a', 'b', 'c']

x, y, z = values # unpack values (which is an iterable) into individual variables

print(x, y, z)
print()

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

for row in people:
    first_name, last_name, _ = row # unpack row into variables
    print(first_name, last_name)
print()

for first_name, last_name, _ in people: # a for loop unpacks if there is more than one variable
    print(first_name, last_name)
print()
```

iterable_unpacking.py

```
a b c
```

```
Bill Gates  
Steve Jobs  
Paul Allen  
Larry Ellison  
Mark Zuckerberg  
Sergey Brin  
Larry Page  
Linux Torvalds
```

```
Bill Gates  
Steve Jobs  
Paul Allen  
Larry Ellison  
Mark Zuckerberg  
Sergey Brin  
Larry Page  
Linux Torvalds
```

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

```
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: # person is a tuple
    print(person[0], person[1])
print('-' * 60)

for person in people:
    first_name, last_name, product = person # unpack person into variables
    print(first_name, last_name)
print('-' * 60)

for first_name, last_name, product in people: # if there is more than one variable in a
for loop, each element is unpacked
    print(first_name, last_name)
print('-' * 60)
```

Operators and keywords for sequences

- Operators `+` (plus) `*` (splat)
- 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

```
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)) # Test for membership in list
print("pink in colors: ", ("pink" in colors))

print("colors: ", ",".join(colors)) # Concatenate iterable using ", " as delimiter

del colors[4] # remove brown

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

colors.remove('green') # Remove element by value

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

sum_of_lists = [True] + [True] + [False] # Add 3 lists together; combines all elements

print("sum of lists:", sum_of_lists)

product = [True] * 5 # Multiply a list; replicates elements

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

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]
```

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)
```

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

len()

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

min(), max(), sorted()

`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`.



`min()`, `max()`, and `sorted()` accept a named parameter `key`, which specifies a key function for converting each element 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()

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

Example

sequence_functions.py

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

print(f"colors: len is {len(colors)}; min is {min(colors)}; max is {max(colors)}")
print(f"months: len is {len(months)}; min is {min(months)}; max is {max(months)}")
print()

print("sorted:", end=' ')
for m in sorted(colors):  # sorted() returns a sorted list
    print(m, end=' ')
print()
```

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
```

Iterators

- Can be iterated over
- Do not contain data
- Do not have a length
- Cannot be indexed or sliced

Some sequence functions return *iterators* — objects that provide a virtual sequence of values. The only operations possible on an iterator are to loop over it or to get just the next value. Iterators can not be indexed, do not have a length, and do not contain data.

`enumerate()`

`enumerate()` returns an **enumerate object** — this is an iterator that provides the index of each element of a sequence, along with the value of the element.

When you iterate through `[x, y, z]` with `enumerate()`, you get a sequence 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.

`reversed()`

`reversed(s)` returns an iterator that loops through `s` in reverse order.

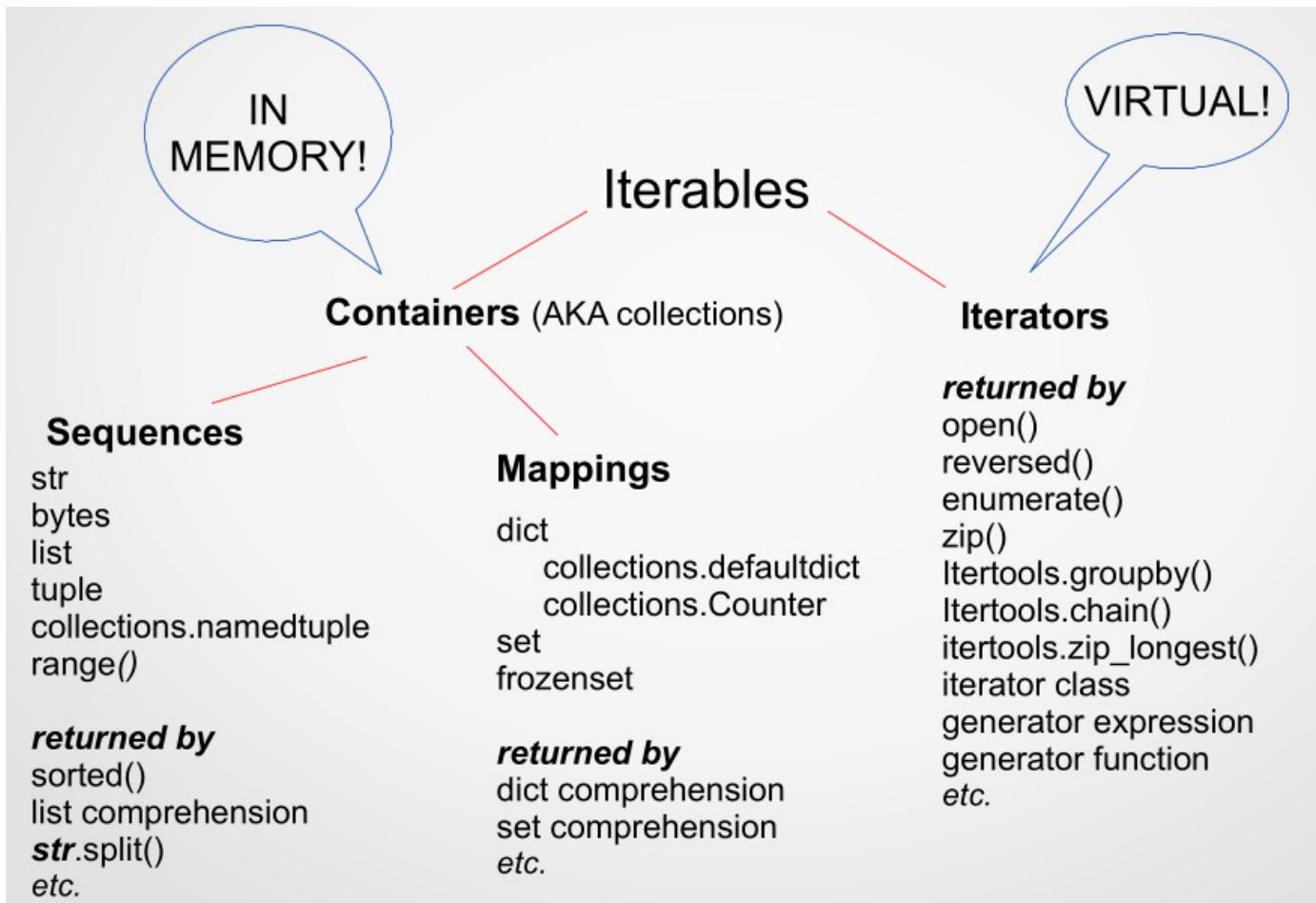
`zip()`

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

`range()`

`range()` returns a **range object**, that provides a sequence of integers. 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.



Examples

enumerate.py

```
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): # enumerate() returns iterable of (index, value) tuples
    print(i, color)

print()

for num, month in enumerate(months, 1): # Second parameter to enumerate is added to index
    print(f"{num} {month}")
```

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
```

iterators.py

```
phrase = ('dog', 'bites', 'man')
print("".join(reversed(phrase))) # reversed() returns a reversed iterator
print()

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

full_names = zip(first_names, last_names) # zip() returns an iterator of tuples created
# from corresponding elements
print("full_names:", full_names)
print()

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

iterators.py

```
man bites dog

full_names: <zip object at 0x101f565c0>

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

using_ranges.py

```
print("range(1, 6): ", end=' ')
for x in range(1, 6): # Start=1, Stop=6 (1 through 5)
    print(x, end=' ')
print()

print("range(6): ", end=' ')
for x in range(6): # Start=0, Stop=6 (0 through 5)
    print(x, end=' ')
print()

print("range(3, 12): ", end=' ')
for x in range(3, 12): # Start=3, Stop=12 (3 through 11)
    print(x, end=' ')
print()

print("range(5, 30, 5): ", end=' ')
for x in range(5, 30, 5): # Start=5, Stop=30, Step=5 (5 through 25 by 5)
    print(x, end=' ')
print()

print("range(10, 0, -1): ", end=' ')
for x in range(10, 0, -1): # Start=10, Stop=1, Step=-1 (10 through 1 by 1)
    print(x, end=' ')
print()
```

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
- 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

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

ufruits = [fruit.upper() for fruit in fruits]          # Simple transformation of all elements
afruits = [fruit.title() for fruit in fruits if fruit.startswith('a')]  # Transformation of selected elements only

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]    # Any kind of data is OK

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

nums = [x for x in values if isinstance(x, int)]    # Select only integers from list
print(nums, '\n')

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

clean = [d.strip().lower() for d in dirty_strings]
for c in clean:
    print(f">>{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]    # More than one for is OK

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

list_comprehensions.py

```
uftuits: ['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
```

...

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

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

ufruits = (fruit.upper() for fruit in fruits) # These are all exactly like the list comprehension example, but return generators rather than lists
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(f">{c}<", end=' ')
print("\n")

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

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 6 Exercises

Exercise 6-1 (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.



For exercises 6-2 and 6-3, 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 6-2 (cicadas.py)

Periodical cicadas are synchronized to emerge en masse, every 13 or 17 years.

Starting with a 17-year-cycle brood that emerges in 1978, print out all the years the brood will emerge through 2050.

Exercise 6-3 (sequences.py)

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

Exercise 6-4 (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 6-5 (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 nested 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 value is False. This will skip the non-prime numbers.



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



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 7: 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.

The file object is actually an instance of `_io.TextIOWrapper`, but in this chapter we'll just refer to it as a "file object".



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 file name and (optional) mode
- Returns file object
- Mode can be read, write, or append
- `with` block
 - Automagically closes file object
 - Not specific to file objects

Open a text file with the builtin `open()` function. Arguments are the file name, which may be specified as a relative or absolute path, and the mode.

For reading, you can skip the mode, since "`r`" is the default.

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

Since Windows uses backslash as a folder separator, and backslash is a special character in literal strings, use forward slashes on all platforms when putting a file path in literal text, and Python will open the file correctly. This will also keep your code platform-neutral.

Examples

```
with open("mary.txt") as mary_in:           # open for reading in text mode (default)
    pass
with open("mary.txt","r") as mary_in:         # open for reading in text mode
    pass
with open("roland.txt","w") as roland_out:   # open for writing in text mode
    pass
with open("roland.txt","x") as roland_out:   # open for writing in text mode, fail if
file exists
    pass
with open("spam.cfg","a") as cfg_out:        # open for append in text mode
    pass
with open("wombat.png","rb") as wombat_in:   # open for reading in binary (raw) mode
    pass
with open("wombat.png","wb") as wombat_out:  # open for writing in binary mode
    pass
```



Adding `_in` or `_out` to the file object is not required, but makes it easy to tell what variables are file objects, and whether they are open for reading or writing.

Table 13. File Modes

Mode	Open for	Alternate forms
text mode		
"r"	reading	"rt" (or omit)
"w"	writing	"wt"
"a"	appending	"at"
"x"	writing (fail if file exists)	"xt"
"r+t"	reading and writing (does not truncate existing data)	"r+t"
"w+t"	reading and writing (truncates existing data)	"w+t"
binary (raw) mode		
"rb"	reading	
"wb"	writing	
"ab"	appending	
"xb"	writing (fail if file exists)	
"r+b"	reading and writing (does not truncate existing data)	
"w+b"	reading and writing writing (truncates existing data)	

Reading a text file

Read file line by line

```
with open("file_path") as file_in:  
    for raw_line in file_in:  
        line = raw_line.rstrip() # trim \n from end  
        # use trimmed line here ...
```

This is the easiest and probably the most common way to read a file. It is safe to use on even huge files, because only one line is in memory at a time. Each line has the newline character on the end, which you can easily remove with `.rstrip()`.

Read entire file into one string

```
with open("file_path") as file_in:  
    contents = file_in.read()
```

Reads the entire file into a single string. This is useful for search and replace, or other kinds of whole-file parsing.

Read part of file into one string (n is number of characters to read)

```
with open("file_path") as file_in:  
    part_contents = file_in.read(n)
```

Same as previous, except only reading part of the file. You can specify where to start reading with `file_object.seek(position)`.

Read entire file into list of lines

```
with open("file_path") as file_in:  
    all_lines_with_nl = file_in.readlines()
```

Read the entire file into a list of strings. Each string contains one line, with its newline character. This is useful for sorting files, or inserting and replacing lines.

Read entire file into list of lines (without newlines)

```
with open("file_path") as file_in:  
    all_lines_without_nl = file_in.read().splitlines()
```

Read the entire file into a list of strings without their newlines. This is useful for using the file as data.

Read next line

```
with open("file_path") as file_in:  
    raw_line = file_in.readline()
```

Read the next line in the file. This is generally not needed unless you really want to read just one line.

Example

reading_files.py

```
FILE_NAME = '../DATA/mary.txt'

mary_in = open(FILE_NAME) # open file for reading
# read file...
mary_in.close() # close file (easy to forget to do this!)

with open(FILE_NAME) as mary_in: # open file for reading
    for raw_line in mary_in: # iterate over lines in file (line retains \n)
        line = raw_line.rstrip() # rstrip('') removes whitespace (including \n or \r )
from end of string
        print(line)
print('-' * 60)

with open(FILE_NAME) as mary_in:
    contents = mary_in.read() # read entire file into one string
    print("NORMAL:")
    print(contents)
    print("=" * 20)
    print("RAW:")
    print(repr(contents)) # print string in "raw" mode
print('-' * 60)

with open(FILE_NAME) as mary_in:
    lines_with_nl = mary_in.readlines() # readlines() reads all lines into an array
    print(lines_with_nl)
print('-' * 60)

with open(FILE_NAME) as mary_in:
    lines_without_nl = mary_in.read().splitlines() # splitlines() splits string on ''
into lines
    print(lines_without_nl)
```

reading_files.py

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

NORMAL:

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

=====

RAW:

```
'Mary had a little lamb,\nIts fleece was white as snow,\nAnd everywhere that Mary  
went\nThe lamb was sure to go\n'
```

```
['Mary had a little lamb,\n', 'Its fleece was white as snow,\n', 'And everywhere that  
Mary went\n', 'The lamb was sure to go\n']
```

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

Processing files from the command line

- `sys.argv[1:]` file names on command line

To iterate over one or more files specified on the command line, import the `sys` module and iterate over `sys.argv[1:]`. Since `sys.argv[0]` is the script name, this will be the list of all words on the command line.

If you need to use the first argument separately from the command line, you can use `.pop(1)` to grab the first argument and remove it from `sys.argv`.



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.

Example

`looping_over_files.py`

```
import sys

for file_path in sys.argv[1:]: # skip script name
    print(f"Processing {file_path}")
    with open(file_path) as file_in:
        pass # read each file here
```

`looping_over_files.py/DATA/alice.txt/DATA/mary.txt`

```
Processing ..../DATA/alice.txt
Processing ..../DATA/mary.txt
```

Example

looping_over_files_using_first_arg.py

```
import sys

first_arg = sys.argv.pop(1)

print(f"first arg is '{first_arg}'")
for file_path in sys.argv[1:]: # skip script name
    print(f"Processing {file_path}")
    with open(file_path) as file_in:
        pass # read each file here
```

looping_over_files_using_first_arg.py spam ../DATA/alice.txt ../DATA/mary.txt

```
first arg is 'spam'
Processing ../DATA/alice.txt
Processing ../DATA/mary.txt
```

Writing to a text file

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

To write to a text file, open the file in write mode ("`w`"). Then use the `.write()` method to write a single string or `.writelines()` to write a list of strings.

Neither method will add newline characters, so make sure the items you're writing have them as needed.

Opening a file in write mode will delete any previous contents.

Example

`write_file.py`

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

with open("states.txt", "w") as states_out: # "w" opens for writing, "a" for append
    for state in states:
        states_out.write(state + "\n") # write() does not automatically add newline
```

`states.txt`

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



`.writelines()` probably should have been called `.writestrings()`

Modifying text files

- Open original file for reading
- Open new (temporary) file for writing
- Read from original, write to new
- Rename old to backup
- Rename new to old

To modify a text file, open the original file for reading, then open a new file for writing. While reading from the original file, make changes as needed and write to the new file.

When finished, you can rename the original file as a backup, then rename the new file to the original name. This may or may not be needed.

Example

`modify_text_file.py`

```
import os
file_name = 'states.txt'
temp_name = "temp.txt"

with open(file_name) as file_in:
    with open(temp_name, "w") as file_out:
        for line in file_in:
            new_line = line.upper()
            file_out.write(new_line)

os.rename(file_name, file_name + '.BAK')
os.rename(temp_name, file_name)
```

Table 14. File Object Methods

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

Chapter 7 Exercises

Exercise 7-1 (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.



Use `enumerate()`.

Test with the following commands (or run from the IDE):

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

Test with other files, as desired

Exercise 7-2 (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`.

Manually compare the original to the two new files to make sure the data went to the right place.

Exercise 7-3 (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).



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

Exercise 7-4 (icount_words.py)

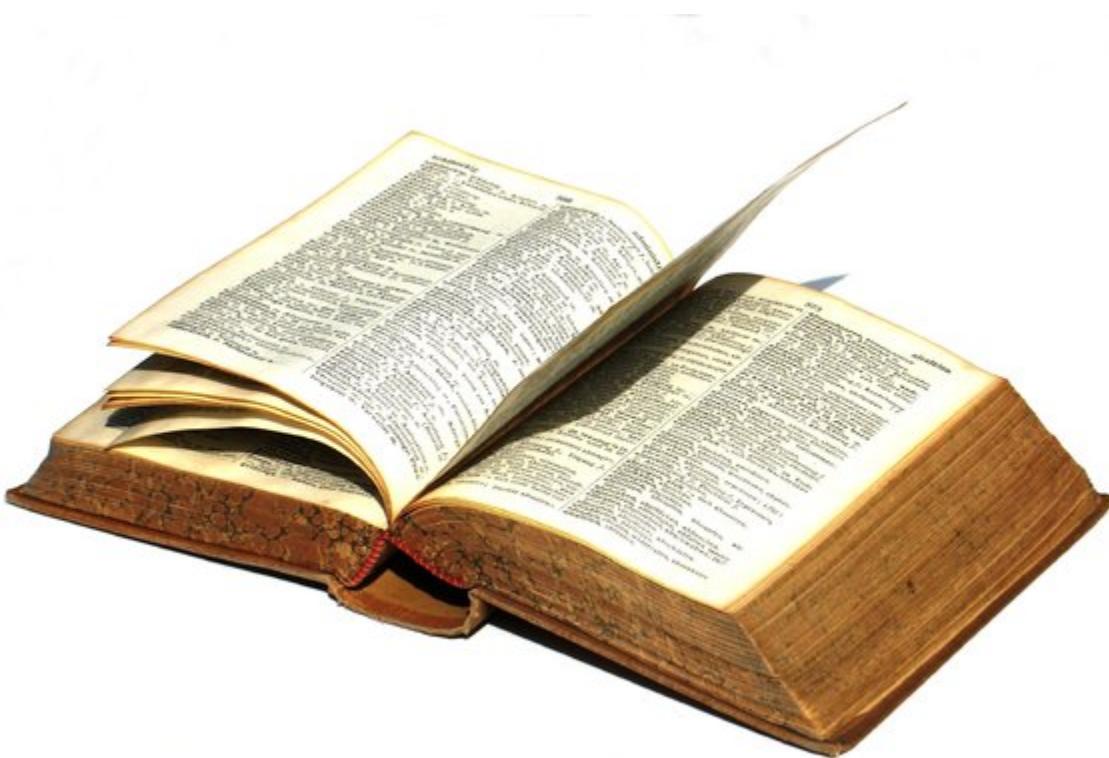
FOR ADVANCED STUDENTS

Modify `count_words.py` to make the search case-insensitive.

Chapter 8: Dictionaries and Sets

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 of key/value pairs
- Called "hashes", "hash tables" or "associative arrays" in other languages
- Rich set of functions available
- When to use?
 - Mapping
 - Counting

A dictionary is a collection (AKA container) 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, sets, and other dictionaries may not be used as keys. Any immutable type may be a key, although typically keys are strings.

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

If you iterate over `dictionary.items()`, `.keys()`, or `.values()`, it will iterate in the order that the elements were added.

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 are the things being counted, and the values are the counts for each thing.

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



Prior to version 3.6, the elements of a dictionary were in indeterminate order.

Creating dictionaries

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

To create a dictionary, use the `dict()` constructor or a literal dictionary `{}`. The dictionary can be created empty, or you can initialize it.

The `dict()` constructor

Initialize the `dict()` constructor with an iterable of key/value pairs, or named parameters.

```
d1 = dict() # empty dict
kv_pairs = [('NC', 'Raleigh'), ('NY', 'Albany'), ('NJ': 'Trenton')]
state_capitals = dict(kv_pairs) # initialize with iterable of pairs
state_capitals = dict(NC='Raleigh', NY='Albany', NJ="Trenton") # named parameters
```

The `{}` literal dictionary constructor

Initialize `{}` with a comma-separated list of `key:value` pairs, or leave empty to create an empty dictionary.

```
d1 = {} # empty dict
state_capitals = {'NC': 'Raleigh', 'NY': 'Albany', 'NJ': 'Trenton'} # literal dict
```

Adding elements

In either case, To add more elements, assign keys and values using square brackets.

```
state_capitals['FL'] = 'Tallahassee'
state_capitals['CA'] = 'Sacramento'
```

Example

creating_dicts.py

```
d1 = dict() # create new empty dict

airports = {'IAD': 'Dulles', 'SEA': 'Seattle-Tacoma', # initialize dict with literal
            key/value pairs (keys can be any string, number or tuple)
            'RDU': 'Raleigh-Durham', 'LAX': 'Los Angeles'}

d2 = {}
d3 = dict(red=5, blue=10, yellow=1, brown=5, black=12) # initialize dict with named
parameters; keys must be valid identifier names

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

state_caps = dict(pairs) # initialize dict with an iterable of pairs

print(d3['red']) # print value for given key
print(airports['LAX'])

airports['SLC'] = 'Salt Lake City' # assign to new key
airports['LAX'] = 'Lost Angels' # overwrite existing key
print(airports['SLC'])
```

creating_dicts.py

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

Table 15. 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>	<code>D[k]</code> 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>	<code>a[k]</code> if k in D, else x (also setting it)

Table 16. 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>	<code>a[k]</code> if k in D, else x (and remove k)
<code>D.popitem()</code>	remove and return a (key, value) pair. Pairs are removed in LIFO order.

Getting dictionary values

- `d[key]` missing key raises `KeyError`
- `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 `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

```
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']) # print value where key is 'SLC'

code = 'PSP'

if code in airports: # is key in dictionary?
    print(airports[code]) # print key if key is in dictionary
else:
    print(f"{code} not in airports")

print(airports.get(code)) # get value if key in dict, otherwise get None
print(airports.get(code, 'NO SUCH AIRPORT')) # get value if key in dict, otherwise get
'NO SUCH AIRPORT'

print(airports.setdefault(code, 'Palm Springs')) # get value if key in dict, otherwise
get 'Palm Springs' AND set key
print(code in airports) # check for key in dict
```

getting_dict_values.py

```
5
Los Angeles
Salt Lake City
PSP not in airports
None
NO SUCH AIRPORT
Palm Springs
True
```

Iterating over a dictionary

- Dictionary itself is iterable of keys
- `d.items()` iterable of key/value tuples
- `d.keys()` iterable of keys
- `d.values()` iterable of values

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

To do something with the elements ordered by keys, pass `dictionary.items()` to the `sorted()` function and loop over the result.



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

Example

`iterating_over_dicts.py`

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

for abbr, airport in airports.items(): # items() returns an iterable of key:value pairs
    print(abbr, airport)

print('-' * 60)

for abbr, airport in sorted(airports.items()): # iterate, sorted by key
    print(abbr, airport)
```

`iterating_over_dicts.py`

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

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

Reading file data into a dictionary

- Data must have unique key
- Get key from one column, value can be any data derived from other columns

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.



Use a tuple of keys if no one column is unique.

Example

`read_into_dict_of_tuples.py`

```
from pprint import pprint

knight_info = {} # create empty dict

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

pprint(knight_info)
print()

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

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

`read_into_dict_of_tuples.py`

```
{'Arthur': ('King', 'blue', 'The Grail', 'King of the Britons'),  
 'Bedevere': ('Sir', 'red', 'no blue!', 'The Grail', 'AARRRRRRGGGGHH'),  
 '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



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.



Check out the `Counter` class in the `collections` module

Example

`count_with_dict.py`

```
counts = {} # create new empty dict

with open("../DATA/breakfast.txt") as breakfast_in:
    for line in breakfast_in:
        breakfast_item = line.rstrip()
        if breakfast_item in counts: # check to see if current item in dict
            counts[breakfast_item] = counts[breakfast_item] + 1 # if so, increment
            count for specified key
        else:
            counts[breakfast_item] = 1 # else add new element

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

count_with_dict.py

```
spam 34
Lucky Charms 4
eggs 3
oatmeal 1
sausage 4
bacon 6
pancakes 2
dosas 4
waffles 2
crumpets 1
```

NOTE:

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
```

See [count_with_dict_terse.py](#) for an example.

About sets

- Find unique values
- Check for membership
- Find union or intersection
- Like a dictionary where all values are True

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 `set(iterable)`
- 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)
 - difference

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

```
set1 = {'red', 'blue', 'green', 'purple', 'green'} # create literal set
set2 = {'green', 'blue', 'yellow', 'orange'}

set1.add('taupe') # add element to set (ignored if already in set)

print(f"{set1 = }")
print(f"{set2 = }")
print(f"{set1 & set2 = }") # intersection -- common items
print(f"{set1 ^ set2 = }") # XOR -- non-common items (in one set but not both)
print(f"{set1 | set2 = }") # union -- unique combination of both sets
print(f"{set1 - set2 = }") # difference -- Remove items in right set from left set
print(f"{set2 - set1 = }")
print()

with open('../DATA/breakfast.txt') as breakfast_in:
    food = breakfast_in.read().splitlines()

unique_food = set(food) # Create set from iterable (e.g., list)
print(unique_food)
```

set_examples.py

```
set1 = {'taupe', 'purple', 'blue', 'green', 'red'}
set2 = {'blue', 'yellow', 'orange', 'green'}
set1 & set2 = {'blue', 'green'}
set1 ^ set2 = {'taupe', 'purple', 'orange', 'yellow', 'red'}
set1 | set2 = {'taupe', 'purple', 'blue', 'orange', 'yellow', 'green', 'red'}
set1 - set2 = {'taupe', 'purple', 'red'}
set2 - set1 = {'yellow', 'orange'}
```

{'eggs', 'waffles', 'oatmeal', 'dosas', 'sausage', 'crumpets', 'pancakes', 'Lucky Charms', 'spam', 'bacon'}

Table 17. Set functions and methods

Function/Operator	Description (S=set, S2=other set, m=member)
<code>m in S</code>	True if S contains m
<code>m not in S</code>	True if S does not contain m
<code>len(s)</code>	the number of items in S
<code>S.add(m)</code>	Add m to S (if S already contains m do nothing)
<code>S.clear()</code>	remove all members from S
<code>S.copy()</code>	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 KeyError 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 8 Exercises

Exercise 8-1 (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 numeric test scores). Print out the student names, one per line with the numeric score and letter grade. After printing all the scores, print the average score.



Be sure to convert the numeric score to an integer when storing it in the dictionary.

Table 18. Grading Scale

Score	Letter grade
95-100	A
89-94	B
83-88	C
75-82	D
< 75	F

Exercise 8-2 (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 8-3 (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 8-4 (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 9: Functions

Objectives

- Creating functions
- Returning values from functions
- Passing required and optional positional parameters
- Passing required and optional named (keyword) parameters
- Understanding variable scope

Defining a function

- Indent body
- Specify parameters
- Variables are local by default

Functions are one of Python's callable types. Once a function is defined, it can be called from anywhere.

Functions can take required or optional parameters.

Functions must be defined before they can be called.

Define a function with the **def** keyword, the name of the function, a (possibly empty) list of parameters in parentheses, and a colon.

Example

function_basics.py

```
def say_hello(): # Function takes no parameters
    print("Hello, world")
    print()
    # If no return statement, return None

say_hello() # Call function (arguments, if any, in () )

def get_hello():
    return "Hello, world" # Function returns value

h = get_hello() # Store return value in h
print(h)
print()

def sqrt(num): # Function takes exactly one argument
    return num ** .5

m = sqrt(1234) # Call function with one argument
n = sqrt(2)

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

function_basics.py

```
Hello, world
Hello, world
m is 35.128 n is 1.414
```

Function parameters

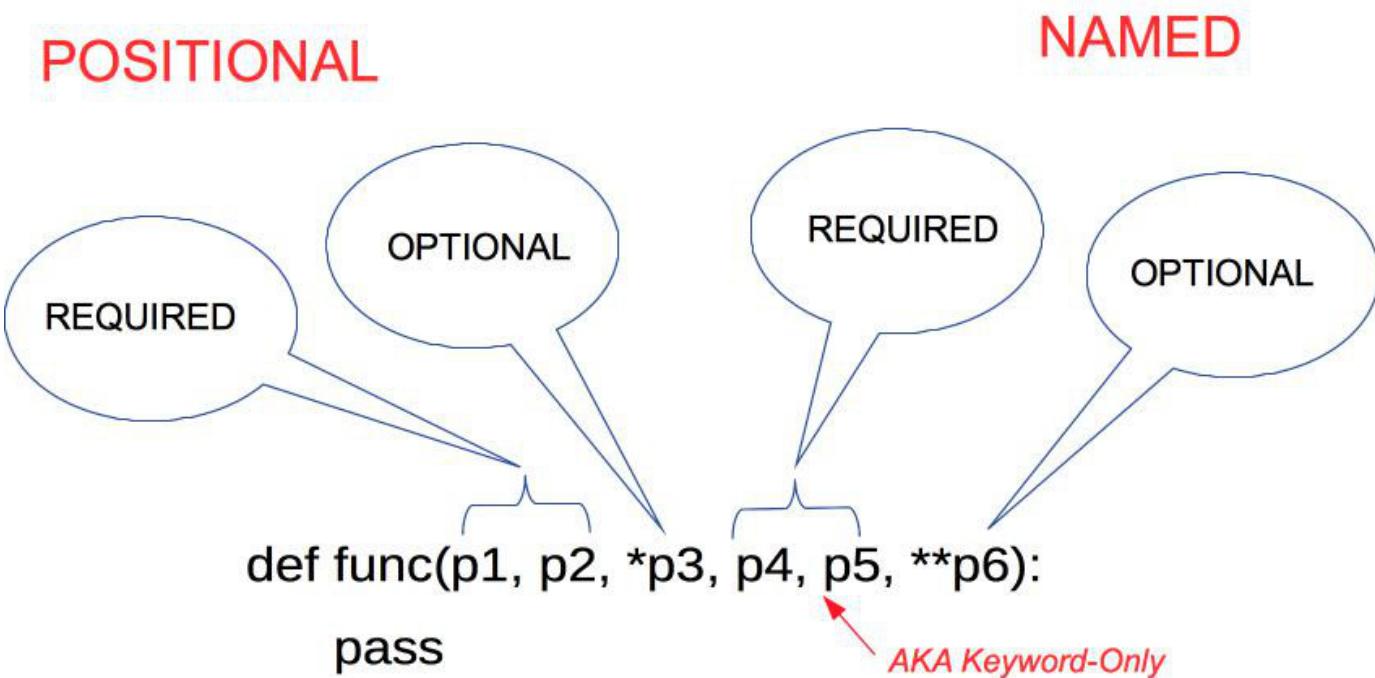
- Positional or named
- Required (fixed) or optional
- Arguments are copied to parameters
- Not typed

Parameters define how the function arguments, passed when the function is called, are used.

Most of the time you will use required positional parameters. These are fixed parameters, and when the function is called, there must be an argument for each parameter.

Default values

You can supply a default value for required parameters, and if an argument is not passed for the parameter, the default value will be used. Parameters with defaults must come after parameters that do not have defaults.



Positional parameters details

Required positional parameters

Specify one or more positional parameters. The interpreter will then expect exactly that many parameters. Positional parameters are available via their names.

```
def spam(a,b,c):  
    # function body
```

This function expects three parameters, which can be of any Python data type.

Positional parameters can have default values, in which case the parameters can be omitted when the function is called.



Positional parameters may also be passed by name.

Optional positional parameters

Prefix a parameter with one asterisk to accept any number of positional parameters. The parameter name will be a tuple of all the values.

```
def eggs(*params):  
    # function body
```

This function will take any number of arguments, which are then available in the tuple **params**.

Named parameters

Keyword-only parameters (required named parameters)

Keyword-only parameters are required parameters with specific names that come after optional parameters, but before optional named parameters. Keyword-only parameters are available in the function via their names. This is in comparison to normal keyword parameters, which are all grouped into a dictionary.

If the function doesn't require optional parameters, use a single '*' character as a placeholder after any fixed parameters.

```
def spam(*, ham=True, eggs=5):  
    # function body
```

The **pandas** `read_csv()` method is a great example of a function where named parameters are a good fit. There are over twenty possible parameters, and it would be difficult for users to provide all of them with every call, so it has named parameters, which all have reasonable defaults. The only required parameter is the name of the file to read.

```
read_csv(filepath_or_buffer, sep=',', delimiter=None, header='infer', names=None,  
index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True,  
dtype=None, engine=None, converters=None, true_values=None, false_values=None,  
skipinitialspace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True,  
na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False,  
infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False,  
iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.',  
lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None,  
encoding=None, dialect=None, tupleize_cols=False, error_bad_lines=True,  
warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True,  
delim_whitespace=False, as_recarray=False, compact_ints=False, use_unsigned=False,  
low_memory=True, buffer_lines=None, memory_map=False, float_precision=None)
```

Keyword parameters (optional named parameters)

Specify optional named parameters. Prefix the parameter with two asterisks. The parameter is a dictionary of the names and values passed in as "name=value" pairs.

```
def spam(**kw):  
    # function body
```

This function takes any number of keyword arguments, which are available in the dictionary kw:

```
spam(name="bob",grade=10)
```



There is a fifth parameter type, *positional-only*, which comes before the normal positional parameters, and which can only be passed by position.

Example

function_parameters.py

```
def fun_one(): # no parameters
    print("Hello, world")

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

def fun_two(n): # one required parameter
    return n ** 2

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

def fun_three(count=3): # one required parameter with default value
    for _ in range(count):
        print("spam", end=' ')
    print()

fun_three()
fun_three(10)
print()

def fun_four(n, *opt): # one fixed, plus optional parameters
    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): # keyword-only parameters
    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): # keyword (named) parameters
    print("fun_six()")
    for name in named_args:
        print(f"{name} ==> {named_args[name]}")

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

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
```

Returning values

- Use `return` statement
- Return any object
- Return `None` by default

The return value of a function is the value of the function when it is called.

To return a value, use the `return` statement. It can return any Python object, including strings, integers, floats, lists, dictionaries, tuples, or any others.

If there is no `return` statement, the function returns `None`. `return` without a value also returns `None`.

Example

```
return      # return None
return 5    # return integer 5
return x    # return object x
return name, quest, color # return tuple of values
```



Remember that `return` is a statement, not a function.

Variable scope

- Assignment inside function creates local variables
- Parameters are local variables
- All other variables are global

Any names created in a function, including variable assignment, are local to the function. They are only visible within the function.

All other names are global to the current file, including function, class, and module names.

If you use an existing variable that has not been assigned to in the function, then it will use the global variable.

It is best practice to minimize the use of global variables; they can make a program hard to read and debug.

Example

variable_scope.py

```
x = 5

def spam():
    x = 22 # Local variable; does not modify global x
    print("spam(): x is", x)
    y = "wolverine" # Local variable
    print("spam(): y is", y)

def eggs():
    print("eggs(): x is", x) # Uses global x since there is no local x
    y = "wolverine"
    print("eggs(): y is", y)

spam()
print()
eggs()
print()
print("main: x is ", x)
```

variable_scope.py

```
spam(): x is 22
spam(): y is wolverine

eggs(): x is 5
eggs(): y is wolverine

main: x is 5
```

builtin***global (file level)***

```
count = 0  
limit = 10
```

local (in subroutine)

```
global count  
count += 1  
limit = 25
```

The global statement

- Use **global** for assignment to global variables

What happens when you want to change a global variable? The **global** statement allows you to declare a global variable within a function. That is, when you assign to the variable, you are assigning to the global variable, instead of a local variable. This is true, even if the global variable does not yet exist.



Use of the **global** statement is discouraged, as it can make code maintenance more difficult.

Example

`global_statement.py`

```
x = 5

def spam():
    global x # Mark x as global, not local
    x = 22 # Modify global variable x
    print("spam(): x is", x)

spam()
print("main: x is ", x)
```

`global_statement.py`

```
spam(): x is 22
main: x is 22
```

Chapter 9 Exercises

Exercise 9-1 (dirty_strings.py)

Using the existing script `dirty_strings.py`, implement the function named `cleanup()`. This function accepts one string as input and returns a copy of the string with whitespace removed from the beginning and the end, and all upper case letters changed to lower case.

The code to loop over a list of "dirty" strings and call the function is already in place. All you have to do is remove the `pass` statement and put your code in the `cleanup()` function.

Exercise 9-2 (c2f_func.py)

Define a function named `c2f()` that takes one number as a parameter, and then returns the value converted from Celsius to Fahrenheit. Test your function by calling it with the values 100, 0, 37, and -40 one at a time.

Example

```
f = c2f(100)
print(f)

f = c2f(-40)
print(f)
```

Exercise 9-3 (calc.py)

Write a simple four-function calculator. Repeatedly prompt the user for a math expression, which should consist of a number, an operator, and another number, all separated by whitespace. The operator may be any of "+", "-", "/", or "*". For example, the user may enter "9 + 5", "4 / 28", or "12 * 5". Exit the program when the user enters "Q" or "q". (Hint: split the input into the 3 parts – first value, operator, second value).

Write a function for each operator (named "add", "subtract", etc). As each line is read, pass the two numbers to the appropriate function, based on the operator, and get the result, which is then output to the screen. The division function should check to see whether the second number is zero, and if so, return an error message, rather than trying to actually do the math.



Expect an expression like "5 + 4" which you can split on whitespace.

FOR ADVANCED STUDENTS

Add more math operations; test the input to make sure it's numeric (although in real life you should use a `try` block to validate numeric conversions).

Chapter 10: Creating and using modules

Objectives

- Using the import statement to load modules
- Aliasing module and package names for convenience
- Understanding what .pyc files are
- Setting locations to search for local modules
- Creating local (user-defined) modules
- Building packages containing multiple modules

What is a module?

- Sharable library of python code
- Can have initialization code

A module is just a Python script that is imported in some other Python script. In some languages it would be called a library.

When a function, class, or variable is used in more than one place, it should be put in a module, in order to avoid "cut-and-paste disease". If the function or class need to change, you only need to change it in one place.

All code in a module will be executed the first time a module is loaded, and thus can provide initialization code.

Modules end in **.py**, like normal scripts.

Modules can also be written in C/C++ or other languages, and implemented as binary libraries (.dll, .so, or .dylib).

Creating Modules

- No special syntax
- Ends in .py
- Add documentation strings

You can create your own modules easily. Any valid Python source file may be loaded as a module.

It is a good idea to provide a documentation string for each function. This is an unassigned string which is the first statement in the body of a function. It will be used by **pydoc**, **Sphinx**, IDEs (**PyCharm**, **VS Code**, etc.), and other tools.

Module names should be lower case, and must follow the standard rules for Python names — letters, digits, and underscores only.

Example

geometry.py

```
"""
General geometry-related functions

Syntax:

area = circle_area(diameter)
area = rectangle_area(length, width)
area = square_area(side)
"""

import math    # load math.py

PI = math.pi

def circle_area(diameter):
    """
    Compute the area of a circle from a given diameter

    :param diameter: Diameter of circle
    :return: Area of circle
    """

    radius = diameter / 2
    return PI * (radius ** 2)

def rectangle_area(length, width):
    """
    Compute the area of a rectangle.

    :param length: length of longer side
    :param width: length of shorter side
    :return: Area of rectangle
    """

    return length * width

def square_area(side):
    """
    Compute area of a square.

    :param side: Length of one side
    :return: Area of square
    """

    return side ** 2

if __name__ == "__main__":
    print("Hello, world!")
```

```
area1 = square_area(15)
print(f"area1: {area1}")

area2 = circle_area(22)
print(f"area2: {area2}")

area3 = rectangle_area(9, 13)
print(f"area3: {area3}")
```

The import statement

- Used to load a module
- Import entire module or just specified objects

The `import` statement is used to load modules. It can be used in two main ways, with several variations. Both modules and names from the module can be aliased — this is useful to save typing, especially when using packages. It can also make code more readable.

```
import MODULE  
from MODULE import name, ...
```

Import module

This imports the module and creates a module object. Use the module as a prefix for any name in the module.

```
import geometry
```

Example

use_geometry_1.py

```
import geometry

a1 = geometry.circle_area(8)
a2 = geometry.rectangle_area(10, 12)
a3 = geometry.square_area(7.9)
print(a1, a2, a3)
```

use_geometry_1.py

```
50.26548245743669 120 62.410000000000004
```

Import module with alias

Same as before, but a shorter name for the module

```
import geometry as g
```

Example

`use_geometry_2.py`

```
import geometry as g

a1 = g.circle_area(8)
a2 = g.rectangle_area(10, 12)
a3 = g.square_area(7.9)
print(a1, a2, a3)
```

`use_geometry_2.py`

```
50.26548245743669 120 62.410000000000004
```

Import names from module

This imports names (classes, functions, variables) from the module, but does not create a module object. Only the specified names are imported. However, this does not save memory, as all of the code in the module is still executed.

```
from geometry import circle_area, rectangle_area, square_area
```

Example

`use_geometry_3.py`

```
from geometry import circle_area, rectangle_area, square_area

a1 = circle_area(8)
a2 = rectangle_area(10, 12)
a3 = square_area(7.9)
print(a1, a2, a3)
```

`use_geometry_3.py`

```
50.26548245743669 120 62.410000000000004
```

Import all names from module

```
from geometry import *
```

This will import all names from the module into the current namespace, unless they start with `_`. Be careful when using this after importing specific names from a different ; you might unknowingly overwrite names from the previous module.

Example

`use_geometry_4.py`

```
from geometry import *

a1 = circle_area(8)
a2 = rectangle_area(10, 12)
a3 = square_area(7.9)
print(a1, a2, a3)
```

`use_geometry_4.py`

```
50.26548245743669 120 62.410000000000004
```

Import names with aliases

You can also alias the names that are imported from the module.

```
from geometry import circle_area as c_area, rectangle_area as r_area, square_area as s_area
```

Example

use_geometry_5.py

```
from geometry import circle_area as c_area, rectangle_area as r_area, square_area as s_area

a1 = c_area(8)
a2 = r_area(10, 12)
a3 = s_area(7.9)
print(a1, a2, a3)
```

Where did `__pycache__` come from?

- import precompiles modules as needed
- Loading (but not execution) is faster

After running a script which uses module `cheese.py` in the current directory, you will notice that a folder named `__pycache__` has appeared. The interpreter saves a compiled version of each imported module into the `__pycache__` folder.

This speeds up the loading of modules, as whitespace, comments, and other non-code items are removed. It does not, however, speed up the execution of the module, as, once loaded, the code in memory is the same.

Subsequent invocations of the script using the module will load the `.pyc` version. If the modification date of the original (`.py`) file is later than the `.pyc`, the interpreter will recompile the module.

Bottom line, you do not need to do anything about the cached versions of modules, since they are managed by the interpreter.

Module search path

- Searches current dir first, then predefined locations
- Paths stored in `sys.path`
- Add locations to `PYTHONPATH`

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`.

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`.

Windows

```
set PYTHONPATH=C:"\Documents and settings\Bob\Python"
```

non-Windows (Mac/Linux)

```
export PYTHONPATH="/home/bob/python"
```



Do not directly append to `sys.path` in scripts. This can result in "brittle" scripts, that will fail if the location of the imported modules changes.

Example

sys_path.py

```
import sys  
  
for path in sys.path:  
    print(path)
```

sys_path.py

```
/Users/jstrick/curr/courses/python/common/examples  
/Library/Frameworks/Python.framework/Versions/3.11/lib/python311.zip  
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11  
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/lib-dynload  
/Library/Frameworks/Python.framework/Versions/3.11/lib/python3.11/site-packages
```

Executing modules as scripts

- `{name}` is name of current module.
 - set to "`{main}`" if run as script
 - set to "`module_name`" if imported
- test with `if {name} == "{main}"`
- Module can be both run directly and 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; all of the file's code, including the code under `if {name} == "{main}"`, is executed.

If it is not `{main}`, then you are importing the file, and the code under `if {name} == "{main}"` will not be executed.

script

(from the command line)

```
python filename.py
```

`{name}` is set to "`{main}`"

module

(in a python script or module)

```
import module
```

`{name}` is set to "`module`"

Example

using_main.py

```
import sys

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

# constants (AKA global variables)

# main function
def main(args): # Program entry point. While main is not a reserved word, it is a strong
    function1()
    function2()

# other functions
def function1():
    print("hello from function1()")

def function2():
    print("hello from function2()")

if __name__ == '__main__':
    main(sys.argv[1:]) # Call main() with the command line parameters (omitting the
script itself)
```

Packages

- Collection
 - Modules
 - Subpackages
- Are just folders

A package is a collection of modules that have been grouped in a directory for convenience.

Load a module from a package with `from package import module`.

Load modules from subpackages with dot notation: `from pkg.pkg import module`

Import functions or classes: `from pkg.pkg.module import func_or_class`

Example

The following directory structure implements package media, which includes modules cd and dvd

```
media
media/__init__.py
media/dvd.py
media/cd.py
```

To use function `search` from module `dvd`:

```
import media.dvd
media.dvd.search("Thor")
```

Or

```
from media import dvd
dvd.search("Thor")
```

Or

```
from media.dvd import search
search("Thor")
```

`__init__.py` module

- Initializes package
- Optional

If a module named `__init__.py` is present in the package, it is executed when the package or any of its modules are imported. There are three things you might put in this module:

1. *docstrings* — documentation for the entire package
2. "*convenience imports*" — imports so that names from modules are available through the package itself
3. *shared code* — code to initialize the package, and that can be shared by all modules and subpackages within the module

When the batteries aren't included

- PyPI (Python Package Index) has over 400,000 projects
- Install with `pip`

Although the Python distribution claims to be "batteries included", functionality beyond the standard library is provided by so-called "third party modules". There are over 400,000 such packages in the Python Package Index (<http://pypi.python.org/pypi>).

These modules can be installed with the `pip` utility.

Chapter 10 Exercises

Exercise 10-1

Part A (temp_conv.py)

Create a module that implements two temperature conversion functions, c2f and f2c.

Conversion formulas

$$\begin{aligned} F &= ((9 * C) / 5.0) + 32 \\ C &= (F - 32) * (5.0/9) \end{aligned}$$

Write some simple code in the `if __name__ == "__main__"` section to do a simple test of the functions.

Part B (c2f_mod.py)

Re-implement **c2f.py** using temp_conv

Part C (f2c.py)

Re-implement **c2f.py** as **f2c.py** (take Fahrenheit on command line) using temp_conv

Part D (c2f_loop_mod.py)

Re-implement **c2f_loop.py** using temp_conv.

Chapter 11: Errors and Logging

Objectives

- Understanding syntax errors
- Handling exceptions with try-except-else-finally
- Learning the standard exception objects
- Setting up basic logging
- Logging exceptions

Exceptions

- Generated when runtime errors occur
- Usually fatal if not handled

Even if code is syntactically correct, run-time errors can occur once a script is launched. A common error is to attempt to open a non-existent file. Such errors are also 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.

Python provides **try-except** blocks to catch, or handle, the exceptions. When an exception is caught, you can log the error, provide a default value, or gracefully shut down the program, depending on the severity of the error.



Custom exceptions can be created by sub-classing the `Exception` object.

Example

`exception_unhandled.py`

```
x = 5
y = "cheese"

z = x + y # Adding a string to an int raises TypeError
```

`exception_unhandled.py 2>&1`

```
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/exception_unhandled.py", line
5, in <module>
    z = x + y # Adding a string to an int raises TypeError
           ~~^~~
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 in a `try` block. After the `try` block, you must specify an `except` block with the expected exception. If an exception is raised in the `try` block, execution stops and the interpreter checks to see if the exception matches the `except` block. If it matches, it executes the `except` block and execution continues; otherwise, the exception is treated as fatal (the default behavior) and the interpreter exits.

Example

`exception_simple.py`

```
try: # Execute code that might have a problem
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")

except TypeError as err: # Catch the expected error; assign error object to err
    print("Naughty programmer!", err)

print("After try-except") # 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

```
try:  
    x = 5  
    y = "cheese"  
    z = x + y  
    f = open("sesame.txt")  
    print("Bottom of try")  
  
except (IOError, TypeError) as err: # Use a tuple of 2 or more exception types  
    print("Naughty programmer! ", err)
```

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`

```
try:  
    x = 5  
    y = "cheese"  
    z = x + y  
    f = open("sesame.txt")  
    print("Bottom of try")  
  
except Exception as err: # Will catch _any_ exception  
    print("Naughty programmer!", err)
```

`exception_generic.py`

```
Naughty programmer! unsupported operand type(s) for +: 'int' and 'str'
```

Ignoring exceptions

- Use `pass`
- Not usually recommended

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`

```
try:  
    x = 5  
    y = "cheese"  
    z = x + y  
    f = open("sesame.txt")  
    print("Bottom of try")  
  
except(TypeError, IOError): # Catch exceptions, and do nothing  
    pass
```

`exception_ignore.py`

no output



It is usually a bad idea to completely ignore an error. It might be indicating an unexpected problem in your code.

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` block 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

```
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(f"uh-oh, when y = {y}, {err}")  
    else:  
        total += quotient # Only if no exceptions were raised  
print(total)
```

exception_else.py

```
uh-oh, when y = 0, division by zero  
5.2
```

Cleaning up with finally

- Code runs whether or not exception raised
 - Even if script exits in `except` block
- Clean up resources

A `finally` block can be used in addition to, or instead of, 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.

What makes `finally` different from just putting statements after try-except-else is that the `finally` block will execute even if there is a `return()` or `exit()` in the `except` block.

The purpose of a `finally` block is to clean up any resources left over from the `try` block. Examples include closing network connections and removing temporary files.

Example

exception_finally.py

```
try:  
    x = 5  
    y = 37  
    z = x + y  
    print("z is", z)  
except TypeError as err:    # Catch TypeError  
    print("Caught exception:", err)  
finally:  
    print("Don't care whether we had an exception") # Print whether TypeError is caught  
or not  
  
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")
```

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 3.11)

```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
└── Exception
    ├── ArithmeticError
    │   ├── FloatingPointError
    │   ├── OverflowError
    │   └── ZeroDivisionError
    ├── AssertionError
    ├── AttributeError
    ├── BufferError
    ├── EOFError
    ├── ExceptionGroup [BaseExceptionGroup]
    ├── 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
    ├── StopAsyncIteration
    └── StopIteration
```

```
    └── SyntaxError
        └── IndentationError
            └── TabError
    └── SystemError
    └── TypeError
    └── ValueError
        └── UnicodeError
            ├── UnicodeDecodeError
            ├── UnicodeEncodeError
            └── UnicodeTranslateError
    └── Warning
        ├── BytesWarning
        ├── DeprecationWarning
        ├── EncodingWarning
        ├── FutureWarning
        ├── ImportWarning
        ├── PendingDeprecationWarning
        ├── ResourceWarning
        ├── RuntimeWarning
        ├── SyntaxWarning
        ├── UnicodeWarning
        └── UserWarning
```

Simple Logging

- Configure with `logging.basicConfig()`
 - Specify file name
 - Configure the minimum logging level
- Messages added at different levels
- Call methods on `logging`

For simple logging, just configure the log file name and minimum logging level with the `basicConfig()` method. Then call one of the per-level methods, such as `logging.debug()` or `logging.error()`, to output a log message for that level. If the message is at or above the minimal level, it will be added to the log file.

The file will continue to grow, and must be manually removed or truncated. If the file does not exist, it will be created.

The logger module provides 5 levels of logging messages, from DEBUG to CRITICAL. When you set up a logger, you specify the minimum level of messages to be logged. If you set up the logger with the minimum level set to ERROR, then only messages at ERROR and CRITICAL levels will be logged. Setting the minimum level to DEBUG allows all messages to be logged.

Table 19. Logging Levels

Level	Value
CRITICAL	50
FATAL	
ERROR	40
WARN	30
WARNING	
INFO	20
DEBUG	10
UNSET	0

Example

logging_simple.py

```
import logging

logging.basicConfig(
    filename='../../LOGS/simple.log',
    level=logging.WARNING,
)

logging.warning('This is a warning') # message will be output
logging.debug('This message is for debugging') # message will NOT be output
logging.error('This is an ERROR') # message will be output
logging.critical('This is ***CRITICAL***') # message will be output
logging.info('The capital of North Dakota is Bismark') # message will not be output
```

..../LOGS/simple.log

```
WARNING:root:This is a warning
WARNING:root:This is a warning
ERROR:root:This is an ERROR
ERROR:root:This is an ERROR
CRITICAL:root:This is ***CRITICAL***
CRITICAL:root:This is ***CRITICAL***
```

...

Formatting log entries

- Add format=format to basicConfig() parameters
- Format is a string containing directives and (optionally) other text
- Use directives in the form of %(item)type
- Other text is left as-is

To format log entries, provide a format parameter to the `basicConfig()` method. This format will be a string contain special directives (i.e. Placeholders) and, optionally, other text. The directives are replaced with logging information; other data is left as-is.

Directives are in the form `%(item)type`, where `item` is the data field, and `type` is the data type.

To format the date from the `%(asctime)s` directive, assign a value to the `datefmt` parameter using date components from the table that follows.

Example

`logging_formatted.py`

```
import logging

logging.basicConfig(
    format='%(levelname)s %(name)s %(asctime)s %(filename)s %(lineno)d %(message)s', #
set the format for log entries
    datefmt="%x-%X",
    filename='../LOGS/formatted.log',
    level=logging.INFO,
)

logging.info("this is information")
logging.warning("this is a warning")
logging.error("this is an ERROR")
value = 38.7
logging.error("Invalid value %s", value)
logging.info("this is information")
logging.critical("this is critical")
```

../LOGS/formatted.log

```
INFO root 02/29/24-10:15:22 logging_formatted.py 11 this is information
INFO root 02/29/24-10:15:22 logging_formatted.py 11 this is information
WARNING root 02/29/24-10:15:22 logging_formatted.py 12 this is a warning
ERROR root 02/29/24-10:15:22 logging_formatted.py 13 this is an ERROR
WARNING root 02/29/24-10:15:22 logging_formatted.py 12 this is a warning
ERROR root 02/29/24-10:15:22 logging_formatted.py 13 this is an ERROR
ERROR root 02/29/24-10:15:22 logging_formatted.py 15 Invalid value 38.7
INFO root 02/29/24-10:15:22 logging_formatted.py 16 this is information
ERROR root 02/29/24-10:15:22 logging_formatted.py 15 Invalid value 38.7
CRITICAL root 02/29/24-10:15:22 logging_formatted.py 17 this is critical
INFO root 02/29/24-10:15:22 logging_formatted.py 16 this is information
CRITICAL root 02/29/24-10:15:22 logging_formatted.py 17 this is critical
```

Table 20. Log entry formatting directives

Directive	Description
<code>%(name)s</code>	Name of the logger (logging channel)
<code>%(levelno)s</code>	Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL)
<code>%(levelname)s</code>	Text logging level for the message ("DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL")
<code>%(pathname)s</code>	Full pathname of the source file where the logging call was issued (if available)
<code>%(filename)s</code>	Filename portion of pathname
<code>%(module)s</code>	Module (name portion of filename)
<code>%(lineno)d</code>	Source line number where the logging call was issued (if available)
<code>%(funcName)s</code>	Function name
<code>%(created)f</code>	Time when the LogRecord was created (time.time() return value)
<code>%(asctime)s</code>	Textual time when the LogRecord was created
<code>%(msecs)d</code>	Millisecond portion of the creation time
<code>%(relativeCreated)d</code>	Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded (typically at application startup time)
<code>%(thread)d</code>	Thread ID (if available)
<code>%(threadName)s</code>	Thread name (if available)
<code>%(process)d</code>	Process ID (if available)
<code>%(message)s</code>	The result of record.getMessage(), computed just as the record is emitted

Table 21. Date Format Directives

Directive	Meaning	Notes
%a	Locale's abbreviated weekday name	
%A	Locale's full weekday name	
%b	Locale's abbreviated month name	
%B	Locale's full month name	
%c	Locale's appropriate date and time representation	
%d	Day of the month as a decimal number [01,31]	
%f	Microsecond as a decimal number [0,999999], zero-padded on the left	(1)
%H	Hour (24-hour clock) as a decimal number [00,23]	
%I	Hour (12-hour clock) as a decimal number [01,12]	
%j	Day of the year as a decimal number [001,366]	
%m	Month as a decimal number [01,12]	
%M	Minute as a decimal number [00,59]	
%p	Locale's equivalent of either AM or PM.	(2)
%S	Second as a decimal number [00,61]	(3)
%U	Week number (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0	(4)
%W	Weekday as a decimal number [0(Sunday),6]	
%w	Week number (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0	(4)
%x	Locale's appropriate date representation	
%X	Locale's appropriate time representation	
%y	Year without century as a decimal number [00,99]	
%Y	Year with century as a decimal number	
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive)	(5)
%Z	Time zone name (empty string if the object is naive)	
%%	A literal '%' character	

Logging exception information

- Use `logging.exception()`
- Adds exception info to message
- Only in `except` blocks

The `logging.exception()` function will add a traceback to the log in addition to the specified message. It should only be called in an `except` block.

This is different from putting the file name and line number in the log entry. That puts the file name and line number where the logging method was called, while `logging.exception()` specifies the line where the error occurred.

Example

`logging_exception.py`

```
import logging

logging.basicConfig( # configure logging
    filename='../../LOGS/exception.log',
    level=logging.WARNING, # minimum level
)

for i in range(3):
    try:
        result = i/0
    except ZeroDivisionError:
        logging.exception('Logging with exception info') # add exception info to the log
```

..../LOGS/exception.log

```
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
      ~~~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
      ~~~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
      ~~~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
      ~~~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
      ~~~
ZeroDivisionError: division by zero
ERROR:root:Logging with exception info
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/logging_exception.py", line
11, in <module>
    result = i/0
      ~~~
ZeroDivisionError: division by zero
```

For more information

- <https://docs.python.org/3/howto/logging.html>
- <https://docs.python.org/3/howto/logging-cookbook.html>
- <https://pymotw.com/3/logging/index.html>

Chapter 11 Exercises

Exercise 11-1 (c2f_loop_safe.py)

Rewrite `c2f_loop.py` to handle the error that occurs if the user enters non-numeric data. The script should print an error message and go back to the top of the loop if an error occurs.

Exercise 11-2 (c2f_batch_safe.py)

Rewrite `c2f_batch.py` to handle the `ValueError` that occurs if `sys.argv[1]` is not a valid number. Log the error to a file named `c2f_batch.log`.

Chapter 12: Introduction to Python Classes

Objectives

- Understanding the big picture of OO 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

About object-oriented programming

- Definitions of objects
- Can be used directly as well
- Objects contain data and methods

Python is an object-oriented language. It supports the creation of classes, which define *objects* (AKA *instances*).

Objects contain both data and the methods (functions) 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 data*.

Each class defines a *constructor*, which initializes and returns an object.

Classes may inherit attributes (data and methods) from other classes.



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. If no base classes are needed, you can skip the parentheses.

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**.

A method is a function defined in a class. All methods, including the constructor, are passed the object itself. This is conventionally named **self**, and while this is not mandatory, it is a strong convention and best practice.

Basic layout of a class:

```
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()  
  
    @property  
    def some_property(self, ...):  
        return self.some_attribute
```

Example

simple_class.py

```
class Simple: # default base class is object
    def __init__(self, message_text): # constructor
        self._message_text = message_text # message text stored in instance object

    def text(self): # instance method
        return self._message_text

if __name__ == "__main__":
    msg1 = Simple('hello') # instantiate an instance of Simple
    print(msg1.text()) # call instance method

    msg2 = Simple('hi there') # create 2nd instance of Simple
    print(msg2.text())
```

simple_class.py

```
hello
hi there
```

Example

card.py

```
class Card:  
    def __init__(self, rank, suit):  
        self._rank = rank  
        self._suit = suit  
  
    @property  
    def rank(self):  
        return self._rank  
  
    @property  
    def suit(self):  
        return self._suit  
  
    @rank.setter  
    def rank(self, value):  
        pass  
  
    def __repr__(self):  
        return f"Card('{self._rank}', '{self._suit}')"  
  
    def __str__(self):  
        return f"{self.rank}-{self.suit}"
```

Example

carddeck.py

```
"""
Provide a CardDeck object and some utility methods

Usage:
from carddeck import CardDeck
c = CardDeck("Dealer-Name")
"""

import random
from card import Card
# or choose one of the following
# from card_named_tuple import Card
# from card_dataclass import Card

class CardDeck:
    """
    A deck of 52 cards for playing standard card games
    """

    # def __new__(): pass
    RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()
    CLUB = '\u2663\uFE0F' # FE0F converts chars to emoji
    DIAMOND = '\u2666\uFE0F'
    HEART = '\u2665\uFE0F'
    SPADE = '\u2660\uFE0F'
    SUITS = CLUB, DIAMOND, HEART, SPADE
    DEALER_NAMES = []

    def __init__(self, dealer_name):
        """
        CardDeck constructor

        :param dealer_name: Name of dealer as a string
        """
        self.dealer_name = dealer_name
        self._make_deck()

    def _make_deck(self):
        self._cards = list()
        for suit in self.SUITS:
            for rank in self.RANKS:
                card = Card(rank, suit)
                self._cards.append(card)
```

```
def draw(self):
    """
    Retrieve next available card from deck.

    Raises IndexError when deck is empty.

    :return: One cards as a (rank, suit) tuple.
    """
    try:
        return self._cards.pop(0) #
    except IndexError:
        print("Sorry, no more cards")

def shuffle(self):
    """
    Randomize the deck of cards

    :return: None
    """
    random.shuffle(self._cards)

@property
def dealer_name(self): # getter property
    """
    Set/Retrieve the dealer.

    If invalid type is assigned, raises TypeError

    :return: Dealer as a string
    """
    return self._dealer_name

@dealer_name.setter
def dealer_name(self, value): # setter property
    if isinstance(value, str):
        self._dealer_name = value
    else:
        raise TypeError("Dealer must be a string")

@property
def cards(self):
    """
    Retrieve cards

    :return: tuple of remaining cards
    """
    return tuple(self._cards) # make it read-only
```

```
@classmethod
def get_ranks(cls):
    """
    Return ranks as a list

    :return:
    """

    return cls.RANKS

def __len__(self):
    return len(self._cards)

def __str__(self):
    my_type = type(self)
    my_name = my_type.__name__
    return f"{my_name}({self.dealer_name},{len(self)})"

def __repr__(self):
    my_type = type(self)
    my_name = my_type.__name__
    return f"""{my_name}("{self.dealer_name}")"""

def __add__(self, other):
    my_type = type(self)
    tmp = my_type(self.dealer_name)
    tmp._cards = self._cards + other._cards
    return tmp

def __eq__(self, other):
    return (
        self._dealer_name == other.dealer
        and
        self._cards == other._cards
    )

if __name__ == '__main__':
    d = CardDeck("Mary")
    d.shuffle()
    print(d.cards)
```

carddeck.py

```
(Card('10', '♦'), Card('8', '♦'), Card('2', '♦'), Card('Q', '♦'), Card('3', '♦'),
Card('7', '♦'), Card('7', '♦'), Card('2', '♦'), Card('A', '♦'), Card('2', '♦'),
Card('J', '♦'), Card('5', '♦'), Card('K', '♦'), Card('10', '♦'), Card('Q', '♦'),
Card('3', '♦'), Card('5', '♦'), Card('6', '♦'), Card('6', '♦'), Card('8', '♦'),
Card('K', '♦'), Card('2', '♦'), Card('8', '♦'), Card('9', '♦'), Card('4', '♦'),
Card('9', '♦'), Card('6', '♦'), Card('Q', '♦'), Card('5', '♦'), Card('10', '♦'),
Card('4', '♦'), Card('7', '♦'), Card('9', '♦'), Card('3', '♦'), Card('J', '♦'),
Card('A', '♦'), Card('5', '♦'), Card('K', '♦'), Card('4', '♦'), Card('8', '♦'),
Card('10', '♦'), Card('6', '♦'), Card('3', '♦'), Card('A', '♦'), Card('4', '♦'),
Card('9', '♦'), Card('J', '♦'), Card('K', '♦'), Card('J', '♦'), Card('Q', '♦'),
Card('A', '♦'), Card('7', '♦'))
```

Example

play_cards.py

```
from carddeck import CardDeck

deck = CardDeck("Mary")

deck.shuffle()

for _ in range(10):
    card = deck.draw()
    print(card)
print()

print(deck)

print(deck.cards)
```

play_cards.py

```
4-♦
3-♦
A-♦
10-♦
J-♦
8-♦
A-♦
A-♦
J-♦
9-♦

CardDeck(Mary,42)
(Card('J', '♦'), Card('2', '♦'), Card('8', '♦'), Card('5', '♦'), Card('5', '♦'),
Card('6', '♦'), Card('Q', '♦'), Card('7', '♦'), Card('J', '♦'), Card('2', '♦'),
Card('8', '♦'), Card('K', '♦'), Card('Q', '♦'), Card('7', '♦'), Card('K', '♦'),
Card('3', '♦'), Card('7', '♦'), Card('7', '♦'), Card('2', '♦'), Card('10', '♦'),
Card('2', '♦'), Card('6', '♦'), Card('9', '♦'), Card('6', '♦'), Card('5', '♦'),
Card('4', '♦'), Card('9', '♦'), Card('4', '♦'), Card('Q', '♦'), Card('3', '♦'),
Card('A', '♦'), Card('K', '♦'), Card('6', '♦'), Card('9', '♦'), Card('Q', '♦'),
Card('4', '♦'), Card('K', '♦'), Card('10', '♦'), Card('5', '♦'), Card('8', '♦'),
Card('10', '♦'), Card('3', '♦'))
```

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 class when it's being created.

As with any Python function, the constructor's parameters can be fixed, optional, keyword-only, or keyword.

Private attributes

Name attributes (variables and methods) 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 *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 the first parameter. As with the constructor, the parameter should be named *self*.

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.

Why use properties? They provide a cleaner, simpler interface. Instead of

```
x = CardDeck("Ali")
x.set_dealer_name("Wilhelm")
print(x.get_dealer_name)
```

you can say

```
x = CardDeck("Ali")
x.dealer_name = "Wilhelm"
print(x.dealer_name)
```

properties.py

```
class Person():

    def __init__(self, firstname=None, lastname=None):
        self.first_name = firstname # calls property for validation
        self.last_name = lastname # calls property

    @property
    def first_name(self): # getter property
        return self._first_name

    @first_name.setter # decorator comes from getter property
    def first_name(self, value): # setter property
        if value is None or value.isalpha():
            self._first_name = value
        else:
            raise ValueError("ERROR: 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' # assign to property
    person2.first_name = 'Hortense'

    print(f"{person1.first_name} {person1.last_name}")
    print(f"{person2.first_name} {person2.last_name}")

    try:
        person3 = Person("R2D2")
    except ValueError as err:
        print(err)
    else:
        print(f"{person3.first_name} {person3.last_name}")
```

properties.py

```
Ferneater Eulalia
Hortense Pepperpot
ERROR: 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`

```
class Rabbit:
    LOCATION = "the Cave of Caerbannog" # class data (not duplicated in instances)

    def __init__(self, weapon):
        self.weapon = weapon

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon)) # instance method

    @classmethod # the @classmethod decorator makes a function receive the class object,
    not the instance object
    def get_location(cls): # get_location() is a _class_ method
        return cls.LOCATION # class methods can access class data via cls

r = Rabbit("a nice cup of tea")
print(Rabbit.get_location()) # call class method from class
print(r.get_location()) # 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
- Should not be used outside class
- 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.

Name private methods with a leading underscore. This does not protect them from use, but gives programmers a hint that they are 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. Specify 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 looked for in the first base class in the list. If not found, it is sought in the parent 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 constructor 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().__init__(args)`.

The simplest derived class would be:

```
class Mammal(Animal):  
    pass
```

A Mammal object will have all the attributes of an Animal object.

Example

jokerdeck.py

```
from carddeck import CardDeck, Card

JOKER = '\U0001F0CF'

class JokerDeck(CardDeck):

    def __init__(self):
        super().__init__()
        for _ in range(2):
            card = Card(JOKER, JOKER)
            self._cards.append(card)
```

Example

play_cards_jokers.py

```
from jokerdeck import JokerDeck

deck = JokerDeck("Ozzy")

deck.shuffle()

for _ in range(10):
    card = deck.draw()
    print(card)
print()

print(deck)

print(deck.cards)
```

play_cards_jokers.py

```
Q-♦
8-♦
A-♦
K-♦
3-♦
10-♦
8-♦
A-♦
Q-♦
9-♦
```

```
JokerDeck(0zzy,44)
(Card('K', '♦'), Card('8', '♦'), Card('4', '♦'), Card('5', '♦'), Card('6', '♦'),
Card('J', '♦'), Card('5', '♦'), Card('8', '♦'), Card('7', '♦'), Card('5', '♦'),
Card('5', '♦'), Card('6', '♦'), Card('K', '♦'), Card('2', '♦'), Card('A', '♦'),
Card('9', '♦'), Card('10', '♦'), Card('6', '♦'), Card('A', '♦'), Card('10', '♦'),
Card('J', '♦'), Card('4', '♦'), Card('10', '♦'), Card('Q', '♦'), Card('3', '♦'),
Card('J', '♦'), Card('2', '♦'), Card('J', '♦'), Card('9', '♦'), Card('2', '♦'),
Card('4', '♦'), Card('7', '♦'), Card('4', '♦'), Card('2', '♦'), Card('K', '♦'),
Card('7', '♦'), Card('Q', '♦'), Card('2', '♦'), Card('3', '♦'), Card('9', '♦'),
Card('7', '♦'), Card('3', '♦'), Card('6', '♦'), Card('2', '♦'))
```

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 22. Object-oriented Nomenclature

Term	Description
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 @classmethod decorator.
constructor	A method that initializes an instance. The constructor receives any arguments passed into the class name.
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 self, 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 @staticmethod decorator.
superclass	Same as base class

Chapter 12 Exercises

Exercise 12-1 (knight.py, knight_info.py)

Part A

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 data from **knight.txt** to initialize the object.

The object should have the following read-only properties:

name
title
favorite_color
quest
comment

Example

```
from knight import Knight  
  
k = Knight("Arthur")  
print k.favorite_color
```

Part B

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

knight_info.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: AARRRRRRGGGGHH
```

Part C

FOR ADVANCED STUDENTS

Add a `joust()` method to the `Knight` class. This method should accept another knight as its argument, and then calculate a random integer for each knight. Whichever knight has the highest value "wins". The `joust()` method should return the winner. If the "score" is the same, the knight from which `.joust()` is called wins.

Example

```
k1 = Knight('Arthur')  
k2 = Knight('Lancelot')  
result = k1.joust(k2)  
print(f"{result.name} wins!")
```

Chapter 13: Sorting

Objectives

- Sorting lists and dictionaries
- Sorting on alternate keys
- Using lambda functions
- Reversing lists

Sorting Overview

- Get a sorted copy of any sequence
- Iterables of iterables sorted item-by-item
- Sort can be customized

It is typically useful to be able to sort a collection of data. You can get a sorted copy of lists, tuples, and dictionaries.

The python sort routines sort strings character by character and sorts numbers numerically. Mixed types cannot be sorted, since there is no valid greater than/less than comparison between different types. That is, if `s` is a string and `n` is a number, `s < n` will raise an exception.

The sort order can be customized by providing a *callback* function to provide one or more sort keys.

What can you sort?

- list elements
- tuple elements
- string elements
- dictionary key/value pairs
- set elements

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`

```
"""Basic sorting example"""

fruits = ["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(fruits) # sorted() returns a list

print(sorted_fruit)
```

`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 — it will be called with one element of the sequence being sorted. It must 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.

`str.lower` can be used to ignore case when sorting strings.

```
sorted_strings = sorted(unsorted_strings, key=str.lower)
```

Example

custom_sort_keys.py

```
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): # Parameter is _one_ element of iterable to be sorted
    return item.lower() # Return value to sort on

fs1 = sorted(fruit, key=ignore_case) # Specify function with named parameter key
print("Ignoring case:")
print(f"fs1: {fs1}\n")

def by_length_then_name(item):
    return (len(item), item.lower()) # Key functions can return tuple of values to
compare, in order

fs2 = sorted(fruit, key=by_length_then_name)
print("By length, then name:")
print(f"fs2: {fs2}\n")

nums = [800, 80, 1000, 32, 255, 400, 5, 5000]

n1 = sorted(nums) # Numbers sort numerically by default
print("Numbers sorted numerically:")
print(f"n1: {n1}\n")

n2 = sorted(nums, key=str) # Sort numbers as strings
print("Numbers sorted as strings:")
print(f"n2: {n2}\n")
```

custom_sort_keys.py

Ignoring case:

```
fs1: ['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:

```
fs2: ['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:

```
n1: [5, 32, 80, 255, 400, 800, 1000, 5000]
```

Numbers sorted as strings:

```
n2: [1000, 255, 32, 400, 5, 5000, 80, 800]
```

Example

sort_holmes.py

```
"""Sort titles, ignoring leading articles"""
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",
]

def strip_article(title): # create function which takes element to compare and returns
    comparison key
    title = title.lower()
    for article in 'a ', 'an ', 'the ':
        if title.startswith(article):
            title = title.removeprefix(article) # remove article
            break
    return title

for book in sorted(books, key=strip_article): # sort using custom function
    print(book)
```

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

- Inline (anonymous) function
- Creates function on-the-fly
- Body is expression
- Any number of parameters
 - One parameter for sorting

A **lambda function** is a shortcut for defining a function. The syntax is

```
lambda parameters: expression
```

The body of a lambda is restricted to being a valid Python expression; block statements and assignments are not allowed.

Using a lambda for sorting is like using a normally defined function. The lambda function should take a single argument, and returns the comparison value(s).

The expression returned can be a tuple containing multiple keys, in the order in which they should be used.

The following can be used as a template:

```
lambda e: expression
```

Example

lambda_sort.py

```
fruit = ["pomegranate", "cherry", "apricot", "date", "Apple",
         "lemon", "Kiwi", "ORANGE", "lime", "Watermelon", "guava",
         "papaya", "FIG", "pear", "banana", "Tamarind", "persimmon",
         "elderberry", "peach", "BLUEberry", "lychee", "grape"]

nums = [800, 80, 1000, 32, 255, 400, 5, 5000]

fs1 = sorted(fruit, key=lambda e: e.lower()) # lambda returns key function that converts
each element to lower case
print("Ignoring case:")
print(' '.join(fs1))
print()

fs2 = sorted(fruit, key=lambda e: (len(e), e.lower())) # lambda returns tuple
print("By length, then name:")
print(' '.join(fs2))
print()

fs3 = sorted(nums)
print("Numbers sorted numerically:")
for n in fs3:
    print(n, end=' ')
print()
print()
```

lambda_sort.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

Sorting nested data

- Collections sorted item-by-item
- Only same kind of items can be compared

You can sort a collection of collections, for instance a list of tuples. For each tuple, `sorted()` will compare the first element of the tuple, then the second, and so forth.

All of the items in the collection must be the same — they all must be tuples, or lists, or dicts, or strings, or anything else.

Use a lambda function, and index each element as necessary. To sort a list of tuples by the third element of each tuple, use

```
sorted_list = sorted(unsorted_list, key=lambda e: e[2])
```

Example

nested_sort.py

```
computer_people = [
    ('Melinda', 'Gates', 'Gates Foundation', '1964-08-15'),
    ('Steve', 'Jobs', 'Apple', '1955-02-24'),
    ('Larry', 'Wall', 'Perl', '1954-09-27'),
    ('Paul', 'Allen', 'Microsoft', '1953-01-21'),
    ('Larry', 'Ellison', 'Oracle', '1944-08-17'),
    ('Bill', 'Gates', 'Microsoft', '1955-10-28'),
    ('Mark', 'Zuckerberg', 'Facebook', '1984-05-14'),
    ('Sergey', 'Brin', 'Google', '1973-08-21'),
    ('Larry', 'Page', 'Google', '1973-03-26'),
    ('Linus', 'Torvalds', 'Linux', '1969-12-28'),
]

# sort by first name (default)
for first_name, last_name, organization, dob in sorted(computer_people):
    print(first_name, last_name, organization, dob)
print('-' * 60)

# sort by last name
for first_name, last_name, organization, dob in sorted(computer_people, key=lambda e: e[1]): # Select element of nested tuple for sorting
    print(first_name, last_name, organization, dob)
print('-' * 60)

# sort by company
for first_name, last_name, organization, dob in sorted(computer_people, key=lambda e: e[2]): # Select different element of nested tuple for sorting
    print(first_name, last_name, organization, dob)
```

nested_sort.py

```
Bill Gates Microsoft 1955-10-28
Larry Ellison Oracle 1944-08-17
Larry Page Google 1973-03-26
Larry Wall Perl 1954-09-27
Linus Torvalds Linux 1969-12-28
Mark Zuckerberg Facebook 1984-05-14
Melinda Gates Gates Foundation 1964-08-15
Paul Allen Microsoft 1953-01-21
Sergey Brin Google 1973-08-21
Steve Jobs Apple 1955-02-24
```

```
Paul Allen Microsoft 1953-01-21
Sergey Brin Google 1973-08-21
Larry Ellison Oracle 1944-08-17
Melinda Gates Gates Foundation 1964-08-15
Bill Gates Microsoft 1955-10-28
Steve Jobs Apple 1955-02-24
Larry Page Google 1973-03-26
Linus Torvalds Linux 1969-12-28
Larry Wall Perl 1954-09-27
Mark Zuckerberg Facebook 1984-05-14
```

```
Steve Jobs Apple 1955-02-24
Mark Zuckerberg Facebook 1984-05-14
Melinda Gates Gates Foundation 1964-08-15
Sergey Brin Google 1973-08-21
Larry Page Google 1973-03-26
Linus Torvalds Linux 1969-12-28
Paul Allen Microsoft 1953-01-21
Bill Gates Microsoft 1955-10-28
Larry Ellison Oracle 1944-08-17
Larry Wall Perl 1954-09-27
```

Sorting dictionaries

- Use `dict.items()`
- By default, sorts by key
- Provide callback to sort by value

While a dictionary can't be sorted in place, you can iterate over the dictionary in sorted order by sorting the keys, the values, or both.

`dict.items()` returns a list of *key,value* tuples. Since the key is first in each pair, sorting `dict.items()` will default to sorting by keys. Use a defined or lambda function to specify the 2nd element of the *key,value* tuple to sort by values.

Sorting `dict.items()` is really just sorting a list of tuples.

Example

`sorting_dicts.py`

```
colors = dict(red=5, scarlet=18, blue=1, pink=0, grey=27, yellow=5, green=18)

# sort by key
for color, num in sorted(colors.items()): # No sort key function needed to sort by key
    print(color, num)

print()

# sort by value
def by_value(item):
    return item[1], item[0] # sort first by key, then by value

for color, num in sorted(colors.items(), key=by_value):
    print(color, num)
```

sorting_dicts.py

```
blue 1
green 18
grey 27
pink 0
red 5
scarlet 18
yellow 5

pink 0
blue 1
red 5
yellow 5
green 18
scarlet 18
grey 27
```

Sorting lists in place

- Use `list.sort()`
- Only for lists (not strings or tuples)

To sort a list in place, use the list's `.sort()` method. It works exactly like `sorted()`, except that the sort changes the order of the items in the list, and does not make a copy.

Example

`sort_in_place.py`

```
fruits = ['pomegranate', 'cherry', 'apricot', 'Apple',
'lemon', 'Kiwi', 'ORANGE', 'lime', 'Watermelon', 'guava',
'Papaya', 'FIG', 'pear', 'banana', 'Tamarind', 'Persimmon',
'elderberry', 'peach', 'BLUEberry', 'lychee', 'GRAPE', 'date' ]

fruits.sort(key=str.lower) # List is sorted in place; cannot be undone

print(f"fruits: {fruits}")
```

`sort_in_place.py`

```
fruits: ['Apple', 'apricot', 'banana', 'BLUEberry', 'cherry', 'date', 'elderberry',
'FIG', 'GRAPE', 'guava', 'Kiwi', 'lemon', 'lime', 'lychee', 'ORANGE', 'Papaya', 'peach',
'pear', 'Persimmon', 'pomegranate', 'Tamarind', 'Watermelon']
```

Sorting in reverse

- Use `reverse=True`

To sort in reverse, add the `reverse` parameter to `sorted()` or `list.sort()` with a true value (e.g. `True`).

Example

`reverse_sort.py`

```
fruits = ["pomegranate", "cherry", "apricot", "date", "Apple",
          "lemon", "Kiwi", "ORANGE", "lime", "Watermelon", "guava",
          "papaya", "FIG", "pear", "banana", "Tamarind", "persimmon",
          "elderberry", "peach", "BLUEberry", "lychee", "grape"]

print("reverse, case-sensitive:")
sorted_fruits = sorted(fruits, reverse=True) # Set reverse to True to reverse sort
print(" ".join(sorted_fruits))
print()

print("reverse, case-insensitive:")
sorted_fruits = sorted(fruits, reverse=True, key=lambda e: e.lower()) # reverse can be
# combined with key functions
print(" ".join(sorted_fruits))
print()
```

reverse_sort.py

reverse, case-sensitive:

pomegranate persimmon pear peach papaya lychee lime lemon guava grape elderberry date
cherry banana apricot Watermelon Tamarind ORANGE Kiwi FIG BLUEberry Apple

reverse, case-insensitive:

Watermelon Tamarind pomegranate persimmon pear peach papaya ORANGE lychee lime lemon Kiwi
guava grape FIG elderberry date cherry BLUEberry banana apricot Apple

Chapter 13 Exercises

Exercise 13-1 (scores_by_score.py)

Redo `scores.py`, printing out the students in descending order by score.



You will not need to change anything in `scores.py` other than the loop that prints out the names and scores.

Exercise 13-2 (alt_sorted.py)

Read in the file `alt.txt`. Put all the words that start with 'a' in to a file named `a_sorted.txt`, in sorted order. Put all the words that start with 'b' in `b_sorted.txt`, in reverse sorted order.



Read through the file once, putting lines into two lists.

Exercise 13-3 (sort_fruit.py)

Using the file `fruit.txt` in the DATA folder, print it out:

- sorted by name case-sensitively (the default)
- sorted by name case-insensitively (ignoring case)
- sorted by length of name, then by name
- sorted by the 2nd letter of the name, then the first letter

Exercise 13-4 (sort_presidents.py)

Using the file `presidents.txt`, print out the presidents' first name, last name, and state of birth, sorted by last name, then first name.



Use the `split()` method on each line to get the individual fields.

Chapter 14: Regular Expressions

Objectives

- Using RE patterns
- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding option flags to a pattern
- Specifying capture groups
- Replacing text with backrefs and callbacks

Regular expressions

- Specialized language for pattern matching
- Begun in UNIX; expanded by **Perl**
- Python adds some conveniences

Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as Does this string match the pattern?", or Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.

— Python Regular Expression HOWTO

Regular expressions were first popularized thirty years ago as part of Unix text processing programs such as **vi**, **sed**, and **awk**. While they were improved incrementally over the years, it was not until the advent of **Perl** that they substantially changed from the originals. **Perl** added extensions of several different kinds – shortcuts for common sequences, look-ahead and look-behind assertions, non-greedy repeat counts, and a general syntax for embedding special constructs within the regular expression itself.

Python uses **Perl**-style regular expressions (AKA PCREs) and adds a few extensions of its own.

Two good web sites for creating and deciphering regular expressions:

Regex 101: <https://regex101.com/#python>

Pythex: <http://www.pythex.org/>

Two sites for having fun (yes, really!) with regular expressions:

Regex Golf: <https://alf.nu/RegexGolf>

Regex Crosswords: <https://regextcrossword.com/>

RE syntax overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more *branches* separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of *atoms*. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a *quantifier* (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.



There is frequently only one branch.

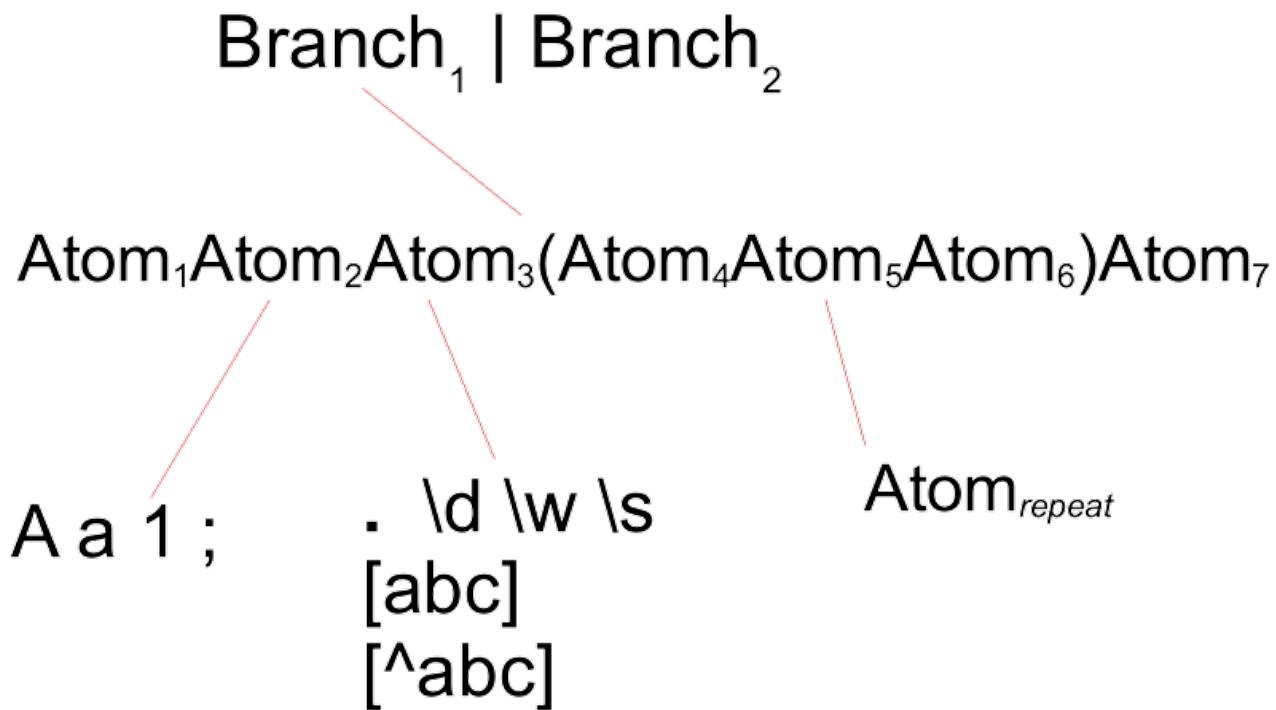


Table 23. Regular Expression Metacharacters

Pattern	Description
.	any character
[abc]	any character in set
[^abc]	any character not in set
\w, \W	any word, non-word char
\d, \D	any digit, non-digit
\s, \S	any space, non-space char
^, \$	beginning, end of string
\b	beginning or end of word
\	escape a special character
*, +, ?	0 or more, 1 or more, 0 or 1
{m}	exactly m occurrences
{m,}	at least m occurrences
{m, n}	m through n occurrences
a b	match a or b
(?aiLmsux)	Set the A, I, L, M, S, U, or X flag for the RE (see below).
(?:...)	Non-capturing version of regular parentheses.
(?P<name>...)	The substring matched by the group is accessible by name.
(?P=name)	Matches the text matched earlier by the group named name.
(?#...)	A comment; ignored.
(?=...)	Matches if ... matches next, but doesn't consume the string.
(?!...)	Matches if ... doesn't match next.
(?=...)	Matches if preceded by ... (must be fixed length).
(?<!...)	Matches if not preceded by ... (must be fixed length).

Finding matches

- Module defines static functions
- Arguments: pattern, string

There are three primary methods for finding matches.

Find first match

```
re.search(pattern, string)
```

Searches `s` and returns the first match. Returns a match object (**SRE_Match**) on success or **None** on failure. A match object is always evaluated as **True**, and so can be used in **if** statements and **while** loops. Call the **group()** method on a match object to get the matched text.

Find all matches

```
re.finditer(pattern, string)
```

Provides a match object for each match found. Normally used with a **for** loop.

Retrieve text of all matches

```
re.findall(pattern, string)
```

Finds all matches and returns a list of matched strings. Since regular expressions generally contain many backslashes, it is usual to specify the pattern with a raw string.

Other search methods

`re.match()` is like `re.search()`, but searches for the pattern at beginning of `s`. There is an implied `^` at the beginning of the pattern.

Likewise `re.fullmatch()` only succeeds if the pattern matches the entire string. `^` and `$` around the pattern are implied.

Example

regex_finding_matches.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}' # store pattern in raw string

if re.search(pattern, s): # search returns True on match
    print("Found pattern.")
print()

m = re.search(pattern, s) # search actually returns match object
print(m)
if m:
    print("Found:", m.group(0)) # group(0) returns text that was matched by entire
expression (or just m.group())
print()

for m in re.finditer(pattern, s): # iterate over all matches in string:
    print(m.group())
print()

matches = re.findall(pattern, s) # return list of all matches
print("matches:", matches)
```

regex_finding_matches.py

```
Found pattern.
```

```
<re.Match object; span=(12, 17), match='M-302'>
```

```
Found: M-302
```

```
M-302
```

```
H-476
```

```
Q-51
```

```
A-110
```

```
H-332
```

```
Y-45
```

```
matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

RE objects

- `re` object contains a compiled regular expression
- Call methods on the object, with strings as parameters.

An `re` object is created by calling `re.compile()` with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. `re.compile()` has an optional argument for flags which enable special features or fine-tune the match.



It is generally a good practice to create your `re` objects in a location near the top of your script, and then use them as necessary

Example

regex_objects.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]-\d{2,3}') # Create an re (regular expression) object

if rx_code.search(s): # Call search() method from the object
    print("Found pattern.")
print()

m = rx_code.search(s)
if m:
    print("Found:", m.group())
print()

for m in rx_code.finditer(s):
    print(m.group())
print()

matches = rx_code.findall(s)
print("matches:", matches)
```

regex_objects.py

```
Found pattern.
```

```
Found: M-302
```

```
M-302
```

```
H-476
```

```
Q-51
```

```
A-110
```

```
H-332
```

```
Y-45
```

```
matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']
```

Compilation flags

- Fine-tune match
- Add readability

When using functions from `re`, or when compiling a pattern, you can specify various flags to control how the match occurs. The flags are aliases for numeric values, and can be combined with `|`, the bitwise **OR** operator. Each flag has a short form and a long form.

Ignoring case

```
re.I, re.IGNORECASE
```

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match "Spam", "spam", or "spAM". This lower-casing doesn't take the current locale into account; it will if you also set the `LOCALE` flag.

Using the locale

```
re.L, re.LOCALIZE
```

Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]`; it won't match "é" or "ç". If your system is configured properly and a French locale is selected, certain C functions will tell the program that "é" should also be considered a letter. Setting the `LOCALE` flag enables `\w+` to match French words as you'd expect.

Ignoring whitespace

```
re.X, re.VERBOSE
```

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a `#` that's neither in a character class or preceded by an unescaped backslash. Use a triple-quoted string for your pattern to make best advantage of this flag.

```
RE_ENTRY = r"""
(?P<month>[A-Z][a-z]{2})\s+(?P<day>\d+)\s+          # date
(?P<hour>\d{2}):(?P<minute>\d{2}):(?P<second>\d{2})\s+    # timestamp
(?P<hostname>\S+?)\s+                                         # hostname
(?P<process_name>.*?)                                       # process name
\[ (?P<pid>\d+)\]:\s+                                         # PID
(?P<message>.*)
"""

```

Example

regex_flags.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'

if re.search(pattern, s, re.IGNORECASE): # make search case-insensitive
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I | re.M) # short version of flag
if m:
    print("Found:", m.group())
```

```
print()

for m in re.finditer(pattern, s, re.I):
    print(m.group())
print()

matches = re.findall(pattern, s, re.I)
print("matches:", matches)
```

regex_flags.py

```
Found pattern.
```

```
Found: M-302
```

```
M-302
```

```
r-99
```

```
H-476
```

```
Q-51
```

```
z-883
```

```
A-110
```

```
H-332
```

```
Y-45
```

```
matches: ['M-302', 'r-99', 'H-476', 'Q-51', 'z-883', 'A-110', 'H-332', 'Y-45']
```

Working with newlines

- `re.S`, `re.DOTALL` lets `.` match newline
- `re.M`, `re.MULTILINE` lets `^` and `$` match lines

Some text contains newlines (`\n`), representing multiple lines within the string. There are two regular expression flags you can use with `re.search()` and other functions to control how they are searched. These flags are not useful if the string has no embedded newlines.

Treating text like a single string

By default, `.` does not match newline. Thus, `spam.*ham` will not match the text if `spam` is on one line and `ham` is on a subsequent line. The `re.DOTALL` flag allows `.` to match newline, enabling searches that span lines.

`re.S` is an abbreviation for `re.DOTALL`.

Treating text like multiple lines

Normally, `^` only matches the beginning of a string and `$` only matches the end. If you use the `re.MULTILINE` flag, these anchors will also match the beginning and end of embedded lines.

`re.M` is an abbreviation for `re.MULTILINE`.

Example

regex_newlines.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

line_start_word = r'^\w+' # match word at beginning of string/line

matches = re.findall(line_start_word, s) # only matches at beginning of string
print("matches:", matches)
print()

matches = re.findall(line_start_word, s, re.M) # matches at beginning of lines
print("matches:", matches)
print()

phrase = r"aliquip.*commodo"

match = re.search(phrase, s)
if match:
    print(match.group(), match.start())
else:
    print(f"{phrase} not found")
print()

match = re.search(phrase, s, re.S)
if match:
    print(repr(match.group()), match.start())
else:
    print(f"{phrase} not found")
print()
```

regex_newlines.py

```
matches: ['lorem']

matches: ['lorem', 'ad', 'ea', 'voluptate', 'Excepteur', 'officia']

aliquip.*commodo not found

'aliquip ex \nea commodo' 223
```

Groups

- Marked with parentheses
- Capture whatever matched pattern within
- Access with `match.group()`

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a `:`. This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked with parentheses, and "capture" whatever matched the pattern inside the parentheses.

`re.findall()` returns a list of tuples, where each tuple contains the matches for all each group.

To access groups in more detail, use `re.finditer()` and call the `.group()` method on each match object. The default group is 0, which is always the entire match. It can be retrieved with either `match.group(0)`, or just `match.group()`. Then, `match.group(1)` returns text matched by the first set of parentheses, `match.group(2)` returns the text from the second set, etc.

In the same vein, `match.start()` or `match.start(0)` return the beginning 0-based offset of the entire match; `match.start(1)` returns the beginning offset of group 1, and so forth. The same is true for `match.end()` and `match.end(n)`.

`match.span()` returns the the start and end offsets for the entire match. `match.span(1)` returns start and end offsets for group 1, and so forth.

Example

regex_group.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'([A-Z])-(\d{2,3})' # parens delimit groups

print("Group 0           Group 1           Group 2")
header2 = "text  start  end  text  start  end  text  start  end"
print(header2)
print("-" * len(header2))

for m in re.finditer(pattern, s):
    print(
        f"{m.group(0)} {m.start(0)} {m.end(0)}"
        f" {m.group(1)} {m.start(1)} {m.end(1)}"
        f" {m.group(2)} {m.start(2)} {m.end(2)}"
    )
print()

matches = re.findall(pattern, s) # findall() returns list of tuples containing groups
print("matches:", matches)
```

regex_group.py

Group 0			Group 1			Group 2		
text	start	end	text	start	end	text	start	end
<hr/>								
M-302	12	17	M	12	13	302	14	17
H-476	102	107	H	102	103	476	104	107
Q-51	134	138	Q	134	135	51	136	138
A-110	398	403	A	398	399	110	400	403
H-332	436	441	H	436	437	332	438	441
Y-45	470	474	Y	470	471	45	472	474

matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('A', '110'), ('H', '332'), ('Y', '45')]

Special groups

- Non-capture groups are used just for grouping
- Named groups allow retrieval of sub-expressions by name rather than number
- Look-ahead and look-behind match, but do not capture

There are two variations on RE groups that are useful. If the first character inside the group is a question mark, then the parentheses contain some sort of extended pattern, designated by the next character after the question mark.

Non-capture groups

The most basic special is **(?:pattern)**, which groups but does not capture.

Named groups

A welcome addition in Python is the concept of named groups. Instead of remembering that the month is the 3rd group and the year is the 4th group, you can use the syntax **(?'P<name>pattern)**. You can then call `match.group("name")` to fetch the text match by that sub-expression; alternatively, you can call `match.groupdict()`, which returns a dictionary where the keys are the pattern names, and the values are the text matched by each pattern.

Lookaround assertions

Another advanced concept is an assertion, either lookahead or lookbehind. A lookahead assertion uses the syntax **(?=pattern)**. The string being matched must match the lookahead, but does not become part of the overall match.

For instance, `\d(?:st|nd|rd|th)(?=street)` matches "1st", "2nd", etc., but only where they are followed by "street".

Example

regex_special.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])- (?P<number>\d{2,3})' # Use (?P<NAME>...) to name groups

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number')) # Use m.group(NAME) to retrieve text
```

regex_special.py

```
M 302
H 476
Q 51
A 110
H 332
Y 45
```

Replacing text

- Use `re.sub(pattern, replacement, string[, count])`
- `re.subn(...)` returns string and count

To find and replace text using a regular expression, use the `re.sub()` method. It takes the pattern, the replacement text and the string to search as arguments, and returns the modified string.

The third (optional) argument is one or more compilation flags. The fourth argument is the maximum number of replacements to make. Both are optional, but if the fourth argument is specified and no flags are needed, use `0` as a placeholder for the third argument.



Be sure to put the arguments in the proper order!

Example

regex_sub.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
    eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
    officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])- (?P<number>\d{2,3})', re.I)

s2 = rx_code.sub("[REDACTED]", s) # replace pattern with string
print(s2)
print()

s3, count = rx_code.subn("___", s) # subn returns tuple with result string and
# replacement count
print(f"Made {count} replacements")
print()
print(s3)
```

regex_sub.py

```
lorem ipsum [REDACTED] dolor sit amet, consectetur [REDACTED] adipiscing elit, sed do  
eiusmod tempor incididunt [REDACTED] ut labore et dolore magna [REDACTED] aliqua. Ut  
enim  
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex  
ea commodo [REDACTED] consequat. Duis aute irure dolor in reprehenderit in  
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.  
Excepteur sint occaecat [REDACTED] cupidatat non proident, sunt in [REDACTED] culpa qui  
officia deserunt [REDACTED] mollit anim id est laborum
```

Made 8 replacements

```
lorem ipsum ___ dolor sit amet, consectetur ___ adipiscing elit, sed do  
eiusmod tempor incididunt ___ ut labore et dolore magna ___ aliqua. Ut enim  
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex  
ea commodo ___ consequat. Duis aute irure dolor in reprehenderit in  
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.  
Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui  
officia deserunt ___ mollit anim id est laborum
```

Replacing with backrefs

- Use match or groups in replacement text
 - `\g<num>` group number
 - `\g<name>` group name
 - `\num` shortcut for group number

It is common to need all or part of the match in the replacement text. To allow this, the `re` module provides *backrefs*, which are special variables that refer back to the groups in the match, including group 0 (the entire match).

To refer to a particular group, use the syntax `\g<num>`.

To refer to a particular named group, use `\g<name>`.

As a shortcut, you can use `\num`. This does not work for group 0 — use `\g<0>` instead.



`\10` is group 10, not group 1 followed by '0'

Example

regex_sub_backrefs.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])- (?P<number>\d{2,3})', re.I)

s2 = rx_code.sub(r"(\g<1>) [\g<2>]", s)
print(f"s2: {s2}")
print('-' * 60)

s3 = rx_code.sub(r"\g<number>- \g<letter>", s)
print(f"s3: {s3}")
print('-' * 60)

s4 = rx_code.sub(r"[\1:\2]", s)
print(f"s4: {s4}")
print('-' * 60)
```

regex_sub_backrefs.py

```
s2: lorem ipsum (M)[302] dolor sit amet, consectetur (r)[99] adipiscing elit, sed do eiusmod tempor incididunt (H)[476] ut labore et dolore magna (Q)[51] aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo (z)[883] consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
```

```
Excepteur sint occaecat (A)[110] cupidatat non proident, sunt in (H)[332] culpa qui officia deserunt (Y)[45] mollit anim id est laborum
```

```
s3: lorem ipsum 302-M dolor sit amet, consectetur 99-r adipiscing elit, sed do eiusmod tempor incididunt 476-H ut labore et dolore magna 51-Q aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo 883-z consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
```

```
Excepteur sint occaecat 110-A cupidatat non proident, sunt in 332-H culpa qui officia deserunt 45-Y mollit anim id est laborum
```

```
s4: lorem ipsum [M:302] dolor sit amet, consectetur [r:99] adipiscing elit, sed do eiusmod tempor incididunt [H:476] ut labore et dolore magna [Q:51] aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo [z:883] consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
```

```
Excepteur sint occaecat [A:110] cupidatat non proident, sunt in [H:332] culpa qui officia deserunt [Y:45] mollit anim id est laborum
```

Replacing with a callback

- Replacement can be a callback function
- Function expects match object, returns replacement text
- Use either normally defined function or a lambda

In addition to using a string, possibly containing backrefs, as the replacement, you can specify a function. This function will be called once for each match, with the match object as its only parameter.

Using a callback is necessary if you need to modify any of the original text.

Whatever string the function returns will be used as the replacement text. This lets you have complete control over the replacement.

Using a callback makes it simple to:

- add text around the replacement (can also be done with backrefs)
- search ignoring case and preserve case in a replacement
- look up the text in a dictionary or database to find replacement text

Example

regex_sub_callback.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui officia deserunt Y-45 mollit anim id est dlaborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])- (?P<number>\d{2,3})', re.I)

def update_code(m): # callback function is passed each match object
    letter = m.group('letter').upper()
    number = int(m.group('number'))
    return f'{letter}:{number:04d}' # function returns replacement text

s2, count = rx_code.subn(update_code, s) # sub takes callback function instead of replacement text
print(s2)
print(count, "replacements made")
```

regex_sub_callback.py

```
lorem ipsum M:0302 dolor sit amet, consectetur R:0099 adipiscing elit, sed do eiusmod tempor incididunt H:0476 ut labore et dolore magna Q:0051 aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo Z:0883 consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A:0110 cupidatat non proident, sunt in H:0332 culpa qui officia deserunt Y:0045 mollit anim id est dlaborum
8 replacements made
```

Splitting a string

- Syntax: `re.split(pattern, string[,max])`

The `re.split()` method splits a string into pieces, using the regex to match the delimiters, and returning the pieces as a list. The optional `max` argument limits the numbers of pieces.

Example

`regex_split.py`

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
    eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
    officia deserunt Y-45 mollit anim id est dlaborum"""

# pattern is one or more non-letters
rx_wordsep = re.compile(r"[^a-zA-Z-]+", re.I) # When splitting, pattern matches what you
don't want

words = rx_wordsep.split(s) # Retrieve text _separated_ by your pattern
unique_words = set(words)

print(sorted(unique_words))
```

regex_split.py

```
['A-110', 'Duis', 'Excepteur', 'H-332', 'H-476', 'M-302', 'Q-51', 'U901', 'Ut', 'Y-45',  
'ad', 'adipiscin', 'aliqua', 'aliquip', 'amet', 'anim', 'aute', 'cillum', 'commodo',  
'consectetur', 'consequat', 'culpa', 'cupidatat', 'deserunt', 'dlaborum', 'do', 'dolor',  
'dolore', 'ea', 'eiusmod', 'elit', 'enim', 'esse', 'est', 'et', 'eu', 'ex',  
'exercitation', 'fugiat', 'id', 'in', 'incididunt', 'ipsum', 'irure', 'labore',  
'laboris', 'lorem', 'magna', 'minim', 'mollit', 'nisi', 'non', 'nostrud', 'nulla',  
'occaecat', 'officia', 'pariatur', 'proident', 'qui', 'quis', 'r-99', 'reprehenderit',  
'sed', 'sint', 'sit', 'sunt', 'tempor', 'ullamco', 'ut', 'velit', 'veniam', 'voluptate',  
'z-883']
```

Chapter 14 Exercises

Exercise 14-1 (pyfind.py)

Write a script which takes two or more arguments. The first argument is the pattern to search for; the remaining arguments are files to search. For each file, print out all lines which match the pattern. [1]

Example

```
python pyfind.py freezer DATA/words.txt DATA/parrot.txt
```

Exercise 14-2 (mark_big_words_callback.py, mark_big_words_backrefs.py)

Copy `parrot.txt` to `bigwords.txt` adding asterisks around all words that are 8 or more characters long.

HINT: Use the `\b` anchor to indicate beginning or end of a word.



There are two solutions to this exercise in the ANSWERS folder: one using a callback function, and one using backrefs.

Exercise 14-3 (print_numbers.py)

Write a script to print out all lines in `custinfo.dat` which contain phone numbers. A phone number consists of three digits, a dash, and four more digits.

Exercise 14-4 (word_freq.py)

Write a script that will read a text file and print out a list of all the words in the file, normalized to lower case, and with the number of times that word occurred in the file. Use the regular expression `[^\w']+` for splitting each line into words.

Test with any of the text files in the DATA folder.

HINT: Use a dictionary for counting.



The specified pattern matches one or more characters that are neither letters, digits, underscores, nor apostrophes.

[1] Any similarity to the Unix `grep` command is purely intentional.

Chapter 15: 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
- Creating and using date objects
- Fetching data from a URL
- Sending an email
- 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

- Three file objects open on the console
 - `stdin` *reading*
 - `stdout` *writing*
 - `stderr` *writing*

The `sys` module defines three file objects representing the three streams of STDIO, AKA "standard I/O".

Unless they have been redirected, `sys.stdin` is the keyboard, and `sys.stdout` and `sys.stderr` are the console screen. You can use `sys.stderr` for error messages.

Example

`stdio.py`

```
import sys
sys.stdout.write("Hello, world\n")
sys.stderr.write("Error message here...\n")
```

`stdio.py`

```
Hello, world
```

`stdio.py 2>spam.txt`

```
Hello, world
```

`spam.txt`

```
Hello, world
Error message here...
```

Launching external programs

- Different ways to launch programs
 - Just launch (`os.system()`)
 - Capture output (`os.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

```
import os

os.system("hostname") # Just run "hostname"

with os.popen('netstat -an') as netstat_in: # Open command line "netstat -an" as a file-like object
    for entry in netstat_in: # Iterate over lines in output of "netstat -an"
        if 'ESTAB' in entry: # Check to see if line contains "ESTAB"
            print(entry, end='')

print()
```

external_programs.py

```
Johns-Macbook.attlocal.net
tcp4      0      0 192.168.1.242.56764  108.60.71.49.443  ESTABLISHED
tcp4      0      0 192.168.1.242.56761  34.117.237.239.443  ESTABLISHED
tcp4      0      0 192.168.1.242.56760  54.152.80.2.443  ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56759  2607:f8b0:402a:8.443  ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56721  2a03:2880:f211:1.443  ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56720  2a03:2880:f011:1.443  ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56716  2a03:2880:f011:7.443  ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56715  2a03:2880:f011:1.443  ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56713  2a03:2880:f011:1.443  ESTABLISHED
tcp6      0      0 ::1.8888          ::1.56711          ESTABLISHED
tcp6      0      0 ::1.56711          ::1.8888          ESTABLISHED
tcp6      0      0 2600:1700:3901:6.56696  2607:f8b0:4002:c.443  ESTABLISHED
tcp6      0      0 ::1.8888          ::1.56687          ESTABLISHED
tcp6      0      0 ::1.56687          ::1.8888          ESTABLISHED
```

Paths, directories and filenames

- import `os` module to get `os.path`
- functions for working with paths.

`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.

Common functions

- `os.path.exists()`
- `os.path.dirname()`
- `os.path.basename`
- `os.path.split()`

Example

paths.py

```
import sys
import os.path

unix_rel = "bin/spam.txt" # Unix relative path
unix_abs = "/usr/local/bin/ham" # Unix absolute path

win_rel = r"spam\ham.doc" # Windows relative path
win_unc = r"\\spam\ham\eggs\toast\jam.doc" # Windows UNC path

if sys.platform == 'win32': # Are we on Windows?
    print("win_rel:", win_rel)
    print("win_unc:", win_unc)
    print("dirname(win_rel):", os.path.dirname(win_rel)) # Just the folder name
    print("dirname(win_unc):", os.path.dirname(win_unc))
    print("basename(win_rel):", os.path.basename(win_rel)) # Just the file (or folder)
name
    print("basename(win_unc):", os.path.basename(win_unc))
    print("isabs(win_rel):", os.path.isabs(win_rel)) # Is it an absolute path?
    print("isabs(win_unc):", os.path.isabs(win_unc))
else: # Mac/Linux
    print("unix_rel:", unix_rel)
    print("unix_abs:", unix_abs)
    print("dirname(unix_rel):", os.path.dirname(unix_rel)) # Just the folder name
    print("dirname(unix_abs):", os.path.dirname(unix_abs))
    print("basename(unix_rel):", os.path.basename(unix_rel)) # Just the file (or folder)
name
    print("basename(unix_abs):", os.path.basename(unix_abs))
    print("isabs(unix_rel):", os.path.isabs(unix_rel)) # Is it an absolute path?
    print("isabs(unix_abs):", os.path.isabs(unix_abs))
    print(f"os.path.expanduser('~'):{os.path.expanduser('~')}")"
    print(f"os.path.expanduser('~root'):{os.path.expanduser('~root')}")"
```

paths.py

```
unix_rel: bin/spam.txt
unix_abs: /usr/local/bin/ham
dirname(unix_rel): bin
dirname(unix_abs): /usr/local/bin
basename(unix_rel): spam.txt
basename(unix_abs): ham
isabs(unix_rel): False
isabs(unix_abs): True
os.path.expanduser('~'): /Users/jstrick
os.path.expanduser('~root'): /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 `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:

- Absolute or relative path to the directory based on stating directory
- List of subdirectories
- List of files

Example

os_walk.py

```
# 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': # check platform
    target = 'C:/Users'
else:
    target = '/etc'

total_files = 0 # initialize counters
total_dirs = 0

for currdir, subdirs, files in os.walk(target): # visit target folder and each
    subfolder; os.walk() returns 3-tuple at each folder
    total_dirs += 1 # increment number of directories seen
    total_files += len(files) # add the number of files in this dir

print(f"{target} contains {total_dirs} dirs and {total_files} files") # output results
```

os_walk.py

```
/etc contains 39 dirs and 330 files
```

Example

os_walk2.py

```
"""
    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: # make sure there is at least one command line argument
    print('Syntax: walk2.py START-DIR')
    sys.exit(1)

for currdir, subdirs, files in os.walk(sys.argv[1]):
    for file in files: # in each folder, iterate over files
        fullpath = os.path.join(currdir, file) # get full path to file
        if os.path.isfile(fullpath): # make sure it's a file (not a link or device, etc.)
            fsize = os.path.getsize(fullpath) # get file size
            if fsize >= MINIMUM_SIZE: # check size to see if it's at least MINIMUM_SIZE
                print(f"{{fsize:8d} {{fullpath:40s}}")
```

os_walk2.py ..//DATA

```
10820 ../DATA/hyper.xlsx
2984133 ../DATA/gm_facts.json
2583 ../DATA/presidents.csv
147807875 ../DATA/Bicycle_Counts.csv
2150588 ../DATA/WorldCupPlayers.csv
22032 ../DATA/uri-schemes-1.csv
21222 ../DATA/mpg.csv
7048 ../DATA/presidents.html
14875 ../DATA/presidents.xlsx
40832 ../DATA/pokemon_data.csv
```

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).

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.



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`

```
import urllib.request

response = urllib.request.urlopen("https://www.python.org")

print(response.info()) # .info() returns a dictionary of HTTP headers
print()

print(response.read(500).decode()) # The text is returned as a bytes object, so it
needs to be decoded to a string
```

On some systems, you may have to update your SSL certificates to use `urllib`.

Try



```
pip install --upgrade certifi
```

On MacOS, try this: * Press "command + space" button or open Spotlight * type "Install Certificates.command"

read_html_urllib.py | head -15

```
Connection: close
Content-Length: 51180
Report-To: {"group":"heroku-
nel","max_age":3600,"endpoints":[{"url":"https://nel.herokuapp.com/reports?ts=1709218001&sid=
=67ff5de4-ad2b-4112-9289-
cf96be89efed&s=jM2ait13ri00r%2Fevt0oktmFStp46EzB4lJcM59gH6FE%3D"}]}
Reporting-Endpoints: heroku-
nel=https://nel.herokuapp.com/reports?ts=1709218001&sid=67ff5de4-ad2b-4112-9289-
cf96be89efed&s=jM2ait13ri00r%2Fevt0oktmFStp46EzB4lJcM59gH6FE%3D
Nel: {"report_to":"heroku-
nel","max_age":3600,"success_fraction":0.005,"failure_fraction":0.05,"response_headers":[
"Via"]}
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: SAMEORIGIN
Via: 1.1 vegur, 1.1 varnish, 1.1 varnish
Accept-Ranges: bytes
Date: Thu, 29 Feb 2024 15:15:24 GMT
Age: 1440
X-Served-By: cache-iad-kiad700025-IAD, cache-pdk-kpdk1780131-PDK
X-Cache: HIT, HIT
X-Cache-Hits: 46, 2
```

...

Example

read_pdf_urllib.py

```
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/wp-content/uploads/2021/12/sls_fact_sheet.pdf' # target URL
saved_pdf_file = 'nasa.pdf' # name of PDF file for saving
file = 'nasa_iss.pdf' # name of PDF file for saving

try:
    URL = urlopen(url) # open the URL
except HTTPError as e: # catch any HTTP errors
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read() # read all data from URL in binary mode
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents) # write data to a local file in binary mode

if sys.platform == 'win32': # select platform and choose the app to open the PDF file
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd) # launch the app
```

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.



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

```
from getpass import getpass # module for hiding password
import smtplib # module for sending email
from email.message import EmailMessage # module for creating message
from datetime import datetime

TIMESTAMP = datetime.now().ctime() # get a time string for the current date/time

SENDER = 'jstrickler@gmail.com'
RECIPIENTS = ['jstrickler@gmail.com']
MESSAGE SUBJECT = 'Python SMTP example'

MESSAGE_BODY = f"""
Hello at {TIMESTAMP}.

Testing email from Python
"""

SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:") # get password (not echoed to screen)

smtp = smtplib.SMTP("smtp2go.com", 2525) # connect to SMTP server
smtp.login(SMTP_USER, SMTP_PASSWORD) # log into SMTP server

msg = EmailMessage() # create empty email message
msg.set_content(MESSAGE_BODY) # add the message body
msg['Subject'] = MESSAGE SUBJECT # add the message subject
msg['from'] = SENDER # add the sender address
msg['to'] = RECIPIENTS # add a list of recipients

try:
    smtp.send_message(msg) # send the message
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtp.quit() # disconnect from SMTP server
```

email_attach.py

```
import smtplib
import os
from datetime import datetime
import magic # module to determine image type (install as python-magic)
from email.message import EmailMessage # module for creating email message
from getpass import getpass # module for reading password privately

SMTP_SERVER = "smtp2go.com" # global variables for external information (IRL should be
from environment -- command line, config file, etc.)
SMTP_PORT = 2525

SMTP_USER = 'pythonclass'

SENDER = 'jstrickler@gmail.com'
RECIPIENTS = ['jstrickler@gmail.com']

def main():
    smtp_server = create_smtp_server()
    now = datetime.now()
    msg = create_message(
        SENDER,
        RECIPIENTS,
        'Here is your attachment',
        f'Testing email attachments from python class at {now}\n\n',
    )
    add_text_attachment('../DATA/parrot.txt', msg)
    add_image_attachment('../DATA/felix_auto.jpeg', msg)
    add_image_attachment('../DATA/zen_of_python.pdf', msg)
    send_message(smtp_server, msg)

def create_message(sender, recipients, subject, body):
    msg = EmailMessage() # create instance of EmailMessage to hold message
    msg.set_content(body) # set content (message text) and various headers
    msg['From'] = sender
    msg['To'] = recipients
    msg['Subject'] = subject
    return msg

def add_text_attachment(file_path, message):
    with open(file_path) as file_in: # read data for text attachment
        attachment_data = file_in.read()
```

```
    file_name = os.path.basename(file_path)
    message.add_attachment(attachment_data, filename=file_name) # add text attachment to
message
# if filename not specified, adds text inline

def add_image_attachment(file_path, message):
    with open(file_path, 'rb') as file_in: # read data for binary attachment
        attachment_data = file_in.read()
    mime_type = magic.from_buffer(attachment_data, mime=True)
    main_type, sub_type = mime_type.split(mime_type)
    file_name = os.path.basename(file_path)
    # add binary attachment to message, including type and subtype (e.g., "image/jpg")
    message.add_attachment(attachment_data, filename=file_name, maintype=main_type,
    subtype=sub_type)

def create_smtp_server():
    password = getpass("Enter SMTP server password:") # get password from user (don't
hardcode sensitive data in script)
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT) # create SMTP server connection
    smtpserver.login(SMTP_USER, password) # log into SMTP connection

    return smtpserver

def send_message(server, message):
    try:
        server.send_message(message) # send message
    finally:
        server.quit()

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

Example

email_html.py

```
import smtplib # module for sending email
from email.message import EmailMessage # module for creating email message
from getpass import getpass # module for reading password privately

from datetime import datetime

TIMESTAMP = datetime.now().ctime() # get a time string for the current date/time

SENDER = 'jstrickler@gmail.com'
RECIPIENTS = 'jstrickler@gmail.com'
MESSAGE SUBJECT = 'Python SMTP example'

PLAIN_BODY = f"""
Hello at {TIMESTAMP}.

Testing text-only email from Python
"""

HTML_BODY = f"""
<html>
<head></head>
<body>
<h1>Hello</h1>
<h2>{TIMESTAMP}</h2>
<h3>Testing HTML email from Python</h3>
See more examples of using EmailMessage <a href="https://docs.python.org/3/library/email.examples.html">here</a>
<body>
</html>
"""

SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:") # get password (not echoed to screen)
smtp = smtplib.SMTP("smtp2go.com", 2525) # connect to SMTP server
smtp.login(SMTP_USER, SMTP_PASSWORD) # log into SMTP server

msg = EmailMessage() # create empty email message
msg['Subject'] = MESSAGE SUBJECT # add the message subject
msg['from'] = SENDER # add the sender address
msg['to'] = RECIPIENTS # add a list of recipients
```

```
msg.set_content(PLAIN_BODY)
msg.add_alternative(HTML_BODY, subtype='html')

try:
    smtp.sendmail(SENDER, RECIPIENTS, msg.as_string()) # send the message
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtp.quit() # disconnect from SMTP server
```

math functions

- use the math module
- Provides functions and constants

Python provides many math functions. It also provides constants pi and e.

Table 24. Math functions

<code>sqrt(x)</code>	Returns the square root of x
<code>exp(x)</code>	Return ex
<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
<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



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 `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

```
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(10):
    print("random():", random.random())
    print("randint(1, 2000):", random.randint(1, 2000))
    print("randrange(1, 5):", random.randrange(1, 10))
    print("choice(fruit):", random.choice(fruits))
    print("sample(fruit, 3):", random.sample(fruits, 3))
    print()
```

random_ex.py

```
random(): 0.888234902031628
randint(1, 2000): 1163
randrange(1, 5): 4
choice(fruit): pear
sample(fruit, 3): ['papaya', 'grape', 'orange']

random(): 0.6130407498070838
randint(1, 2000): 590
randrange(1, 5): 9
choice(fruit): banana
sample(fruit, 3): ['kiwi', 'apricot', 'orange']

random(): 0.8690537966013343
randint(1, 2000): 553
randrange(1, 5): 4
choice(fruit): fig
sample(fruit, 3): ['cherry', 'lemon', 'kiwi']

random(): 0.7964245473569109
randint(1, 2000): 776
randrange(1, 5): 1
choice(fruit): guava
sample(fruit, 3): ['banana', 'kiwi', 'apple']

random(): 0.46740855934954795
randint(1, 2000): 1581
randrange(1, 5): 8
choice(fruit): elderberry
sample(fruit, 3): ['fig', 'pear', 'watermelon']
```

Dates and times

- Use `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

```
from datetime import datetime, date, timedelta

today = date.today()
print(f"today: {today}")
print(f"type(today): {type(today)}")
print(f"today.month: {today.month}")
print(f"today.day: {today.day}")
print(f"today.year: {today.year}")
print()

now = datetime.now() # get today's date and time
print(f"now: {now}")
print(f"now.year: {now.year}")
print(f"now.month: {now.month}")
print(f"now.day: {now.day}")
print(f"now.minute: {now.minute}")
print(f"now.second: {now.second}")
print(f"now.microsecond: {now.microsecond}")
print()

d1 = datetime(2018, 6, 13, 4, 55, 27, 8082) # create a date object
d2 = datetime(2018, 8, 24)
delta = d2 - d1 # date objects can be subtracted from other date objects

print(f"delta: {delta}") # timedelta has days, seconds, and microseconds
print(f"delta.days: {delta.days}")
```

datetime_ex.py

```
today: 2024-02-29
type(today): <class 'datetime.date'>
today.month: 2
today.day: 29
today.year: 2024
```

```
now: 2024-02-29 10:15:25.005719
now.year: 2024
now.month: 2
now.day: 29
now.minute: 15
now.second: 25
now.microsecond: 5719
```

```
delta: 71 days, 19:04:32.991918
delta.days: 71
```

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, bzipped, and compressed archives.

Example

`zipfile_read.py`

```
from zipfile import ZipFile

# read & extract from zip file
zip_in = ZipFile("../DATA/textfiles.zip") # Open zip file for reading
print(zip_in.namelist()) # Print list of members in zip file
tyger_text = zip_in.read('tyger.txt').decode() # Read (raw binary) data from member and
convert from bytes to string
print(tyger_text[:100], '\n')
zip_in.extract('parrot.txt') # Extract member
```

`zipfile_read.py`

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']
The Tyger
```

```
Tyger! Tyger! burning bright
In the forests of the night,
What immortal hand o
```

zipfile_write.py

```
from zipfile import ZipFile, ZIP_DEFLATED
import os.path

file_names = ["parrot.txt", "tyger.txt", "knights.txt", "alice.txt", "poe_sonnet.txt",
"spam.txt"]
file_folder = ".../DATA"

# creating new, empty, zip file
zip_out = ZipFile("example.zip", mode="w", compression=ZIP_DEFLATED) # Create new zip
file

# add files to zip file
for file_name in file_names:
    file_path = os.path.join(file_folder, file_name)
    zip_out.write(file_path, file_name) # Add member to zip file
```

Chapter 15 Exercises

Exercise 15-1 (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 15-2 (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.



You will need the `os.path` module.

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](#)

Just getting started

Here are some resources that can help you get started learning how to code.

- [Code Newbie Podcast](#)
- [Dive into Python3](#)
- [Learn Python the Hard Way](#)
- [Learn Python the Hard Way](#)
- [Automate the Boring Stuff with Python](#)
- [Automate the Boring Stuff with Python](#)

So you want to be a data scientist?

- [Data Wrangling with Python](#)
- [Data Analysis in Python](#)
- [Titanic: Machine Learning from Disaster](#)
- [Deep Learning with Python](#)
- [How to do X with Python](#)
- [Machine Learning: A Probabilistic Prospective](#)

So you want to write code for the web?

- [Learn flask, some great resources are listed here](#)
- [Django Polls Tutorial](#)
- [Hello Web App](#)
- [Hello Web App Intermediate](#)
- [Test-Driven-Development for Web Programming](#)
- [2 Scoops of Django](#)

- [HTML and CSS: Design and Build Websites](#)
- [JavaScript and JQuery](#)

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](#)
- [Base CS Podcast](#)
- [Writing Idiomatic Python](#)
- [Fluent Python](#)
- [Pro Python](#)
- [Refactoring](#)
- [Clean Code](#)
- [Write music with Python, since that's my favorite way to learn a new language](#)

Appendix B: Python Bibliography

Data Science

- ***Building machine learning systems with Python.*** William 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

- ***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
- **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.TJ 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
- **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 3.X: Best Practices for the Django Web Framework.***Daniel Roy Greenfeld, Audrey Roy Greenfeld*.Two Scoops Press
- **Web Development with Django Cookbook.***Aidas Bendoraitis*.Packt Publishing

Appendix C: An Overview of Python

What is Python?

- All-purpose interpreted language
- Created by Guido van Rossum
- First released (ver. 0.9) February 20, 1991

Python is an open-source, all-purpose programming language.

Python was created by Guido van Rossum beginning in 1989. He was involved with the development of Amoeba, a distributed operating system, and had previously worked on ABC, a scripting language designed to be easier to learn for non-programmers.

Van Rossum took ABC and improved it, adding new features, some of which came from other languages such as Perl and Lisp. His design goal was

to serve as a second language for people who were C or C++ programmers, but who had work where writing a C program was just not effective.

The first public release was version 0.9 (beta) in 1991.

Guido van Rossum



The Birth of Python

About the origin of Python, Van Rossum wrote in 1996:

Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office ... would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus). (Introduction to Programming Python, by Mark Lutz, published by O'Reilly)

He says, "The first sound bite I had for Python was, 'Bridge the gap between the shell and C.'".

— Guido van Rossum

Monty Python in 1970



Table 25. Python Timeline

Year	Event	Notes
1969	"Monty Python's Flying Circus" premieres on the BBC	
1991	version 0.9.0 (first release)	classes, try/except, lists, dictionaries, modules
1992	version 0.9.6 ported to MS-DOS	
1993	comp.lang.python created	
1994 (Jan)	version 1.0	lambda, reduce(), filter() and map()
1995	version 1.4	keyword arguments, complex numbers
1997 (Dec 31)	version 1.5	
2000 (Oct 16)	version 2.0 final release	list comprehensions
2002 (Dec 21)	version 2.2	unification of types and classes
2004 (Nov 30)	Version 2.4	
2006 (Sept 19)	version 2.5	
2008 (Oct 1)	version 2.6	backward-compatible with 2.5
2008 (Dec 3)	version 3.0	removing old features and adding new features (not backward-compatible with Python 2)
2009 (June 27)	Version 3.1	Backward-compatible with 3.0; new features/modules; deprecated modules
2018 (July 3)	Version 2.7	Final 2.x version
2012 (Sept 29)	Version 3.3	
2014 (Mar 16)	Version 3.4	pip is included, pathlib
2015 (Sept 13)	Version 3.5	subprocess.run
2016 (Dec 23)	Version 3.6	f-strings, variable annotations
2018 (June 27)	Version 3.7	data classes
2019 (Oct 14)	Version 3.8	assignment expressions, pos-only parameters
2020 (Oct 5)	Version 3.9	
2021 (Oct 4)	Version 3.10	structural pattern matching
2022 (Oct 24)	Version 3.11	exception groups, tomllib , many legacy modules deprecated, misc speedups
2023 (Oct 2)	Version 3.12	remove distutils , misc speedups, better error messages, SQLite3 shell

About Interpreted Languages

- Python is an interpreted language
- The Python interpreter reads a script and interprets it on-the-fly.
- Since there is no compile phase, the development cycle can be very rapid

Like Perl, Ruby, and Bash, Python is an *interpreted* language. The program consists of a text file containing Python commands. To run the program, you run the interpreter (normally called "python.exe", "python", *etc.*) and tell it which file contains the commands.

Advantages of Python

- Clear, readable syntax
- Multi-paradigm
 - object-oriented programming
 - procedural
 - functional
- Dynamic data structures (e.g., lists and dictionaries)
- Exception-based error handling
- Code can be organized into modules and packages
- Extensive standard library and third party modules
- Fun!!

Disadvantages of Python

How to get Python

- Download from www.python.org
- Versions available for most operating systems
- **Anaconda** is superset of standard Python

The latest version of Python is always available via the Python home page. <http://www.python.org/download> will direct you to the latest binaries.

The above URL has Windows MSI installation files.

Linux and OS X come with Python.

For scientific and engineering tasks, the **Anaconda** bundle is a great choice. It contains the Python interpreter plus hundreds of libraries in addition to the standard modules. Among others, it contains **NumPy**, **SciPy**, **pandas**, **iPython**, and **Matplotlib**. Even if you're not doing scientific programming, it includes **Requests**, **PyQt**, **OpenPyxl**, and many other useful modules.



Get Anaconda for Windows, Linux, or Mac at <https://www.anaconda.com>

Which version of Python?

- `python -V`

The `-V` option displays the version of the Python interpreter. Note the *capital V*.

Check <https://www.python.org> for the latest release.

```
$ python -V  
Python 3.10.0
```

The end of Python 2

Python 2.7 is intended to be the last minor release in the 2.x series. The Python maintainers are planning to focus their future efforts on Python 3.

This means that 2.7 will continue to run production systems that have not been ported to Python 3. Two consequences of the long-term significance of 2.7 are:

1. It's very likely the 2.7 release will have a longer period of maintenance compared to earlier 2.x versions. Python 2.7 will continue to be maintained while the transition to 3.x continues, and the developers are planning to support Python 2.7 with bug-fix releases beyond the typical two years.
2. A policy decision was made to silence warnings only of interest to developers. `DeprecationWarning` and its descendants are now ignored unless otherwise requested, preventing users from seeing warnings triggered by an application. This change was also made in the branch that will become Python 3.2. (Discussed on stdlib-sig and carried out in issue 7319.)

-- from the Python 2.7 documentation_

At PyCon 2014 in Montreal, Guido van Rossum extended the end-of-life date for 2.7 to 2020. More recently, the end-of-life date for Python 2.7 has been established on January 1, 2020. This means that there will be no more support from the core Python developers, including bug fixes.

Getting Help

- Books
- Web sites
- `pydoc`

There are many ways of getting help with Python. [The bibliography at the end of this course](#) lists some of the best Python books.

A good starting place is the documentation for the core Python library: <https://docs.python.org/3/library/index.html>.

Packt Publishing produces a large number of Python-oriented books: <https://www.packtpub.com/>
`pydoc` is a command line utility for accessing the builtin Python help system.

Appendix D: String Formatting

Overview

- Strings have a `format()` method
- Allows values to be inserted in strings
- Values can be formatted
- Add a field as placeholders for variable
- Field syntax: `{SELECTOR:FORMATTING}`
- Selector can be empty, index, or keyword
- Formatting controls alignment, width, padding, etc.

Python provides a powerful and flexible way to format data. The string method `format()` takes one or more parameters, which are inserted into the string via placeholders.

The placeholders, called fields, consist of a pair of braces enclosing parameter selectors and formatting directives.

The selector can be followed by a set of formatting directives, which always start with a colon. The simplest directives specify the type of variable to be formatted. For instance, `{1:d}` says to format the second parameter as an integer; `{0:.2f}` says to format the first parameter as a float, rounded to two decimal points.

The formatting part can consist of the following components, which will be explained in detail in the following pages:

```
:[[fill]align][sign][#][0][width][,][.precision][type]
```

Parameter Selectors

- Null for auto-numbering
- Can be numbers or keywords
- Start at 0 for numbers

Selectors refer to which parameter will be used in a placeholder.

Null (empty) selectors — the most common — will be treated as though they were filled in with numbers from left to right, beginning with 0. Null selectors cannot be mixed with numbered or named selectors — either all of the selectors or none of the selectors must be null.

Non-null selectors can be either numeric indices or keywords (strings). Thus, {0} will be replaced with the first parameter, {4} will be replaced with the fifth parameter, and so on. If using keywords, then {name} will be replaced by the value of keyword 'name', and {age} will be replaced by keyword 'age'.

Parameters do not have to be in the same order in which they occur in the string, although they typically are. The same parameter can be used in multiple fields.

If positional and keyword parameters are both used, the keyword parameters must come after all positional parameters.

Example

fmt_params.py

```
person = 'Bob'  
age = 22  
  
print("{0} is {1} years old.".format(person, age)) # Placeholders can be numbered  
print("{0}, {0}, {0} your boat".format('row')) # Placeholders can be reused  
print("The {1}-year-old is {0}".format(person, age)) # They do not have to be in order  
(but usually are)  
print("{name} is {age} years old.".format(name=person, age=age)) # Selectors can be  
named  
print()  
print("{} is {} years old.".format(person, age)) # Empty selectors are autonumbered (but  
all selectors must either be empty or explicitly numbered)  
print("{name} is {} and his favorite color is {}".format(22, 'blue', name='Bob')) #  
Named and numbered selectors can be mixed
```

fmt_params.py

```
Bob is 22 years old.  
row, row, row your boat  
The 22-year-old is Bob  
Bob is 22 years old.  
  
Bob is 22 years old.  
Bob is 22 and his favorite color is blue
```

f-strings

- **f** in front of literal strings
- More readable
- Same rules as `string.format()`

Starting with version 3.6, Python also supports *f-strings*.

The big difference from the `format()` method is that the parameters are inside the {} placeholders. Place formatting details after a : as usual.

Since the parameters are part of the placeholders, parameter numbers are not used.

All of the following formatting tools work with both `string.format()` and f-strings.

Example

`fmt_fstrings.py`

```
person = 'Bob'  
age = 22  
result = 39.128935  
  
print(f"{person} is {age} years old.")  
print(f"The {age}-year-old is {person}.")  
print(f"Result is {result:.2f}")  
print()
```

`fmt_fstrings.py`

```
Bob is 22 years old.  
The 22-year-old is Bob.  
Result is 39.13
```

Data types

- Fields can specify data type
- Controls formatting
- Raises error for invalid types

The type part of the format directive tells the formatter how to convert the value. Built-in types have default formats – 's' for strings, 'd' for integers, 'f' for float.

Some data types can be specified as either upper or lower case. This controls the output of letters. E.g., {:x} would format the number 48879 as 'beef', but {:X} would format it as 'BEEF'.

The type must generally match the type of the parameter. An integer cannot be formatted with type 's'. Integers can be formatted as floats, but not the other way around. Only integers may be formatted as binary, octal, or hexadecimal.

Example

fmt_types.py

```
person = 'Bob'  
value = 488  
bigvalue = 3735928559  
result = 234.5617282027  
  
print('{:s}'.format(person))      # String  
print('{name:s}'.format(name=person))    # String  
print('{:d}'.format(value))        # Integer (displayed as decimal)  
print('{:b}'.format(value))        # Integer (displayed as binary)  
print('{:o}'.format(value))        # Integer (displayed as octal)  
print('{:x}'.format(value))        # Integer (displayed as hex)  
print('{:X}'.format(bigvalue))     # Integer (displayed as hex with uppercase digits)  
print('{:f}'.format(result))       # Float (defaults to 6 places after the decimal point)  
print('{:.2f}'.format(result))     # Float rounded to 2 decimal places
```

fmt_types.py

```
Bob
Bob
488
111101000
750
1e8
DEADBEEF
234.561728
234.56
```

Table 26. Formatting Types

Type	Description
b	Binary – converts number to base 2
c	Character – converts to corresponding character, like chr()
d	Decimal – outputs number in base 10
e, E	Exponent notation. 'e' prints the number in scientific notation using the letter 'e' to indicate the exponent. 'E' is the same, except it uses the letter 'E'
f, F	Floating point. 'F' and 'f' are the same.
g	General format. For a given precision p >= 1, rounds the number to p significant digits and then formats the result in fixed-point or scientific notation, depending on magnitude. This is the default for numbers
G	Same as g, but upper-cases 'e', 'nan', and 'inf'
n	Same as d, but uses locale setting for number separators
o	Octal – converts number to base 8
s	String format. This is the default type for strings
x, X	Hexadecimal – convert number to base 16; A-F match case of 'x' or 'X'
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

Field Widths

- Specified as {0:width.precision}
- Width is really minimum width
- Precision is either maximum width or # decimal points

Fields can specify a minimum width by putting a number before the type. If the parameter is shorter than the field, it will be padded with spaces, on the left for numbers, and on the right for strings.

The precision is specified by a period followed by an integer. For strings, precision means the maximum width. Strings longer than the maximum will be truncated. For floating point numbers, precision means the number of decimal places displayed, which will be padded with zeros as needed.

Width and precision are both optional. The default width for all fields is 0; the default precision for floating point numbers is 6.

It is invalid to specify precision for an integer.

Example

fmt_width.py

```
name = 'Ann Elk'
value = 10000
airspeed = 22.347
# note: [] are used to show blank space, and are not part of the formatting
print('{:s}'.format(name))      # Default format -- no padding
print('{:10s}'.format(name))    # Left justify, 10 characters wide
print('{:3s}'.format(name))     # Left justify, 3 characters wide, displays entire
string
print('{:3s}'.format(name))     # Left justify, 3 characters wide, truncates string to
max width
print()
print('{:8d}'.format(value))    # Right justify, decimal, 8 characters wide (all
numbers are right-justified by default)
print('{:8f}'.format(value))    # Right justify int as float, 8 characters wide
print('{:8f}'.format(airspeed))  # Right justify float as float, 8 characters wide
print('{:.2f}'.format(airspeed)) # Right justify, float, 3 decimal places, no maximum
width
print('{:.3f}'.format(airspeed)) # Right justify, float, 3 decimal places, maximum
width 8
```

fmt_width.py

```
[Ann Elk]
[Ann Elk    ]
[Ann Elk]
[Ann]

[ 10000]
[10000.000000]
[22.347000]
[22.35]
[ 22.347]
```

Alignment

- Alignment within field can be left, right, or centered
 - < left align
 - > right align
 - ^ center
 - = right align but put padding after sign

You can align the data to be formatted. It can be left-aligned (the default), right-aligned, or centered. If formatting signed numbers, the minus sign can be placed on the left side.

Example

fmt_align.py

```
name = 'Ann'
value = 12345
nvalue = -12345

# note: all of the following print in a field 10 characters wide
print('{:10s}'.format(name))      # Default (left) alignment
print('{:<10s}'.format(name))     # Explicit left alignment
print('{:>10s}'.format(name))     # Right alignment
print('{:^10s}'.format(name))      # Centered
print()
print('{:10d} {:10d}'.format(value, nvalue))    # Default (right) alignment
print('{:>10d} {:>10d}'.format(value, nvalue))  # Explicit right alignment
print('{:<10d} {:<10d}'.format(value, nvalue))  # Left alignment
print('{:>10d} {:>10d}'.format(value, nvalue))  # Centered
print('{:=10d} {:=10d}'.format(value, nvalue))    # Right alignment, but pad _after_
sign
```

fmt_align.py

```
[Ann      ]  
[Ann      ]  
[      Ann]  
[  Ann    ]  
  
[    12345] [    -12345]  
[    12345] [    -12345]  
[12345    ] [-12345    ]  
[ 12345    ] [  -12345   ]  
[    12345] [-    12345]
```

Fill characters

- Padding character must precede alignment character
- Default is one space
- Can be any character except }

By default, if a field width is specified and the data does not fill the field, it is padded with spaces. A character preceding the alignment character will be used as the fill character.

Example

fmt_fill.py

```
name = 'Ann'
value = 123

print('{:>10s}'.format(name))      # Right justify string, pad with space (default)
print('{.:>10s}'.format(name))      # Right justify string, pad with '.'
print('{:->10s}'.format(name))      # Right justify string, pad with '-'
print('{:.10s}'.format(name))        # Left justify string, pad with ':'
print()
print('{:10d}'.format(value))        # Right justify number, pad with space (default)
print('{:010d}'.format(value))        # Right justify number, pad with zeroes
print('{:_>10d}'.format(value))      # Right justfy, pad with '_' ('>' required)
print('{:+>10d}'.format(value))      # Right justfy, pad with '+' ('>' required)
```

fmt_fill.py

```
[      Ann]  
[.....Ann]  
[-----Ann]  
[Ann]
```

```
[      123]  
[0000000123]  
[_____123]  
[++++++123]
```

Signed numbers

- Can pad with any character except '{}'
- Sign can be '+', '-', or space
- Only appropriate for numeric types

The sign character follows the alignment character, and can be plus, minus, or space.

A plus sign means always display + or – preceding non-zero numbers.

A minus sign means only display a sign for negative numbers.

A space means display a – for negative numbers and a space for positive numbers.

Example

fmt_signed.py

```
values = 123, -321, 14, -2, 0

for value in values:
    print("default: |{:d}|".format(value)) # default (pipe symbols just to show white
space)
print()

for value in values:
    print(" plus: |{:+d}|".format(value)) # plus sign puts '+' on positive numbers
(and zero) and '-' on negative
print()

for value in values:
    print(" minus: |{:-d}|".format(value)) # minus sign only puts '-' on negative
numbers
print()

for value in values:
    print(" space: |{: d}|".format(value)) # space puts '-' on negative numbers and
space on others
print()
```

fmt_signed.py

```
default: |123|
default: |-321|
default: |14|
default: |-2|
default: |0|  
  
plus: |+123|
plus: |-321|
plus: |+14|
plus: |-2|
plus: |+0|  
  
minus: |123|
minus: |-321|
minus: |14|
minus: |-2|
minus: |0|  
  
space: | 123|
space: |-321|
space: | 14|
space: |-2|
space: | 0|
```

Parameter Attributes

- Specify elements or properties in template
- No need to repeat parameters
- Works with sequences, mappings, and objects

When specifying container variables as parameters, you can select elements in the format rather than in the parameter list. For sequences or dictionaries, index on the selector with []. For object attributes, access the attribute from the selector with . (period).

Example

fmt_attrib.py

```
from datetime import date

fruits = 'apple', 'banana', 'mango'
values = [5, 18, 27, 6]
dday = date(1944, 6, 6)
pythons = {'Idle': 'Eric', 'Cleese': 'John', 'Gilliam': 'Terry',
           'Chapman': 'Graham', 'Palin': 'Michael', 'Jones': 'Terry'}

print('{0[0]} {0[2]}'.format(fruits)) # select from tuple
print('{f[0]} {f[2]}'.format(f=fruits)) # named parameter + select from tuple
print()
print('{0[0]} {0[2]}'.format(values)) # Select from list
print()
print('{0[Palin]} {0[Cleese]}'.format(pythons)) # select from dict
print('{names[Palin]} {names[Cleese]}'.format(names=pythons)) # named parameter + select
from dict
print()
print('{0.month}-{0.day}-{0.year}'.format(dday)) # select attributes from date
```

fmt_attrib.py

```
apple mango  
apple mango
```

```
5 27
```

```
Michael John  
Michael John
```

```
6-6-1944
```

Formatting Dates

- Special formats for dates
- Pull appropriate values from date/time objects

To format dates, use special date formats. These are placed, like all formatting codes, after a colon. For instance, {0:%B %d, %Y} will format a parameter (which must be a `datetime.datetime` or `datetime.date`) as "Month DD, YYYY".

Example

`fmt_dates.py`

```
from datetime import datetime

event = datetime(2016, 1, 2, 3, 4, 5)

print(event) # Default string version of date
print()

print("Date is {0:%m}/{0:%d}/{0:%y}".format(event)) # Use three placeholders for month,
# day, year
print("Date is {:%m/%d/%y}".format(event)) # Format month, day, year with a single
# placeholder
print("Date is {:%A, %B %d, %Y}".format(event)) # Another single placeholder format
```

fmt_dates.py

```
2016-01-02 03:04:05
```

```
Date is 01/02/16
```

```
Date is 01/02/16
```

```
Date is Saturday, January 02, 2016
```

Table 27. Date Formats

Directive	Meaning	See note
%a	Locale's abbreviated weekday name.	
%A	Locale's full weekday name.	
%b	Locale's abbreviated month name.	
%B	Locale's full month name.	
%c	Locale's appropriate date and time representation.	
%d	Day of the month as a decimal number [01,31].	
%f	Microsecond as a decimal number [0,999999], zero-padded on the left	1
%H	Hour (24-hour clock) as a decimal number [00,23].	
%I	Hour (12-hour clock) as a decimal number [01,12].	
%j	Day of the year as a decimal number [001,366].	
%m	Month as a decimal number [01,12].	
%M	Minute as a decimal number [00,59].	
%p	Locale's equivalent of either AM or PM.	2
%S	Second as a decimal number [00,61].	3
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.	4
%W	Weekday as a decimal number [0(Sunday),6].	
%w	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.	4
%x	Locale's appropriate date representation.	
%X	Locale's appropriate time representation.	
%y	Year without century as a decimal number [00,99].	
%Y	Year with century as a decimal number.	
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).	5
%Z	Time zone name (empty string if the object is naive).	
%%	A literal '%' character.	

1. When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in `datetime` objects, and therefore always available).
2. When used with the `strptime()` method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.
3. The range really is 0 to 61; according to the Posix standard this accounts for leap seconds and the (very rare) double leap seconds. The `time` module may produce and does accept leap seconds since it is based on the Posix standard, but the `datetime` module does not accept leap seconds in `strptime()` input nor will it produce them in `strftime()` output.
4. When used with the `strptime()` method, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.
5. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string '`-0330`'.

Run-time formatting

- Use parameters to specify alignment, precision, width, and type
- Use {} placeholders for runtime values for the above

To specify formatting values at runtime, use a {} placeholder for the value, and insert the desired value in the parameter list. These placeholders are numbered along with the normal placeholders.

Example

fmt_runtime.py

```
FIRST_NAME = 'Fred'  
LAST_NAME = 'Flintstone'  
AGE = 35  
  
print("{0} {1}".format(FIRST_NAME, LAST_NAME))  
  
WIDTH = 12  
print("{0:{width}s} {1:{width}s}".format( # value of WIDTH used in format spec  
    FIRST_NAME,  
    LAST_NAME,  
    width=WIDTH,  
)  
  
PAD = '-'  
WIDTH = 20  
ALIGNMENTS = ('<', '>', '^')  
  
for alignment in ALIGNMENTS:  
    print("{0:{pad}{align}{width}s} {1:{pad}{align}{width}s}".format( # values of PAD,  
    WIDTH, ALIGNMENTS used in format spec  
    FIRST_NAME,  
    LAST_NAME,  
    width=WIDTH,  
    pad=PAD,  
    align=alignment,  
)
```

fmt_runtime.py

```
Fred Flintstone  
Fred      Flintstone  
Fred----- Flintstone-----  
-----Fred -----Flintstone  
-----Fred----- Flintstone-----
```

Miscellaneous tips and tricks

- Adding commas to large numbers {n:,}
- Auto-converting parameters to strings (!s)
- Non-decimal prefixes

- Adding commas to large numbers {n:,}

You can add a comma to the format to add commas to numbers greater than 999.

Using a format type of !s will call str() on the parameter and force it to be a string.

Using a # (pound sign) will cause binary, octal, or hex output to be preceded by '0b', '0o', or '0x'. This is only valid with type codes b, o, and x.

Example

fmt_misc.py

```
'''Demonstrate misc formatting'''

big_number = 2303902390239

print("Big number: {:.d}".format(big_number)) # Add commas for readability
print()

value = 27

print("Binary: {:#010b}".format(value)) # Binary format with leading 0b
print("Octal:  {:#010o}".format(value)) # Octal format with leading 0o
print("Hex:     {:#010x}".format(value)) # Hexadecimal format with leading 0x
print()
```

fmt_misc.py

```
Big number: 2,303,902,390,239
```

```
Binary: 0b00011011
```

```
Octal: 0o00000033
```

```
Hex: 0x0000001b
```

Index

@

`@name.setter`, 227
`@property`, 227
`@staticmethod`, 232
`__pycache__`, 183

A

Amoeba, 329
Anaconda, 8
Anaconda bundle, 335
Anaconda Prompt, 8
Anaconda prompt, 8
Array types, 82
ASCIIbetically, 241
Atom, 15

B

`backrefs`, 284
`bool`, 43
Boolean operators, 74
break statement, 77
builtin classes, 28
builtin functions, 28
 table, 29

C

`callback` function, 241
`CamelCase`, 217
`CapWords`, 217
class
 constructor, 225
 definition, 217
 instance methods, 226
 private method, 233
 static method, 232
`class data`, 216
class data, 230
class method, 230
class object, 230
class statement, 217
classes, 216
command line parameters, 62

command prompt, 8

`complex`, 43
conditional expression, 71
constructor, 225
constructor, 216
continue statement, 77
counting, 135

D

`date`, 317
dates and times, 317
`datetime`, 317
derived class, 234
`dict()`, 136
dictionary, 135
 adding elements, 136
 counting with, 146
 getting values, 139
 iterating over, 142
 reading file into, 144
dictionary functions, 138

E

Eclipse, 174
Eclipse, 15
editors and IDEs, 15
`email.mime`, 307
`enumerate()`, 106
Escape characters, 31
escape codes, 32
escape sequences
 table, 37
`exceptions`, 194
exceptions, 194
 else, 199
 finally, 201
 generic, 197
 ignoring, 198
 list, 204
 multiple, 196
exponentiation, 45

F

file modes
 table, 123
file(), 130
files
 opening, 128
floats, 43
floored division, 45
flow control, 67
for loop, 92
format, 54
format strings, 55
formatting, 54
functions
 definition, 156
 keyword parameters, 161
 keyword-only parameters, 160
 named parameters, 161
 optional parameters, 159
 positional parameters, 159
 returning values, 165

G

generator expressions, 114
global statement, 169
global variables, 166
grabbing data from the web, 304
Guido van Rossum, 329, 337

H

help on Python, 338
help(), 13
HTTP download, 304

I

if statement, 68
if-else, 71
if/elif/else, 68
in, 102
indentation, 70
indexing, 89
inheritance, 234
initializer, 216
instance data, 216

instance methods, 226
integers, 43
interactive prompt, 10
interpreted languages, 332
interpreter, 8
interpreter attributes, 293
IPython, 11
iterable, 97
iterating through a sequence, 92

K

key/value pairs, 136
keywords, 28

L

lambda function, 248
lambda functions, 248
launching external programs, 296
legacy string formatting, 61
len(), 104
Linux, 335
list comprehension, 111
list methods
 table, 85
lists, 84
literal dictionary constructor, 136
literal numbers, 43
literal Unicode characters, 35
local variables, 166
logging
 exceptions, 211
 formatted, 207
 simple, 205

M

Mac OS X, 335
MacOS, 335
mapping, 135
math functions, 313
math operators
 table, 47
math operators and expressions, 45
max(), 104
min(), 104
modules, 173

alias, 177
creating, 174
definition, 173
naming, 174
search path, 184
Monty Python, 330

N

nested sequences, 100
non-printable characters, 31
None, 27
Notepad++, 15
numeric literals, 43

O

object-oriented language, 216
object-oriented programming, 216
objects, 216
opening files, 122, 128
order of operations, 46
OS X, 335
os.system(), 296
os.walk(), 301

P

package
 alias, 177
packages, 189
PATH, 9
PATH variable, 8
PATH variable, 9
Perl, 261
pip, 191
polymorphic, 216
popen(), 296
print() function, 52
private data, 225
private methods, 233
properties, 227
PyCharm, 174
PyCharm Community Edition, 15
pydoc, 13
PyPI, 191
Python 2.7 end-of-life, 337
Python advantages, 333

Python disadvantages, 334
Python home page, 335
Python Interpreter, 10
Python Package Index, 191
Python script, 12
Python style guide, 27
Python timeline, 331
Python version, 336
Python Wiki, 15
python3, 8
PYTHONPATH, 184

R

random, 314
random.choice(), 314
random.randint(), 314
random.randrange(), 314
random.sample(), 314
random.shuffle(), 314
raw strings, 34
re.compile(), 267
re.findall(), 264
re.finditer(), 264
re.search(), 264
read(), 130
Reading from the keyboard, 64
reading text files, 124
readline(), 130
readlines(), 130
Regular Expression Metacharacters
 table, 263
regular expressions, 261, 261
 about, 261
 atoms, 262
 branches, 262
 compilation flags, 271
 finding matches, 264
 grouping, 277
 re objects, 267
 replacing text, 282
 replacing text with callback, 287
 special groups, 280
 splitting text, 289
 syntax overview, 262

Relational Operators, 72

`reversed()`, 104

Running Python scripts, 12

S

scripts vs. modules, 173

`self`, 226

sending email, 307

sequence functions, 104

Sequences, 82

set, 148

 creating, 149

 functions and methods (table), 152

 operations, 150

sets, 148

`sh`, 296

slicing, 89

smtplib, 307

sort, 241

sorted

`key` parameter, 243

`sorted()`, 104, 242

`sorted()`, 243

sorting

 custom key, 245

 dictionaries, 255

 in place, 256

 nested data, 251

 reverse, 257, 258

Spyder, 15

standard exception hierarchy, 204

standard I/O, 295

starting python, 8

static method, 232

`stderr`, 295

`stdin`, 295

`stdio`, 295

`stdout`, 295

string formatting, 54

 alignment, 349

 data types, 343

 dates, 359

 field widths, 346

 fill characters, 352

misc, 365

parameter attributes, 357

run-time, 363

selectors, 340

signed numbers, 354

string methods, 38

string operators, 38

Strings, 31

strings

 literal, 32

 single-delimited, 32

StudlyCaps, 217

Sublime, 15

`subprocess`, 296

`sum()`, 104

`sys` module, 293

`sys.executable`, 294

`sys.modules`, 294

`sys.path`, 294

`sys.platform`, 294

`sys.prefix`, 294

`sys.version`, 294

T

terminal window, 8

ternary operator, 71

`time`, 317

`timedelta`, 317

triple-delimited strings, 33

triple-quoted strings, 33

tuple unpacking, 99

tuples, 95

type conversions, 48

U

Unicode, 31

Unicode characters, 35

UpperCamelCase, 217

`urllib` module, 304

`urlopen()`, 304

using try/except, 195

V

variable names, 27

variable scope, 166

variable typing, [30](#)

Variables, [27](#)

Visual Studio Code, [15](#), [22](#)

W

walking directory trees, [301](#)

while loop, [76](#)

whitespace, [70](#)

Windows, [335](#)

write(), [130](#)

writelines(), [130](#)

Z

`zip()`, [104](#)

`zipfile`, [320](#)

zipped archives, [320](#)