

# Custom Intro to Python for BNP

John Strickler

Version 1.0, February 2020



# Table of Contents

About this course .....	1
Welcome! .....	2
Classroom etiquette .....	3
Course Outline .....	4
Student files .....	5
Extracting the student files .....	6
Examples .....	7
Lab Exercises .....	8
Appendices .....	8
Chapter 1: An Overview of Python .....	9
What is Python? .....	10
The Birth of Python .....	12
About Interpreted Languages .....	14
Advantages of Python .....	15
Disadvantages of Python .....	15
How to get Python .....	16
Which version of Python? .....	16
The end of Python 2 .....	17
Getting Help .....	18
One day on Dagobah .....	19
Chapter 2: The Python Environment .....	21
Starting Python .....	22
If the interpreter is not in your PATH .....	23
Using the interpreter .....	25
Trying out a few commands .....	26
Running Python scripts (explicit) .....	27
Running Python scripts (implicit) .....	28
Using pydoc .....	31
Python Editors and IDEs .....	32
Chapter 3: Getting Started .....	35
Using variables .....	36
Keywords and Builtins .....	37

Variable typing .....	39
Strings .....	40
Single-delimited string literals .....	41
Triple-delimited string literals .....	42
Raw string literals .....	43
Unicode characters .....	44
String operators and methods .....	47
String Methods .....	49
Numeric literals .....	53
Math operators and expressions .....	55
Converting among types .....	58
Writing to the screen .....	58
String Formatting .....	61
Legacy String Formatting .....	63
Command line parameters .....	66
Reading from the keyboard .....	67
Chapter 4: Flow Control .....	71
About flow control .....	72
What's with the white space? .....	73
if and elif .....	74
Conditional Expressions .....	76
Relational Operators .....	77
Boolean operators .....	79
while loops .....	81
Alternate ways to exit a loop .....	82
Chapter 5: Array types .....	87
About Array Types .....	88
Lists .....	90
Indexing and slicing .....	93
Iterating through a sequence .....	97
Tuples .....	99
Iterable Unpacking .....	101
Nested sequences .....	102
Functions for all sequences .....	105
Using enumerate() .....	108

Operators and keywords for sequences .....	111
The range() function .....	114
List comprehensions .....	117
Generator Expressions .....	120
Chapter 6: Working with Files .....	127
Text file I/O .....	128
Opening a text file .....	129
The <i>with</i> block .....	130
Reading a text file .....	131
Writing to a text file .....	136
Chapter 7: Dictionaries .....	141
About dictionaries .....	142
When to use dictionaries? .....	143
Creating dictionaries .....	144
Getting dictionary values .....	148
Iterating through a dictionary .....	151
Reading file data into a dictionary .....	153
Counting with dictionaries .....	155
About sets .....	157
Creating Sets .....	158
Working with sets .....	159
Chapter 8: Functions .....	165
Defining a function .....	166
Returning values .....	169
Function parameters .....	170
Variable scope .....	178
Chapter 9: Sorting .....	185
Sorting Overview .....	186
The sorted() function .....	187
Custom sort keys .....	188
Lambda functions .....	193
Sorting nested data .....	196
Sorting dictionaries .....	199
Sorting in reverse .....	201
Sorting lists in place .....	203

Chapter 10: Errors and Exception Handling	205
Syntax errors	206
Exceptions	207
Handling exceptions with try	208
Handling multiple exceptions	209
Handling generic exceptions	210
Ignoring exceptions	211
Using else	212
Cleaning up with finally	214
Chapter 11: Using Modules	221
What is a module?	222
Creating Modules	223
The import statement	226
Where did <code>__pycache__</code> come from?	227
Module search path	228
Packages	229
Example	230
Module Aliases	231
When the batteries aren't included	232
Chapter 12: Regular Expressions	235
Regular Expressions	236
RE Syntax Overview	237
Finding matches	239
RE Objects	242
Compilation Flags	245
Groups	249
Special Groups	252
Replacing text	254
Replacing with a callback	256
Splitting a string	259
Chapter 13: Using the Standard Library	261
The sys module	262
Interpreter Information	262
STDIO	263
Launching external programs	264

Paths, directories and filenames .....	266
Walking directory trees .....	270
Grabbing data from the web .....	273
Sending email .....	276
math functions .....	282
Random values .....	283
Dates and times .....	286
Zipped archives .....	289
Chapter 14: An Introduction to Python Classes .....	293
About O-O programming .....	294
Defining classes .....	295
Constructors .....	297
Instance methods .....	298
Properties .....	301
Class methods and data .....	305
Static Methods .....	307
Private methods .....	308
Inheritance .....	309
Untangling the nomenclature .....	312
Appendix A: Python Bibliography .....	315
Index .....	321





# About this course

# Welcome!

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

**Instructor name:**

**Instructor e-mail:**



**Have Fun!**

## Classroom etiquette

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

Please turn off cell phone ringers and other noisemakers.

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

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

### IMPORTANT

Please do not bring killer rabbits to class. They might maim, dismember, or otherwise disturb your fellow students.

# Course Outline

## Day 1

**Chapter 1** An overview of Python

**Chapter 2** The Python environment

**Chapter 3** Getting started

**Chapter 4** Flow control

**Chapter 5** Array types

**Chapter 6** Working with files

## Day 2

**Chapter 7** Dictionaries and sets

**Chapter 8** Functions

**Chapter 9** Sorting

**Chapter 10** Errors and exception handling

**Chapter 11** Regular expressions

**Chapter 12** Using the standard library

### NOTE

The actual schedule varies with circumstances. There may not be time to cover all topics.

## Student files

You will need to load some 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 **py3bnpintro**.

What's in the files?

**py3bnpintro** contains data and other files needed for the exercises

**py3bnpintro/EXAMPLES** contains the examples from the course manuals.

**py3bnpintro/ANSWERS** contains sample answers to the labs.

### WARNING

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

# Extracting the student files

## Windows

Open the file **py3bnpintro.zip**. Extract all files to your desktop. This will create the folder **py3bnpintro**.

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

Copy or download **py3bnpintro.tgz** to your home directory. In your home directory, type

```
tar xzvf py3bnpintro.tgz
```

This will create the **py3bnpintro** directory under your home directory.

# Examples

Nearly all examples from the course manual are provided in the EXAMPLES subdirectory.

It will look like this:

## Example

### **cmd\_line\_args.py**

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

import sys ①

print(sys.argv) ②

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

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

### **cmd\_line\_args.py Fred**

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

## Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in py3bnpintro/ANSWERS

## Appendices

- Appendix A: Python Bibliography



# Chapter 1: An Overview of Python

## Objectives

- Learning brief history of Python
- Understanding Python's good points (or bad points)
- Downloading and installing Python
- Comparing Python 2 to Python 3
- Getting help

# 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 1. 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 2.5 or 2.6)
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	
2018 (June 27)	Version 3.7	

## 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
- Code can be organized into modules and packages
- Exception-based error handling
- Dynamic data structures (e.g., lists and dictionaries)
- Extensive standard library and third party modules
- Strong introspection capabilities
- Can be extended with C/C++

# 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.

### NOTE

Get Anaconda for Windows, Linux, or Mac at <https://www.anaconda.com>

## Which version of Python?

- `python -V` displays current version

The `-V` option displays the version of the Python interpreter. Note the *capital V*.

The current release is 3.7

```
$ python -V
Python 3.6.0
```



## 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. [Appendix A](#) lists some of the best Python books.

A good starting place is <http://docs.python.org/3/index.html>.

A good source of Python books is Packt Publishing: <https://www.packtpub.com/>

## One day on Dagobah

EXTERIOR: DAGOBAH -- DAY

With Yoda strapped to his back, Luke climbs up one of the many thick vines that grow in the swamp until he reaches the Dagobah statistics lab. Panting heavily, he continues his exercises -- grepping, installing new packages, logging in as root, and writing replacements for two-year-old shell scripts in Python.

YODA: Code! Yes. A programmer's strength flows from code maintainability. But beware of Perl. Terse syntax... more than one way to do it... default variables. The dark side of code maintainability are they. Easily they flow, quick to join you when code you write. If once you start down the dark path, forever will it dominate your destiny, consume you it will.

LUKE: Is Perl better than Python?

YODA: No... no... no. Quicker, easier, more seductive.

LUKE: But how will I know why Python is better than Perl?

YODA: You will know. When your code you try to read six months from now.



# Chapter 2: The Python Environment

## Objectives

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

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

— Guido van Rossum

## Starting Python

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

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

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

If Python is not installed, the solution is evident.

## If the interpreter is not in your PATH

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

### Type the full path

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

### Add the directory to PATH temporarily

#### Windows (at a command prompt)

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

#### non-Windows

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

## Add the directory to PATH permanently

### Windows

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

### non-Windows

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

The command should look something like

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



## Using the interpreter

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

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

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

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

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

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

*Table 2. emacs-style command line editing*

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

## Trying out a few commands

Try out the following commands in the interpreter:

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

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

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

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

## Running Python scripts (explicit)

- Use Python interpreter
- Same for any OS

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

```
python __myscript.py__
```

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

**NOTE**

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

## Running Python scripts (implicit)

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

### Windows

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

The script doesn't need any special permissions.

#### TIP

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

## Non-Windows

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

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

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

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

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

```
chmod 755 __scriptname__.py
```

*or*

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

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

## Example

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

If . is in PATH

```
hello.py
```

If . is not in PATH

```
./hello.py
```

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

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

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

# Using pydoc

## From the Python interpreter

Type

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

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

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

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

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

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

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

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

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

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

### TIP

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

## Python Editors and IDEs

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

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

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

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



## Chapter 2 Exercises

### Exercise 2-1 (hello.py)

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

Run the script explicitly

Run the script implicitly

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

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



# Chapter 3: Getting Started

## Objectives

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

## Using variables

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

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

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

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

```
all_lower_case_with_underscores
```

### Example

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

# Keywords and Builtins

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

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

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

## TIP

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

## Python 3 Keywords

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

*Table 3. Builtin functions*

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

\*These functions are class constructors

## Variable typing

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

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

Certain operations are only valid with the appropriate types.

### WARNING

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

# Strings

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

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



## Single-delimited string literals

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

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

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

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

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

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

### Example

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

## Triple-delimited string literals

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

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

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

### Example

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

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

#### NOTE

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

## Raw string literals

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

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

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

### Example

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

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

## Unicode characters

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

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

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

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

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

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

## Example

### unicode.py

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

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

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

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

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

### unicode.py

```
26°
26°
26\u00B0\n

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

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

Table 4. *Escape Sequences*

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

## String operators and methods

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

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

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

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

Use + (plus) to concatenate two strings.

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

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

String methods may be called on literal strings as well

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

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

## Example

### strings.py

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

a = "My hovercraft is full of EELS"

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

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

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

### strings.py

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



## String Methods

Table 5. string methods

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

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

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

Method	Description
S.strip([chars])	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.
S.swapcase()	Return a copy of S with uppercase characters converted to lowercase and vice versa.
S.title()	Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.
S.translate(table)	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.
S.upper()	Return a copy of S converted to uppercase.
S.zfill(width)	Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

# Numeric literals

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

## Boolean

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

## Integers

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

## Floats

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

## Complex Numbers

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

## Example

### **numeric.py**

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

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

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

① Octal

② Hex

③ Binary

### **numeric.py**

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

# Math operators and expressions

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

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

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

## Division

Division (/) always returns a float result.

## Assignment-with-operation

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

VARIABLE OP=VALUE      e.g. `x += 1`

is equivalent to

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

## Exponentiation

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

## Floored Division

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

## Order of operations

Please Excuse My Drunk Aunt Sally!

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

## Example

### **math\_operators.py**

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

x = 22
x += 10 ①

y = 5
y *= 3 ①

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

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

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

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

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

### **math\_operators.py**

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

#### NOTE

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



Table 6. Python Math Operators and Functions

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

## Converting among types

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

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

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

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

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

## Writing to the screen

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

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

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

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

## Example

### print\_examples.py

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

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

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

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

x = "Hello"
y = "world"

print(x, y) ③
print("#-----")

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

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

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

***print\_examples.py***

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

# String Formatting

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

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

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

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

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

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

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

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

## TIP

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

## NOTE

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

<<<

## Example

### string\_formatting.py

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

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

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

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

print("${: ,d}".format(38293892)) ⑥

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

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

② > means right justify

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

④ d is decimal, o is octal, x is hex

⑤ # means add 0x, 0o, etc.

⑥ , means add commas to numeric value

⑦ parameters can be selected by name instead of position

### string\_formatting.py

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

## Legacy String Formatting

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

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

```
%flagW.Ptype
```

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

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

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

### WARNING

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

*Table 7. Legacy formatting types*

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

*Table 8. Legacy formatting flags*

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



## Example

### string\_formatting\_legacy.py

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

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

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

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

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

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

### string\_formatting\_legacy.py

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

## Command line parameters

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

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

### Example

#### `sys_argv.py`

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

import sys

print(sys.argv)
print()

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

① First command line parameter

#### `sys_argv.py` *Gawain*

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

name is Gawain
```

#### TIP

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

## Reading from the keyboard

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

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

**TIP**

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

## Example

### keyboard\_input.py

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

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

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

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

① input is always a string

② convert to numbers as needed

### keyboard\_input.py

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

## Chapter 3 Exercises

### Exercise 3-1 (c2f.py)

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

What the user types:

```
python c2f.py
```

*(or run from PyCharm/Spyder etc)*

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

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

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

### Exercise 3-2 (c2f\_batch.py)

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

```
python c2f_batch.py 100
```

*(or run from PyCharm/Spyder etc)*

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

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

### Exercise 3-3 (string\_fun.py)

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

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

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

# Chapter 4: Flow Control

## Objectives

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

## About flow control

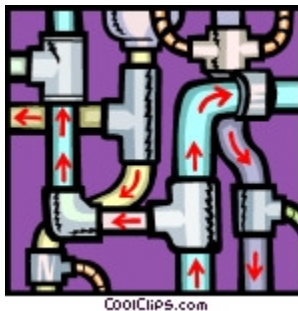
- Controls order of execution
- Conditionals and loops
- Uses Boolean logic

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

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

### NOTE

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





## What's with the white space?

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

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

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

### Example

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

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

#### TIP

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

## if and elif

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

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

### Example

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

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

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

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

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

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

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

**WARNING**

Don't say

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

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

```
if x:
```

## Conditional Expressions

- Used for simple if-then-else conditions

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

```
expr1 if condition else expr2
```

### Example

```
print(long_message if DEBUGGING else short_message)

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

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

## Relational Operators

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

==    !=    <    >    >=    <=

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

## Example

### *if\_else.py*

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

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

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

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

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

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

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

### *if\_else.py*

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

## Boolean operators

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

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

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

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

*Table 9. Boolean Operators*

Expression	Value
<b>AND</b>	
12 and 5	5
5 and 12	12
0 and 12	0
12 and 0	0
"" and 12	""
12 and ""	""
<b>OR</b>	
12 or 5	12
5 or 12	5
0 or 12	12
12 or 0	12
"" or 12	12
12 or ""	12



## while loops

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

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

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

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

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

## Alternate ways to exit a loop

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

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

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

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

### Example

`while_loop_examples.py`

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

print("Welcome to ticket sales\n")

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

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

### ***while\_loop\_examples.py***

```
Welcome to ticket sales
```

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

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

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

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

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

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

```
Please enter a number
```

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

```
goodbye!
```

## Chapter 4 Exercises

### Exercise 4-1 (c2f\_loop.py)

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

**TIP**

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

### Exercise 4-2 (guess.py)

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

**TIP**

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

**TIP**

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

### Exercise 4-3 (guessx.py)

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

## Pseudocode for guess.py

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



# Chapter 5: Array types

## Objectives

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

## About Array Types

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

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

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

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

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

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



## Example

### **typical\_arrays.py**

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

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

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

### **typical\_arrays.py**

```
kiwi
E
Arthur
```

# Lists

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

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

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

Table 10. List Methods

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

## Example

### creating\_lists.py

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

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

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

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

print("list4[-1]:", list4[-1]) ⑦
```

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

### creating\_lists.py

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

## Indexing and slicing

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

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

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

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

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

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

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

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

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

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

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

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

The general syntax for a slice is

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

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

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

and start is incremented by step

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

## Example

### indexing\_and\_slicing.py

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

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

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

phrase = "She turned me into a newt"

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

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

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

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

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

### ***indexing\_and\_slicing.py***

```

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

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

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

```



## Iterating through a sequence

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

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

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

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

**for** loops can be used with any iterable object.

### TIP

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

## Example

### *iterating\_over\_arrays.py*

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

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

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

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

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

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

### *iterating\_over\_arrays.py*

```
Idle
Cleese
Chapman
Gilliam
Palin
Jones

Roger
Old Woman
Prince Herbert
Brother Maynard

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

# Tuples

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

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

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

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

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

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

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

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

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

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

## TIP

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

## Example

### creating\_tuples.py

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

birth_date = 1901, 5, 5

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

latlon = 35.99, -72.390

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

### creating\_tuples.py

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

#### TIP

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

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

## Iterable Unpacking

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

If you have a tuple like this:

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

You can access the elements with

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

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

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

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

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

## Nested sequences

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

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

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

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

## Example

### nested\_sequences.py

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

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

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

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

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

① person is a tuple

② unpack person into variables

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

*nested\_sequences.py*

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

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

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



## Functions for all sequences

- Many builtin functions expect a sequence
- Syntax

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

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

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

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

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

### NOTE

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

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

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

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

## Example

### sequence\_functions.py

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

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

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

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

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

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

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

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

① sorted() returns a sorted list

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

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

***sequence\_functions.py***

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

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

```
full_names: <zip object at 0x10d853908>
```

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

## Using enumerate()

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

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

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

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

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

you get this (virtual) list of tuples:

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

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

## Example

### enumerate.py

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

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

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

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

print()

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

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

② Second parameter to enumerate is added to index

*enumerate.py*

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

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

## Operators and keywords for sequences

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

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

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

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

**+** adds one sequence to another

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

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

## Example

### sequence\_operators.py

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

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

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

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

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

del colors[4] # remove brown ③

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

colors.remove('green') ④

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

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

print("sum of lists:", sum_of_lists)

product = [True] * 5 ⑥

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

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



***sequence\_operators.py***

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

## The range() function

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

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

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

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

## Example

### using\_ranges.py

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

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

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

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

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

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

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

*using\_ranges.py*

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

## List comprehensions

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

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

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

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

can be rewritten as

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

A conditional if may be added:

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

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

## Example

### list\_comprehensions.py

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

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

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

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

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

doubles = [v * 2 for v in values] ③

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

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

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

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

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

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

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

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

### *list\_comprehensions.py*

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

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

[2, 42, 18, 92]

>gronk< >pulaba< >floog<

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

*etc etc*

## Generator Expressions

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

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

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

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

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



## Example

### generator\_expressions.py

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

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

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

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

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

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

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

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

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

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

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

***generator\_expressions.py***

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

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

2 42 18 92

>gronk< >pulaba< >floog<

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

## Chapter 5 Exercises

### Exercise 5-1 (pow2.py)

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

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

**TIP**

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

### Exercise 5-2 (sequences.py)

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

### Exercise 5-3 (sequences.py)

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

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

## Exercise 5-4 (sieve.py)

### FOR ADVANCED STUDENTS

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

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

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

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

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

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

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

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

*See next page for the pseudocode for this program:*

## Pseudocode for sieve.py

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

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

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



# Chapter 6: Working with Files

## Objectives

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

## Text file I/O

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

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

### NOTE

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



## Opening a text file

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

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

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

### Example

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

#### TIP

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

## The *with* block

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

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

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

## Reading a text file

- Iterate through file with for/in

```
for line in file_in
```

- Use methods of the file object

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

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

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

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

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

## Example

### read\_tyger.py

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

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

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

*read\_tyger.py*

## The Tyger

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

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

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

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

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

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

by William Blake

## Example

### reading\_files.py

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

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

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

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

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

*reading\_files.py*

**\*\* About Spam \*\***

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

Flash back to Austin, Minnesota, in 1937.

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

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

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

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

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

**\*\* About Eggs \*\***

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

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

## Writing to a text file

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

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

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

### Example

#### `write_file.py`

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

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

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

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

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



***write\_file.py***

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

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

"writelines" should have been called "writestrings"

Table 11. File Methods

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

## Chapter 6 Exercises

### Exercise 6-1 (line\_no.py)

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

**TIP** | Use `enumerate()`.

Test with the following commands:

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

Test with other files, as desired

### Exercise 6-2 (alt\_lines.py)

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

### Exercise 6-3 (count\_alice.py, count\_words.py)

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

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

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

FOR ADVANCED STUDENTS (icount\_words.py) Modify count\_words.py to make the search case-insensitive.



# Chapter 7: Dictionaries

## Objectives

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

## About dictionaries

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

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

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

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

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

For instance, a dictionary might

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

## When to use dictionaries?

- Mapping
- Counting

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

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

## Creating dictionaries

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

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

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

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



## Example

### creating\_dicts.py

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

d1 = dict() ①

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

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

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

state_caps = dict(pairs) ④

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

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

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

### ***creating\_dicts.py***

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

*Table 12. Frequently used dictionary functions and operators*

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

*Table 13. Less frequently used dictionary functions*

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

## Getting dictionary values

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

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

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

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

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

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

## Example

### getting\_dict\_values.py

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

d1 = dict()

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

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

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

state_caps = dict(pairs)

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

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

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

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

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

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

***getting\_dict\_values.py***

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

## Iterating through a dictionary

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

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

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

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

### TIP

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

## Example

### iterating\_over\_dicts.py

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

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

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

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

*iterating\_over\_dicts.py*

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



## Reading file data into a dictionary

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

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

### Example

#### read\_into\_dict\_of\_tuples.py

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

from pprint import pprint

knight_info = {} ①

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

pprint(knight_info)
print()

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

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

① create empty dict

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

***read\_into\_dict\_of\_tuples.py***

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

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

```
Not Sure
```

**TIP**

See also [read\\_into\\_dict\\_of\\_dicts.py](#) and [read\\_into\\_dict\\_of\\_named\\_tuples.py](#) in the [EXAMPLES](#) folder.

## Counting with dictionaries

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

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

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

**TIP** | Check out the **Counter** class in the **collections** module

### Example

#### count\_with\_dict.py

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

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

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

① create empty dict

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

***count\_with\_dict.py***

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

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

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

## About sets

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

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

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

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

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

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

## Creating Sets

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

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

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

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

## Working with sets

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

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

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

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

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

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

## Example

### set\_examples.py

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

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

set1.add('taupe') ②

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

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

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



***set\_examples.py***

```
{'blue', 'taupe', 'green', 'red', 'purple'}  
{'blue', 'orange', 'yellow', 'green'}  
{'blue', 'green'}  
{'blue', 'yellow', 'taupe', 'green', 'red', 'purple', 'orange'}  
{'yellow', 'taupe', 'red', 'purple', 'orange'}  
{'red', 'purple', 'taupe'}  
{'yellow', 'orange'}  
  
{'ham', 'spam', 'cheese', 'eggs'}
```

Table 14. Set functions and methods

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

## Chapter 7 Exercises

### Exercise 7-1 (scores.py)

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

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

### Exercise 7-2 (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 7-3 (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 7-4 (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 8: 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 fixed or variable parameters and return single or multiple values.

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

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

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

say_hello() ③

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

h = get_hello() ⑤
print(h)
print()

def sqrt(num): ⑥
    return num **.5

m = sqrt(1234) ⑦
n = sqrt(2)

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

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

### ***function\_basics.py***

```
Hello, world
```

```
Hello, world
```

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



# Returning values

- Use the **return** statement
- Return any Python object

To return a value from a function, use the return statement. It can return any Python object, including scalar values, lists, tuple, and dictionaries.

**return** without a value returns None.

## Example

```
return ①  
return 5 ②  
return x ③  
return name,quest,color ④
```

① return None

② return integer 5

③ return object x

④ return tuple of values

### TIP

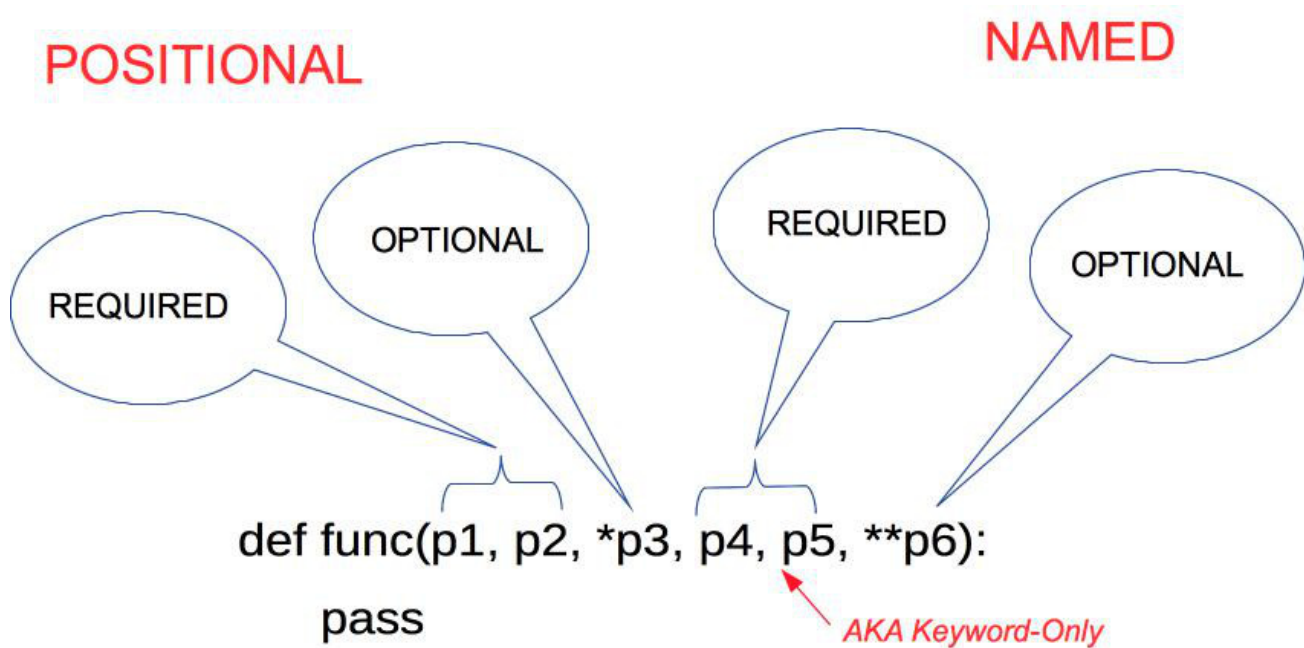
Remember that **return** is a statement, not a function.

## Function parameters

- Four kinds of parameters
  - Required positional
  - Optional positional
  - Required named (AKA keyword-only)
  - Optional named
- No type checking

When defining a function, you need to specify the parameters that the function expects. There are four ways to do this, as described below. Parameters must be specified in the below order (i.e., fixed, optional, keyword-only, keyword).

Required parameters may have default values.



## Positional parameters

### 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.

## 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.

```
pandas.read_csv = 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)
```

## Example

### function\_parameters.py

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

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

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

def fun_two(n): ②
    return n ** 2

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

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

fun_three()
fun_three(10)
print()

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

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

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

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

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

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

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



***function\_parameters.py***

```
fun_one(): Hello, world

fun_two(5) is 25

spam spam spam
spam spam spam spam spam spam spam spam 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
```

## Variable scope

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

When you assign to a variable in a function, that variable is local – it is only visible within the function. If you use an existing variable that has not been assigned to in the function, then it will use the global variable.

Too many globals can make a program hard to read and debug.

## Example

### variable\_scope.py

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

x = 5

def spam():
    x = 22 ①
    print("spam(): x is", x)
    y = "wolverine" ②
    print("spam(): y is", y)

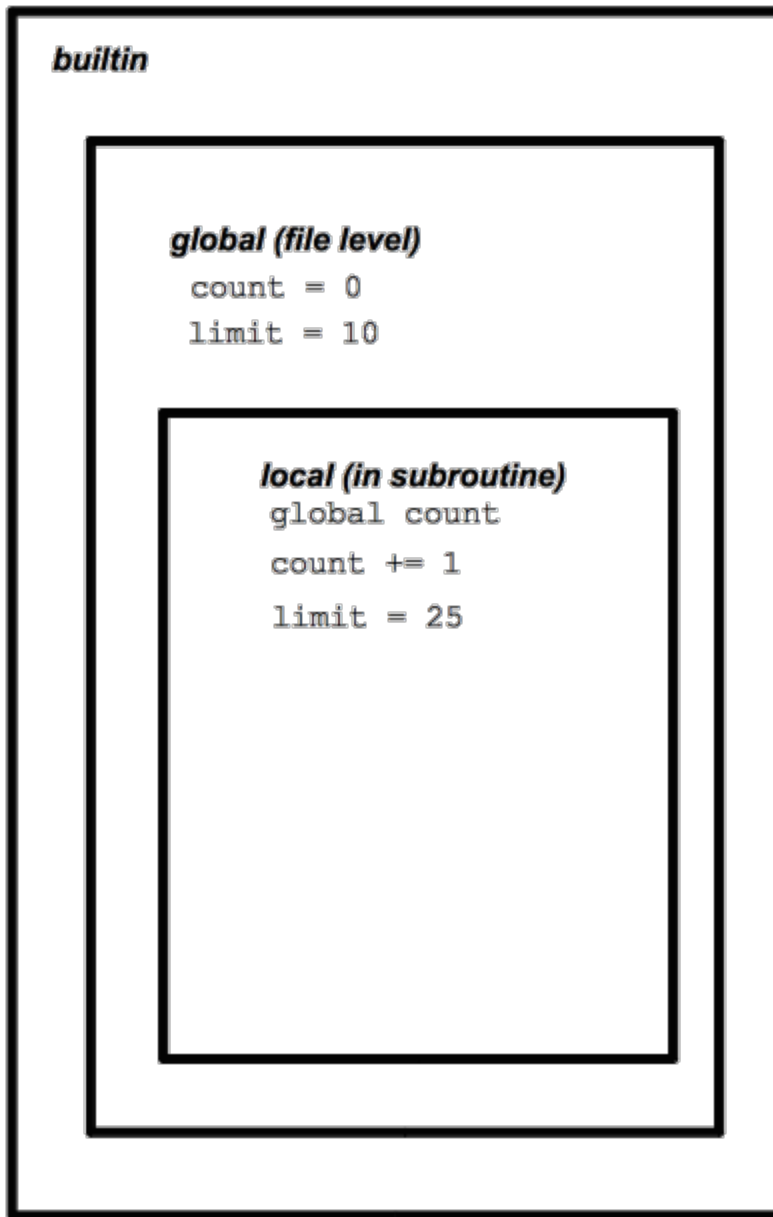
def eggs():
    print("eggs(): x is", x) ③
    y = "wolverine"
    print("eggs(): y is", y)

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

- ① Local variable; does not modify global x
- ② Local variable
- ③ Uses global x since there is no local x

***variable\_scope.py***

```
spam(): x is 22  
spam(): y is wolverine  
  
eggs(): x is 5  
eggs(): y is wolverine  
  
main: x is 5
```



## 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.

### WARNING

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

## Example

### *global\_statement.py*

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

x = 5

def spam():
    global x ①
    x = 22 ②
    print("spam(): x is", x)

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

① Mark x as global, not local

② Modify global variable x

### *global\_statement.py*

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

## Chapter 8 Exercises

### Exercise 8-1 (dirty\_strings.py)

Using the existing script **dirty\_strings.py**, write a function named **cleanup()**. This function should accept one string as input and returns a copy of the string with whitespace trimmed from the beginning and the end, and all upper case letters changed to lower case.

#### NOTE

The `cleanup()` function should not expect a list, but just one string. It should also return a single string. You should not print the cleaned-up string in the `cleanup()` function.

Test the function by looping through the list **spam** and printing each value before and after calling your function.

### Exercise 8-2 (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, not all at once).

#### Example

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

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

### Exercise 8-3 (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.

**NOTE**

It is tricky to parse the expression "5+4", so just expect an expression like "5 + 4" which you can split on whitespace. To easily parse "5+4", you need the **re** (regular expressions) library.

*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 9: 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 (AKA "ASCIIbetically") and it sorts numbers numerically. Mixed types cannot be sorted, since there is no 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 function to calculate 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

```
#!/usr/bin/env python  
  
"""Basic sorting example"""  
  
fruit = ["pomegranate", "cherry", "apricot", "date", "Apple", "lemon", "Kiwi",  
         "ORANGE", "lime", "Watermelon", "guava", "papaya", "FIG", "pear", "banana",  
         "Tamarind", "persimmon", "elderberry", "peach", "BLUEberry", "lychee",  
         "grape"]  
  
sorted_fruit = sorted(fruit) ①  
  
print(sorted_fruit)
```

① sorted() returns a list

#### basic\_sorting.py

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

## Custom sort keys

- Use **key** parameter
- Specify name of function to use
- Key function takes exactly one parameter
- Useful for case-insensitive sorting, sorting by external data, etc.

You can specify a function with the **key** parameter of the `sorted()` function. This function will be used once for each element of the list being sorted, to provide the comparison value. Thus, you can sort a list of strings case-insensitively, or sort a list of zip codes by the number of Starbucks within the zip code.

The function must take exactly one parameter (which is one element of the sequence being sorted) and return either a single value or a tuple of values. The returned values will be compared in order.

You can use any builtin Python function or method that meets these requirements, or you can write your own function.

**TIP**

The `lower()` method can be called directly from the builtin object `str`. It takes one string argument and returns a lower case copy.

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

## Example

### custom\_sort\_keys.py

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

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

def ignore_case(item): ①
    return item.lower() ②

fs1 = sorted(fruit, key=ignore_case) ③
print("Ignoring case:")
print(" ".join(fs1), end="\n\n")

def by_length_then_name(item):
    return (len(item), item.lower()) ④

fs2 = sorted(fruit, key=by_length_then_name)
print("By length, then name:")
print(" ".join(fs2))
print()

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

n1 = sorted(nums) ⑤
print("Numbers sorted numerically:")
for n in n1:
    print(n, end=' ')
print("\n")

n2 = sorted(nums, key=str) ⑥
print("Numbers sorted as strings:")
for n in n2:
    print(n, end=' ')
print()
```

- ① Parameter is *one* element of iterable to be sorted
- ② Return value to sort on
- ③ Specify function with named parameter **key**
- ④ Key functions can return tuple of values to compare, in order
- ⑤ Numbers sort numerically by default
- ⑥ Sort numbers as strings

### ***custom\_sort\_keys.py***

Ignoring case:

Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon  
lime lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon

By length, then name:

FIG date Kiwi lime pear Apple grape guava lemon peach banana cherry lychee ORANGE  
papaya apricot Tamarind BLUEberry persimmon elderberry Watermelon pomegranate

Numbers sorted numerically:

5 32 80 255 400 800 1000 5000

Numbers sorted as strings:

1000 255 32 400 5 5000 80 800

## Example

### sort\_holmes.py

```
#!/usr/bin/env python
"""Sort titles, ignoring leading articles"""
import re

books = [
    "A Study in Scarlet",
    "The Sign of the Four",
    "The Hound of the Baskervilles",
    "The Valley of Fear",
    "The Adventures of Sherlock Holmes",
    "The Memoirs of Sherlock Holmes",
    "The Return of Sherlock Holmes",
    "His Last Bow",
    "The Case-Book of Sherlock Holmes",
]

rx_article = re.compile(r'^(the|a|an)\s+', re.I) ①

def strip_articles(title): ②
    stripped_title = rx_article.sub('', title.lower()) ③
    return stripped_title

for book in sorted(books, key=strip_articles): ④
    print(book)
```

- ① compile regex to match leading articles
- ② create function which takes element to compare and returns comparison key
- ③ strip off article and convert title to lower case
- ④ sort using custom function

***sort\_holmes.py***

```
The Adventures of Sherlock Holmes
The Case-Book of Sherlock Holmes
His Last Bow
The Hound of the Baskervilles
The Memoirs of Sherlock Holmes
The Return of Sherlock Holmes
The Sign of the Four
A Study in Scarlet
The Valley of Fear
```



## Lambda functions

- Shortcut for function definition
- Create function on-the-fly
- Body must be an expression
- May take any number of parameters
- For sorting, takes one parameter

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.

When using a lambda function with the key parameter of `sorted()`, it expects a single parameter, which is one element of the list being sorted. Lambda functions are particularly useful for sorting nested collections, such as lists of tuples.

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

Thus, the following can be used as a template:

```
lambda e: expression
```

### Example

```
s = sorted(mylist, key=lambda e: (len(e), e.lower()))  
  
cnums = sorted(custnums, key=lambda e: get_qtr3_earnings(e))
```

## Example

### lambda\_sort.py

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

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

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

fs1 = sorted(fruit, key=lambda e: e.lower()) ①
print("Ignoring case:")
print(' '.join(fs1))
print()

fs2 = sorted(fruit, key=lambda e: (len(e), e.lower())) ②
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()
```

① lambda returns key function that converts each element to lower case

② lambda returns tuple

***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

```
list2 = sorted(list1, key=lambda e: e[2])
```

## Example

### nested\_sort.py

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

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]): ①
    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]): ②
    print(first_name, last_name, organization, dob)
```

① Select element of nested tuple for sorting

② Select different element of nested tuple for sorting

*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
- Use a lambda function or `itemgetter()` to sort by value

While a dictionary can't be sorted, the keys to a dictionary can. Better yet, the list of tuples returned by `DICT.items()` can be sorted. This list will be sorted by keys, unless you specify a key function.

Use a lambda function or `operator.itemgetter()` to specify the 2nd element of the key,value tuple to sort by values.

Sorting `dictionary.items()` is really just sorting a list of tuples.

### Example

#### `sorting_dicts.py`

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

count_of = dict(red=5, green=18, blue=1, pink=0, grey=27, yellow=5)

# sort by key
for color, num in sorted(count_of.items()): ①
    print(color, num)

print()

# sort by value
for color, num in sorted(count_of.items(), key=lambda e: e[1]): ②
    print(color, num)
```

① No special sort needed to sort by key

② Sorting by value uses second element of nested (key, value) pairs returned by `items()`

*sorting\_dicts.py*

```
blue 1  
green 18  
grey 27  
pink 0  
red 5  
yellow 5
```

```
pink 0  
blue 1  
red 5  
yellow 5  
green 18  
grey 27
```



## Sorting in reverse

- Use `reverse=True`

To sort in reverse, add the `reverse` parameter to `list.sort()` or `sorted()` with a true value (e.g. `True`).

### Example

#### `reverse_sort.py`

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

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) ①
print(" ".join(sorted_fruits))
print()

print("reverse, case-insensitive:")
sorted_fruits = sorted(fruits, reverse=True, key=lambda e: e.lower()) ②
print(" ".join(sorted_fruits))
print()
```

- ① Set **reverse** to `True` to reverse sort
- ② **reverse** can be combined with key functions

***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

## 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`

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

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

fruit.sort(key=str.lower) ①

print(" ".join(fruit))
```

① List is sorted in place; cannot be undone

#### `sort_in_place.py`

```
Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon
lime lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon
```

## Chapter 9 Exercises

### Exercise 9-1 (`scores_by_score.py`)

Redo `scores.py`, printing out the students in descending order by score.

**TIP**

You will not need to change anything in `scores.py` other than the loop that prints out the names and scores.

### Exercise 9-2 (`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.

**TIP**

Read through the file once, putting lines into two lists.

### Exercise 9-3 (`sort_fruit.py`)

Using the file `fruit.txt`, print it out:

- sorted by name case-sensitively
- sorted by name case-insensitively.
- sorted by length of name, then by name
- sorted by the 2nd letter of the name, then the first letter

### Exercise 9-4 (`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.

**TIP**

Use the `split()` method on each line to get the individual fields.

# Chapter 10: Errors and Exception Handling

## Objectives

- Understanding syntax errors
- Handling exceptions with try-except-else-finally
- Learning the standard exception objects

## Syntax errors

- Generated by the parser
- Cannot be trapped

Syntax errors are generated by the Python parser, and cause execution to stop (your script exits). They display the file name and line number where the error occurred, as well as an indication of where in the line the error occurred.

Because they are generated as soon as they are encountered, syntax errors may not be handled.

### Example

```
File "<stdin>", line 1
  for x in bargle
              ^
SyntaxError: invalid syntax
```

**TIP**

When running in interactive mode, the filename is `<stdin>`.

# Exceptions

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

Even if code is syntactically correct, errors can occur. A common run-time error is to attempt to open a non-existent file. Such errors are called exceptions, and cause the interpreter to stop with an error message.

Python has a hierarchy of builtin exceptions; handling an exception higher in the tree will handle any children of that exception.

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

## Example

### `exception_unhandled.py`

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

x = 5
y = "cheese"

z = x + y ①
```

① Adding a string to an int raises **TypeError**

### `exception_unhandled.py`

```
Traceback (most recent call last):
  File "exception_unhandled.py", line 6, in <module>
    z = x + y ①
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

## Handling exceptions with try

- Use try/except clauses
- Specify expected exception

To handle an exception, put the code which might generate an exception in a try block. After the try block, you must specify a except block with the expected exception. If an exception is raised in the try block, execution stops and the interpreter looks for the exception in the except block. If found, it executes the except block and execution continues; otherwise, the exception is treated as fatal and the interpreter exits.

### Example

#### exception\_simple.py

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

try: ①
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")

except TypeError as err: ②
    print("Naughty programmer! ", err)

print("After try-except") ③
```

- ① Execute code that might have a problem
- ② Catch the expected error; assign error object to **err**
- ③ Get here whether or not exception occurred

#### exception\_simple.py

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



## Handling multiple exceptions

- Use a tuple of exception names, but with single argument

If your try clause might generate more than one kind of exception, you can specify a tuple of exception types, then the variable which will hold the exception object.

### Example

#### **exception\_multiple.py**

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except (IOError, TypeError) as err: ①
    print("Naughty programmer! ", err)
```

- ① Use a tuple of 2 or more exception types

#### **exception\_multiple.py**

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

## Handling generic exceptions

- Use **Exception**
- Specify `except` with no exception list
- Clean up any uncaught exceptions

As a shortcut, you can specify **Exception** or an empty exception list. This will handle any exception that occurs in the `try` block.

### Example

**exception\_generic.py**

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except Exception as err: ①
    print("Naughty programmer! ", err)
```

① Will catch *any* exception

**exception\_generic.py**

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

## Ignoring exceptions

- Use the **pass** statement

Use the **pass** statement to do nothing when an exception occurs

Because the except clause must contain some code, the pass statement fulfills the syntax without doing anything.

### Example

#### **exception\_ignore.py**

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except(TypeError, IOError): ①
    pass
```

① Catch exceptions, and do nothing

#### **exception\_ignore.py**

```
_no output_
```

This is probably a bad idea...

## Using else

- executed if no exceptions were raised
- not required
- can make code easier to read

The last except block can be followed by an else block. The code in the else block is executed only if there were no exceptions raised in the try block. Exceptions in the else block are not handled by the preceding except blocks.

The else lets you make sure that some code related to the try clause (and before the finally clause) is only run if there's no exception, without trapping the exception specified in the except clause.

```
try:
    something_that_can_throw_ioerror()
except IOError as e:
    handle_the_IO_exception()
else:
    # we don't want to catch this IOError if it's raised
    something_else_that_throws_ioerror()
finally:
    something_we_always_need_to_do()
```

## Example

### **exception\_else.py**

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

numpairs = [(5, 1), (1, 5), (5, 0), (0, 5)]

total = 0

for x, y in numpairs:
    try:
        quotient = x / y
    except Exception as err:
        print("uh-oh, when y = {}, {}".format(y, err))
    else:
        total += quotient ①
print(total)
```

① Only if no exceptions were raised

### **exception\_else.py**

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

## Cleaning up with finally

- Executed whether or not exception occurs
- Code executed whether or not exception raised

A finally block can be used instead of, or in addition to, an except block. The code in a finally block is executed whether or not an exception occurs. The finally block is executed after the try, except, and else blocks.

## Example

### exception\_finally.py

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

try:
    x = 5
    y = 37
    z = x + y
    print("z is", z)
except TypeError as err: ①
    print("Caught exception:", err)
finally:
    print("Don't care whether we had an exception") ②

print()

try:
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")
except TypeError as err:
    print("Caught exception:", err)
finally:
    print("Still don't care whether we had an exception")
```

① Catch **TypeError**

② Print whether **TypeError** is caught or not

*exception\_finally.py*

```
z is 42
Don't care whether we had an exception

Caught exception: unsupported operand type(s) for +: 'int' and 'str'
Still don't care whether we had an exception
```



## The Standard Exception Hierarchy (Python 3.7)

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
    |   +-- FloatingPointError
    |   +-- OverflowError
    |   +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
    |   +-- ModuleNotFoundError
    +-- LookupError
    |   +-- IndexError
    |   +-- KeyError
    +-- MemoryError
    +-- NameError
    |   +-- UnboundLocalError
    +-- OSError
    |   +-- BlockingIOError
    |   +-- ChildProcessError
    |   +-- ConnectionError
    |   |   +-- BrokenPipeError
    |   |   +-- ConnectionAbortedError
    |   |   +-- ConnectionRefusedError
    |   |   +-- ConnectionResetError
    |   +-- FileExistsError
    |   +-- FileNotFoundError
    |   +-- InterruptedError
    |   +-- IsADirectoryError
    |   +-- NotADirectoryError
    |   +-- PermissionError
    |   +-- ProcessLookupError
    |   +-- TimeoutError
    +-- ReferenceError
    +-- RuntimeError
```

```
|    +-- NotImplementedError
|    +-- RecursionError
+-- SyntaxError
|    +-- IndentationError
|        +-- TabError
+-- SystemError
+-- TypeError
+-- ValueError
|    +-- UnicodeError
|        +-- UnicodeDecodeError
|        +-- UnicodeEncodeError
|        +-- UnicodeTranslateError
```

## Chapter 10 Exercises

### Exercise 10-1 (`c2f_safe.py`)

Rewrite `c2f.py` to handle the error that occurs if the user enters non-numeric data.

### Exercise 10-2 (`c2f_batch_safe.py`)

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



# Chapter 11: 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 a library of python code. Any time a function, class, or variable might be useful in more than one place, it should be put in a module, in order to avoid cut-and-paste disease.

If the function needs to change, you only need to change it in one place.

Any code in a module that is not inside a function definition will be executed the first time a module is loaded, and thus can provide initialization code.

Modules end in **.py**.

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, IDEs (PyCharm, Eclipse, 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

### breakfast.py

```
#!/usr/bin/env python
'''
Spam -- provides a tasty breakfast
'''

# separator for toast toppings
TOPSEP = " and "

def eggs(how):
    '''cook some eggs'''
    print("Cooking up some lovely {} eggs".format(how))

def toast(*toppings):
    '''cook some toast'''
    print("Toasting up some toast with ", TOPSEP.join(toppings))
```

### eat.py

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

from breakfast import eggs, toast

eggs("fried")
toast("butter", "strawberry jam")

print()
help(eggs)
print()
help(toast)
```



***eat.py***

```
Cooking up some lovely fried eggs  
Toasting up some toast with  butter and strawberry jam
```

```
Help on function eggs in module breakfast:
```

```
eggs(how)  
    cook some eggs
```

```
Help on function toast in module breakfast:
```

```
toast(*toppings)  
    cook some toast
```

## 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 several ways:

```
import breakfast
```

Load module breakfast.py. To call function **eggs** which is defined in breakfast.py, use breakfast.eggs() breakfast.eggs("scrambled")

```
from breakfast import eggs [, toast, coffee, ...]
```

Load module breakfast.py and add eggs to the current namespace, so it can be called without the module name:

```
eggs("fried")
```

To import multiple objects, separate them with commas.

```
from breakfast import *
```

Load module breakfast.py and add all objects (except those than begin with "\_") to the current namespace. eggs("poached") toast("butter", "jam")

This is generally not considered a **Good Thing** unless you know what you're doing, or the module documentation suggests it.

## 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__` appeared. The Python interpreter saves a compiled version of each imported module into the `__pycache__` folder. For portability, each compiled version embeds a version string (such as `"cpython-64"`) into the name, so that the same `__pycache__` folder can contain compiled modules for different versions of Python.

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`.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python24.zip', '/usr/lib/python2.4', '/usr/lib/python2.4/plat-
linux2', '/usr/lib/python2.4/lib-tk', '/usr/lib/python2.4/lib-dynload',
'/usr/local/lib/python2.4/site-packages', '/usr/lib/python2.4/site-packages',
'/usr/lib/python2.4/site-packages/...']
```

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

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

#### WARNING

You can also append to `sys.path` in your scripts, but this can result in scripts that will fail if the location of the imported modules changes.

# Packages

- Collection of modules
- Corresponds to a directory
- Can have subpackages

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

Load modules in a package with **import package**.

Access subpackages via **package.sub**.

Access functions via **package.sub.function()**.

## Example

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

```
media
media/{dunder}init{dunder}.py
media/dvd.py
media/videotape.py
media/cd.py
```

To use the function `Search` that is defined in module `dvd`, import `media.dvd`:

```
import media.dvd
media.dvd.Search("Uma Thurman")
```

Other variations are:

```
from media import dvd
dvd.Search("Uma Thurman")
_or_
from media.dvd import Search
Search("Uma Thurman")
```

If a module named `__init__.py` is present in the package, it is executed when the package or any of its modules are imported.

## Module Aliases

- Provide alternate name for module
- Save typing
- Make code more readable

To load a module with a different name, use the syntax

```
import module as alias
```

This is useful to save typing, especially when using packages. It can also make code more readable.

### Example

```
import media.dvd as dvd
found = dvd.search('beatles')

import xml.etree.ElementTree as ET
import Tkinter as tk
```

## When the batteries aren't included

- Python Package Index (PyPI) has over 99,000 packages
- Install with the **pip** utility

Although the Python distribution claims to be "batteries included", functionality beyond the standard library is provided by so-called third party modules. There are about 99,000 such packages in the Python Package Index (<http://pypi.python.org/pypi>).

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



## Chapter 11 Exercises

### Exercise 11-1 (temp\_conv.py)

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

Conversion formulas

$$F = ((9 * C) / 5.0) + 32$$
$$C = (F - 32) * (5.0/9)$$

**TIP**

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

### Exercise 11-2 (c2f\_mod.py)

Re-implement **c2f.py** using temp\_conv

### Exercise 11-3 (f2c.py)

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

### Exercise 11-4 (c2f\_loop\_mod.py)

Re-implement **c2f\_loop.py** using temp\_conv.



# Chapter 12: Regular Expressions

## Objectives

- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding options to a pattern
- Replacing text with callbacks
- Specifying capture groups
- Using RE patterns without creating objects

## 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.

## 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.

**TIP** | There is frequently only one branch.

Two good web apps for working with Python regular expressions are

<https://regex101.com/#python>

<http://www.pythex.org/>

Table 15. 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-grouping 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.

### **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.

### **re.finditer(pattern, string)**

Provides a match object for each match found. Normally used with a **for** loop.

### **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 match methods

**re.match()** is like **match()**, 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.

Use the **search()** method unless you only want to match the beginning of the string.

## Example

### *regex\_finding\_matches.py*

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
    eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo z883 consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
    officia deserunt Y45 mollit anim id est laborum"""

pattern = r'[A-Z]\d{2,3}' ①

if re.search(pattern, s): ②
    print("Found pattern.")
print()

m = re.search(pattern, s) ③
print(m)
if m:
    print("Found:", m.group(0)) ④
print()

for m in re.finditer(pattern, s): ⑤
    print(m.group())
print()

matches = re.findall(pattern, s) ⑥
print("matches:", matches)
```



- ① store pattern in raw string
- ② search returns True on match
- ③ search actually returns match object
- ④ group() returns text that was matched
- ⑤ iterate over all matches in string:
- ⑥ return list of all matches

### ***regex\_finding\_matches.py***

Found pattern.

```
<re.Match object; span=(12, 16), match='M302'>  
Found: M302
```

```
M302  
H476  
Q51  
U901  
A110  
H332  
Y45
```

```
matches: ['M302', 'H476', 'Q51', 'U901', 'A110', 'H332', 'Y45']
```

## 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 the `compile()` function, from the **re** module, with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. The `re.compile()` function has an optional argument for flags which enable special features or fine-tune the match.

**TIP**

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

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
      eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
      ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
      ea commodo Z883 consequat. Duis aute irure dolor in reprehenderit in
      voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
      Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
      officia deserunt Y45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]\d{2,3}', re.I) ①

if rx_code.search(s): ②
    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)
```

- ① Create an re (regular expression) object
- ② Call search() method from the object

*regex\_objects.py*

Found pattern.

Found: M302

M302

r99

H476

Q51

Z883

U901

A110

H332

Y45

matches: ['M302', 'r99', 'H476', 'Q51', 'Z883', 'U901', 'A110', 'H332', 'Y45']

# Compilation Flags

- Control match
- Add features

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 by ORing them together. Each flag has a short for and a long form.

## **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.

## **re.L, re.LOCALE**

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.

## **re.M, re.MULTILINE**

Usually `^` matches only at the beginning of the string, and `$` matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, `^` matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the `$` metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

## **re.S, re.DOTALL**

Makes the `"."` special character match any character at all, including a newline; without this flag, `"."` will match anything except a newline.

**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.

## Example

### regex\_flags.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
    eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo Z883 consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
    officia deserunt Y45 mollit anim id est laborum"""

pattern = r'[A-Z]\d{2,3}'

if re.search(pattern, s, re.IGNORECASE): ①
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I) ②
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)
```

① make search case-insensitive

② short version of flag

*regex\_flags.py*

Found pattern.

Found: M302

M302

r99

H476

Q51

Z883

U901

A110

H332

Y45

matches: ['M302', 'r99', 'H476', 'Q51', 'Z883', 'U901', 'A110', 'H332', 'Y45']



## 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 match for each group.

To access groups in more detail, use **`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()`**. **`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

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
    eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo Z883 consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
    officia deserunt Y45 mollit anim id est laborum"""

pattern = r'([A-Z])(\d{2,3})' ①

for m in re.finditer(pattern, s):
    print(m.group(0), m.group(1), m.group(2)) ②
print()

matches = re.findall(pattern, s) ③
print("matches:", matches)
```

- ① parens delimit groups
- ② group 1 is first group, etc. (group 0 is entire match)
- ③ findall() returns list of tuples containing groups

***regex\_group.py***

```
M302 M 302
H476 H 476
Q51 Q 51
Z883 Z 883
U901 U 901
A110 A 110
H332 H 332
Y45 Y 45
```

```
matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('Z', '883'), ('U', '901'), ('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. The most basic is `(?:pattern)`, which groups but does not capture.

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.

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*

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
      eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
      ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
      ea commodo Z883 consequat. Duis aute irure dolor in reprehenderit in
      voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
      Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
      officia deserunt Y45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])(?P<number>\d{2,3})' ①

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number')) ②
```

- ① Use (?<NAME>...) to name groups
- ② Use m.group(NAME) to retrieve text

### *regex\_special.py*

```
M 302
H 476
Q 51
Z 883
U 901
A 110
H 332
Y 45
```

## Replacing text

- Use `RE.sub(replacement,string[,count])`
- `RE.subn()` returns tuple with string and count

To find and replace text using a regular expression, use the `sub()` method. It takes the replacement text and the string to search as arguments, and returns the modified string.

The third, optional argument is the maximum number of replacements to make.

Be sure to put the arguments in the proper order!

### Example

#### `regex_sub.py`

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
officia deserunt Y45 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) ❶
print(s2)
print()

s3, count = rx_code.subn("___", s) ❷
print("Made {} replacements".format(count))
print(s3)
```

❶ replace pattern with string

❷ `subn` returns tuple with result string and replacement count

*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 [REDACTED] 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 9 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 ___ eu fugiat nulla pariatur.
Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui
officia deserunt ___ mollit anim id est laborum
```

## Replacing with a callback

- Replacement can be function
- Function expects match object, returns replacement text
- Use normally defined function or a lambda

In addition using a string as the replacement, you can specify a function. This function will be called once for each match, and passed in the match object. 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 do things such as:

- preserving case in a replacement
- adding text around the replacement
- looking up the original text in a dictionary or database



## Example

### regex\_sub\_callback.py

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

import re

s = """lorem ipsum M302 dolor sit amet, consectetur r99 adipiscing elit, sed do
    eiusmod tempor incididunt H476 ut labore et dolore magna Q51 aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo Z883 consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat A110 cupidatat non proident, sunt in H332 culpa qui
    officia deserunt Y45 mollit anim id est laborum"""

# my $rx_code = qr/(?P<letter>[A-Z])(?P<number>\d{2,3})/i;
# if ($foo =~ /$rx_code/) { }

rx_code = re.compile(r'(?P<letter>[A-Z])(?P<number>\d{2,3})', re.I)

def update_code(m): ①
    letter = m.group('letter').upper()
    number = int(m.group('number'))
    return '{}{:04d}'.format(letter, number) ②

s2 = rx_code.sub(update_code, s) ③
print(s2)
```

- ① callback function is passed each match object
- ② function returns replacement text
- ③ sub takes callback function instead of replacement text

*regex\_sub\_callback.py*

```
lorem ipsum M0302 dolor sit amet, consectetur R0099 adipiscing elit, sed do  
    eiusmod tempor incididunt H0476 ut labore et dolore magna Q0051 aliqua. Ut enim  
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex  
ea commodo Z0883 consequat. Duis aute irure dolor in reprehenderit in  
voluptate velit esse cillum dolore U0901 eu fugiat nulla pariatur.  
Excepteur sint occaecat A0110 cupidatat non proident, sunt in H0332 culpa qui  
officia deserunt Y0045 mollit anim id est laborum
```

## Splitting a string

- Syntax: `re.split(string[,max])`

The `re.split()` method splits a string into pieces, returning the pieces as a list. The optional `max` argument limits the numbers of pieces.

### Example

#### `regex_split.py`

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

import re

rx_wordsep = re.compile(r"^[a-z]+") ①

s1 = '''There are 10 kinds of people in a Binary world, I hear" -- Geek talk'''

words = rx_wordsep.split(s1) ②
print(words)
```

① When splitting, pattern matches what you **don't** want

② Retrieve text *separated* by your pattern

#### `regex_split.py`

```
['', 'here', 'are', 'kinds', 'of', 'people', 'in', 'a', 'inary', 'world', 'hear',  
'eek', 'talk']
```

## Chapter 12 Exercises

### Exercise 12-1 (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.<sup>1</sup>

### Exercise 12-2 (mark\_big\_words.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 and end of a word.

### Exercise 12-3 (print\_numbers.py)

Write a script to print out all lines in custinfo.dat which contain phone numbers.

### Exercise 12-4 (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.

# Chapter 13: Using the Standard Library

## Objectives

- Overview of the standard library
- Getting information on the Python interpreter's environment
- Running external programs
- Walking through a directory tree
- Working with path names
- Calculating dates and times
- Fetching data from a URL
- Generating random values

## The sys module

- Import the sys module to provide access to the interpreter and its environment
- Get interpreter attributes
- Interact with the operating system

The sys module provides access to some objects used or maintained by the interpreter and to functions that interact strongly with the interpreter.

This module provides details of the current Python interpreter; it also provides objects and methods to interact with the operating system.

Even though the sys module is built into the Python interpreter, it must be imported like any other module.

## Interpreter Information

- sys provides details of interpreter

To get the folder where Python is installed, use **sys.prefix**.

To get the path to the Python executable, use **sys.executable**.

To get a version string, use **sys.version**.

To get the details of the interpreter as a tuple, use **sys.version\_info**.

To get the list of directories that will be searched for modules, examine **sys.path**.

To get a list of currently loaded modules, use **sys.modules**.

To find out what platform (OS/architecture) the script is running on, use **sys.platform**.

## STDIO

- `stdin`
- `stdout`
- `stderr`

The `sys` object defines three file objects representing the three streams of STDIO, or "standard I/O".

Unless they have been redirected, `sys.stdin` is the keyboard, and `sys.stdout` and `sys.stderr` are the console screen. You should use `sys.stderr` for error messages.

### Example

#### `stdio.py`

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

import sys
sys.stdout.write("Hello, world\n")
sys.stderr.write("Error message here...\n")
```

#### *`stdio.py 2>spam.txt`*

```
Hello, world
```

#### *`type spam.txt` windows*

#### *`cat spam.txt` non-windows*

```
Error message here...
```

## Launching external programs

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

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

`os.system()` launches any external command, as though you had typed it at a command prompt. `popen()` opens a pipe to a command so you can read the output of the command one line at a time. `popen()` is very similar to the `open()` function; it returns an iterable object.

For more control over external processes, use the `subprocess` module (part of the standard library), or check out the `sh` module (not part of the standard library).



## Example

### external\_programs.py

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

os.system("hostname") ①

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

- ① Just run "hostname"
- ② Open command line "netstat -an" as a file-like object
- ③ Iterate over lines in output of "netstat -an"
- ④ Check to see if line contains "ESTAB"

### external\_programs.py

```
MacBook-Pro-8.local
tcp4      0      0 10.0.75.29.62193      99.84.41.128.443      ESTABLISHED
tcp4      0      0 10.0.75.29.62192      99.84.41.108.443      ESTABLISHED
tcp4      0      0 10.0.75.29.62189      3.95.121.119.443      ESTABLISHED
tcp4      0      0 10.0.75.29.62188      99.84.127.20.443      ESTABLISHED
tcp4      0      0 10.0.75.29.62187      172.217.10.69.443     ESTABLISHED
tcp4      0      0 10.0.75.29.62183      35.186.241.40.443     ESTABLISHED
tcp4      0      0 10.0.75.29.62171      140.82.113.25.443     ESTABLISHED
tcp4      0      0 10.0.75.29.62159      140.82.113.26.443     ESTABLISHED
tcp4      0      0 10.0.75.29.61429      31.13.71.1.443        ESTABLISHED
tcp4      0      0 10.0.75.29.61428      172.253.122.188.5228  ESTABLISHED
tcp4      0      0 10.0.75.29.60704      17.57.144.164.5223    ESTABLISHED
tcp4      0      0 10.0.75.29.60696      35.201.124.9.443      ESTABLISHED
tcp4      0      0 10.0.75.29.60330      31.13.71.1.443        ESTABLISHED
```

## Paths, directories and filenames

- import os.path module
- path is mapped to appropriate package for current os
- The os.path module provides many functions for working with paths.
- Some of the more common methods:
  - os.path.exists()
  - os.path.dirname()
  - os.path.basename
  - os.path.split()

**os.path** is the primary module for working with filenames and paths. There are many methods for getting and modifying a file or folder's path.

Also provide are methods for getting information about a file.

## Example

### paths.py

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

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

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

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

- ① Unix relative path
- ② Unix absolute path
- ③ Windows relative path
- ④ Windows UNC path
- ⑤ What platform are we on?
- ⑥ Just the folder name
- ⑦ Just the file (or folder) name
- ⑧ Is it an absolute path?
- ⑨ ~ is current user's home
- ⑩ ~NAME is NAME's home

***paths.py***

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

***paths.py*** (windows)

```
dirname(win_p1): \\marmoset\sharing\technology\docs\bonsai
dirname(win_p2): \\marmoset\sharing\technology\docs\bonsai
basename(win_p1): foo.doc
basename(win_p2): foo.doc
os.path.split(win_p1) Head: \\marmoset\sharing\technology\docs\bonsai Tail: foo.doc
os.path.split(win_p1) Head: bonsai Tail: foo.doc
os.path.splitunc(win_p1) Head: \\marmoset\sharing Tail:
\technology\docs\bonsai\foo.doc
os.path.splitunc(win_p1) Head: Tail: bonsai\foo.doc
```

## Walking directory trees

- Import os module
- Use the os.walk() iterator
- Returns tuple for each directory starting with the specified top directory
- Tuple contains full path to directory, list of subdirectories, and list of files \*syntax:

```
for currdir,subdirs,files in os.walk("start-dir"):
    pass
```

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

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

**TIP**

Remember to not use "dir" or "file" as variables when looping through the iterator, because they will overwrite builtins.

## Example

### **walk.py**

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

# count number of files and dirs in a directory tree
# note "files" includes devices, symbolic links, and pipes
import os
import sys

if sys.platform == 'win32': ①
    target = 'C:/Windows'
else:
    target = '/etc'

total_files = 0 ②
total_dirs = 0

for currdir, subdirs, files in os.walk(target): ③
    total_dirs += 1 # increment number of directories seen
    total_files += len(files) # add the number of files in this dir

print("{} contains {} dirs and {} files".format(target, total_dirs, total_files))
④
```

### **walk.py**

```
/etc contains 37 dirs and 343 files
```

**walk2.py**

```
#!/usr/bin/env python
"""
    find files whose size is greater than or equal to specified number of bytes
"""
import sys
import os

MINIMUM_SIZE = 1000

if len(sys.argv) < 2: ①
    print('Syntax: walk2.py START-DIR')
    sys.exit(1)

for currdir, subdirs, files in os.walk(sys.argv[1]):
    for file in files: ②
        fullpath = os.path.join(currdir, file) ③
        if os.path.isfile(fullpath): ④
            fsize = os.path.getsize(fullpath) ⑤
            if fsize >= MINIMUM_SIZE: ⑥
                print("{:40s} {:8d}".format(fullpath, fsize))
```

**walk2.py**

./xml_from_presidents.py	2458
./paramiko_remote_cmd.py	1272
./sa_movie_models.py	1349
./example.zip	56343
./xml_create_knights.py	1396
./threadexample.ui	2808
./pylintrc	14755
./db_mysql_metadata.py	1431
./new_magic.py	1387
./iterable_recipes.py	8359

...



## Grabbing data from the web

- import module urllib
- urlopen() similar to open()
- Iterate through (or read from) URL object
- Use info() method for metadata

Python makes grabbing web pages easy with the urllib module. The urllib.request.urlopen() method returns an HTTP response object (which also acts like a file object).

Iterating through this object returns the lines in the specified web page (the same lines you would see with "view source" in a browser).

Since the URL is opened in binary mode; you can use *response.read()* to download any kind of file which a URL represents – PDF, MP3, JPG, and so forth.

### NOTE

Grabbing web pages is even easier with the **requests** modules. See [read\\_html\\_requests.py](#) and [read\\_pdf\\_requests.py](#) in the EXAMPLES folder.

## Example

### read\_html\_urllib.py

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

import urllib.request

u = urllib.request.urlopen("https://www.python.org")

print(u.info()) ①
print()

print(u.read(500).decode()) ②
```

① .info() returns a dictionary of HTTP headers

② The text is returned as a bytes object, so it needs to be decoded to a string

*read\_html\_urllib.py*

```
Connection: close
Content-Length: 49014
Server: nginx
Content-Type: text/html; charset=utf-8
X-Frame-Options: DENY
Via: 1.1 vegur
Via: 1.1 varnish
Accept-Ranges: bytes
Date: Tue, 04 Feb 2020 20:03:44 GMT
Via: 1.1 varnish
Age: 2671
X-Served-By: cache-iad2151-IAD, cache-lga21952-LGA
X-Cache: HIT, HIT
X-Cache-Hits: 1, 7
X-Timer: S1580846625.959715,VS0,VE0
Vary: Cookie
Strict-Transport-Security: max-age=63072000; includeSubDomains

<!doctype html>
<!--[if lt IE 7]> <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9"> <![endif]-->
<!--[if IE 7]> <html class="no-js ie7 lt-ie8 lt-ie9"> <![endif]-->
<!--[if IE 8]> <html class="no-js ie8 lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--><html class="no-js" lang="en" dir="ltr"> <!--<![endif]-->

<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
```

...

## Example

### read\_pdf\_urllib.py

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

import sys
import os
from urllib.request import urlopen
from urllib.error import HTTPError

# url to download a PDF file of a NASA ISS brochure

url = 'https://www.nasa.gov/pdf/739318main_ISS%20Utilization%20Brochure%202012%20Screenres%203-8-13.pdf' ①

saved_pdf_file = 'nasa_iss.pdf' ②

try:
    URL = urlopen(url) ③
except HTTPError as e: ④
    print("Unable to open URL:", e)
    sys.exit(1)

pdf_contents = URL.read() ⑤
URL.close()

with open(saved_pdf_file, 'wb') as pdf_in:
    pdf_in.write(pdf_contents) ⑥

if sys.platform == 'win32': ⑦
    cmd = saved_pdf_file
elif sys.platform == 'darwin':
    cmd = 'open ' + saved_pdf_file
else:
    cmd = 'acroread ' + saved_pdf_file

os.system(cmd) ⑧
```

## Sending email

- use `smtplib`
- For attachments, use `email.mime.*`
- Can provide authentication
- Can work with proxies

It is easy to send a simple email message with Python. The `smtplib` module allows you to create and send the message.

To send an attachment, use `smtplib` plus one or more of the submodules of `email.mime`, which are needed to put the message and attachments in proper MIME format.

**TIP**

When sending attachments, be sure to use the `.as_string()` method on the MIME message object. Otherwise you will be sending binary gibberish to your recipient.

## Example

### email\_simple.py

```
#!/usr/bin/env python
from getpass import getpass ①
import smtplib ②
from email.message import EmailMessage ③
from datetime import datetime

TIMESTAMP = datetime.now().ctime() ④

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']
MESSAGE_SUBJECT = 'Python SMTP example'

MESSAGE_BODY = """
Hello at {}.

Testing email from Python
""".format(TIMESTAMP)

SMTP_USER = 'pythonclass'
SMTP_PASSWORD = getpass("Enter SMTP server password:") ⑤

smtpserver = smtplib.SMTP("smtp2go.com", 2525) ⑥
smtpserver.login(SMTP_USER, SMTP_PASSWORD) ⑦

msg = EmailMessage() ⑧
msg.set_content(MESSAGE_BODY) ⑨
msg['Subject'] = MESSAGE_SUBJECT ⑩
msg['from'] = SENDER ⑪
msg['to'] = RECIPIENTS ⑫

try:
    smtpserver.send_message(msg) ⑬
except smtplib.SMTPException as err:
    print("Unable to send mail:", err)
finally:
    smtpserver.quit() ⑭
```

① module for hiding password

- ② module for sending email
- ③ module for creating message
- ④ get a time string for the current date/time
- ⑤ get password (not echoed to screen)
- ⑥ connect to SMTP server
- ⑦ log into SMTP server
- ⑧ create empty email message
- ⑨ add the message body
- ⑩ add the message subject
- ⑪ add the sender address
- ⑫ add a list of recipients
- ⑬ send the message
- ⑭ disconnect from SMTP server

**email\_attach.py**

```
#!/usr/bin/env python
import smtplib
from datetime import datetime
from imghdr import what ①
from email.message import EmailMessage ②
from getpass import getpass ③

SMTP_SERVER = "smtp2go.com" ④
SMTP_PORT = 2525

SMTP_USER = 'pythonclass'

SENDER = 'jstrick@mindspring.com'
RECIPIENTS = ['jstrickler@gmail.com']

def main():
    smtp_server = create_smtp_server()
    now = datetime.now()
    msg = create_message(
        SENDER,
        RECIPIENTS,
        'Here is your attachment',
        'Testing email attachments from python class at {}\n\n'.format(now),
    )
    add_text_attachment('../DATA/parrot.txt', msg)
    add_image_attachment('../DATA/felix_auto.jpeg', msg)
    send_message(smtp_server, msg)

def create_message(sender, recipients, subject, body):
    msg = EmailMessage() ⑤
    msg.set_content(body) ⑥
    msg['From'] = sender
    msg['To'] = recipients
    msg['Subject'] = subject
    return msg

def add_text_attachment(file_name, message):
```

```
with open(file_name) as file_in: ⑦
    attachment_data = file_in.read()
message.add_attachment(attachment_data) ⑧

def add_image_attachment(file_name, message):
    with open(file_name, 'rb') as file_in: ⑨
        attachment_data = file_in.read()
    image_type = what(None, h=attachment_data) ⑩
    message.add_attachment(attachment_data, maintype='image', subtype=image_type)
    ⑪

def create_smtp_server():
    password = getpass("Enter SMTP server password:") ⑫
    smtpserver = smtplib.SMTP(SMTP_SERVER, SMTP_PORT) ⑬
    smtpserver.login(SMTP_USER, password) ⑭

    return smtpserver

def send_message(server, message):
    try:
        server.send_message(message) ⑮
    finally:
        server.quit()

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



- ① module to determine image type
- ② module for creating email message
- ③ module for reading password privately
- ④ global variables for external information (IRL should be from environment — command line, config file, etc.)
- ⑤ create instance of EmailMessage to hold message
- ⑥ set content (message text) and various headers
- ⑦ read data for text attachment
- ⑧ add text attachment to message
- ⑨ read data for binary attachment
- ⑩ get type of binary data
- ⑪ add binary attachment to message, including type and subtype (e.g., "image/jpg")
- ⑫ get password from user (don't hardcode sensitive data in script)
- ⑬ create SMTP server connection
- ⑭ log into SMTP connection
- ⑮ send message

## math functions

- use the math module
- Provides functions and constants

Python provides many math functions. It also provides constants pi and e.

*Table 16. Math functions*

<code>sqrt(x)</code>	Returns the square root of x
<code>exp(x)</code>	Return $e^x$
<code>log(x)</code>	Returns the natural log, i.e. $\ln x$
<code>log10(x)</code>	Returns the log to the base 10 of x
<code>sin(x)</code>	Returns the sine of x
<code>cos(x)</code>	Return the cosine of x
<code>tan(x)</code>	Returns the tangent of x
<code>asin(x)</code>	Return the arc sine of x
<code>acos(x)</code>	Return the arc cosine of x
<code>atan(x)</code>	Return the arc tangent of x
<code>fabs(x)</code>	Return the absolute value, i.e. the modulus, of x
<code>ceil(x)</code>	Rounds x (which is a float) up to next highest integer <sup>1</sup>
<code>floor(x)</code>	Rounds x (which is a float) down to next lowest integer
<code>degrees(x)</code>	converts angle x from radians to degrees
<code>radians(x)</code>	converts angle x from degrees to radians

### TIP

This table is not comprehensive – see [docs for math module](#) for some more functions.

For more math and engineering functions, see the external modules `numpy` and `scipy`.

## Random values

- Use the random module
- Useful methods
  - `random()`
  - `randint(start,stop)`
  - `randrange(start,limit)`
  - `choice(seq)`
  - `sample(seq,count)`
  - `shuffle(seq)`

The random module provides methods based on selected a random number. In addition to `random()`, which returns a fractional number between 0 and 1, there are a number of convenience functions.

`randint()` and `randrange()` return a random integer within a range of numbers; the difference is that `randint()` includes the endpoint of the specified range, and `randrange()` does not.

`choice()` returns one element from any of Python's sequence types; `sample()` is the same, but returns a specified number of elements.

`shuffle()` randomizes a sequence.

## Example

### random\_ex.py

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

import random

fruits = ["pomegranate", "cherry", "apricot", "date", "apple",
"lemon", "kiwi", "orange", "lime", "watermelon", "guava",
"papaya", "fig", "pear", "banana", "tamarind", "persimmon",
"elderberry", "peach", "blueberry", "lychee", "grape" ]

for i in range(1, 11):
    print("random():", random.random())
    print("randint(1, 2000):", random.randint(1, 2000))
    print("randrange(1, 5):", random.randrange(1, 5))
    print("choice(fruit):", random.choice(fruits))
    print("sample(fruit, 3):", random.sample(fruits, 3))
    print()
```

***random\_ex.py***

```
random(): 0.961566993949853
randint(1, 2000): 219
randrange(1, 5): 1
choice(fruit): lime
sample(fruit, 3): ['cherry', 'apple', 'pear']

random(): 0.41645436815435755
randint(1, 2000): 59
randrange(1, 5): 3
choice(fruit): banana
sample(fruit, 3): ['persimmon', 'kiwi', 'apple']

random(): 0.43479364451663594
randint(1, 2000): 1962
randrange(1, 5): 3
choice(fruit): lemon
sample(fruit, 3): ['pear', 'tamarind', 'kiwi']

random(): 0.8968839658737962
randint(1, 2000): 1924
randrange(1, 5): 2
choice(fruit): date
sample(fruit, 3): ['guava', 'peach', 'date']

random(): 0.5209150915625959
randint(1, 2000): 1155
randrange(1, 5): 1
choice(fruit): grape
sample(fruit, 3): ['lemon', 'persimmon', 'pear']
```

## Dates and times

- Use the datetime module
- Provides several classes
  - datetime
  - date
  - time
  - timedelta

Python provides the datetime module for manipulating dates and times. Once you have created date or time objects, you can combine them and extract the time units you need.

## Example

### datetime\_ex.py

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

from datetime import datetime, date, timedelta

print("date.today():", date.today()) ①

now = datetime.now() ②
print("now.day:", now.day) ③
print("now.month:", now.month)
print("now.year:", now.year)
print("now.hour:", now.hour)
print("now.minute:", now.minute)
print("now.second:", now.second)

d1 = datetime(2018, 6, 13) ④
d2 = datetime(2018, 8, 24)

d3 = d2 - d1 ⑤

print("raw time delta:", d3)
print("time delta days:", d3.days) ⑥

interval = timedelta(10) ⑦
print("interval:", interval)

d4 = d2 + interval ⑧
d5 = d2 - interval
print("d2 + interval:", d4)
print("d2 - interval:", d5)
print()

t1 = datetime(2016, 8, 24, 10, 4, 34) ⑨
t2 = datetime(2018, 8, 24, 22, 8, 1)
t3 = t2 - t1

print("datetime(2016, 8, 24, 10, 4, 34):", t1)
print("datetime(2018, 8, 24, 22, 8, 1):", t2)
print("time diff (t2 - t1):", t3)
```

***datetime\_ex.py***

```
date.today(): 2020-02-04
now.day: 4
now.month: 2
now.year: 2020
now.hour: 15
now.minute: 3
now.second: 45
raw time delta: 72 days, 0:00:00
time delta days: 72
interval: 10 days, 0:00:00
d2 + interval: 2018-09-03 00:00:00
d2 - interval: 2018-08-14 00:00:00

datetime(2016, 8, 24, 10, 4, 34): 2016-08-24 10:04:34
datetime(2018, 8, 24, 22, 8, 1): 2018-08-24 22:08:01
time diff (t2 - t1): 730 days, 12:03:27
```



## Zipped archives

- import zipfile for (PK)zipped files
- Get a list of files
- Extract files

The zipfile module allows you to read and write to zipped archives. In either case you first create a zipfile object; specifying a mode of "w" if you want to create an archive, and a mode of "r" (or nothing) if you want to read an existing zip file.

There are also modules for gzipped, bziped, and compressed archives.

### Example

#### zipfile\_ex.py

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

from zipfile import ZipFile, ZIP_DEFLATED
import os.path

# reading & extracting
rzip = ZipFile("../DATA/textfiles.zip") ①
print(rzip.namelist()) ②
ty = rzip.read('tyger.txt').decode() ③
print(ty[:50])
rzip.extract('parrot.txt') ④

# creating a zip file
wzip = ZipFile("example.zip", mode="w", compression=ZIP_DEFLATED) ⑤
for base in "parrot tyger knights alice poe_sonnet spam".split():
    filename = os.path.join("../DATA", base + '.txt')
    print("adding {} as {}".format(filename, base + '.txt'))
    wzip.write(filename, base + '.txt') ⑥
```

- ① Open zip file for reading
- ② Print list of members in zip file
- ③ Read (raw binary) data from member and convert from bytes to string
- ④ Extract member
- ⑤ Create new zip file
- ⑥ Add member to zip file

***zipfile\_ex.py***

```
['fruit.txt', 'parrot.txt', 'tyger.txt', 'spam.txt']  
    The Tyger
```

```
Tyger! Tyger! burning bright  
adding ../DATA/parrot.txt as parrot.txt  
adding ../DATA/tyger.txt as tyger.txt  
adding ../DATA/knights.txt as knights.txt  
adding ../DATA/alice.txt as alice.txt  
adding ../DATA/poe_sonnet.txt as poe_sonnet.txt  
adding ../DATA/spam.txt as spam.txt
```

## Chapter 13 Exercises

### Exercise 13-1 (`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 13-2 (`file_size.py`)

Write a script that accepts one or more files on the command line, and prints out the size, one file per line. If any argument is not a file, print out an error message.

**TIP** | You will need the `os.path` module.



# Chapter 14: An Introduction to Python Classes

## Objectives

- Understanding the big picture of O-O programming
- Defining a class and its constructor
- Creating object methods
- Adding attributes and properties to a class
- Using inheritance for code reuse
- Adding class data and methods

A class is the representation of an idea, a concept, in the code. An object of a class represents a particular example of the idea in the code. Without classes, a reader of the code would have to guess about the relationships among data items and functions - classes make such relationships explicit and "understood" by compilers. With classes, more of the high-level structure of your program is reflected in the code, not just in the comments.

— Bjarne Stroustrup (creator of C++)

## About O-O programming

- Classes are modules that can create objects
- Objects contain data and methods

Python is an object-oriented language. It supports the creation of classes, which are modules that can create objects.

Objects contain both data and the methods that operate on the data. Each object created has its own personal data, called instance data. It can also have data that is shared with all the objects created from its class, called class or static data.

Each class defines an initializer, which initializes and returns an object.

Objects may inherit attributes (data and methods) from other objects.

Methods are not polymorphic; i.e., you can't define multiple versions of a method, with different signatures, and have the corresponding method selected at runtime. However, because Python has dynamic typing, this is seldom needed.

## Defining classes

- Use the **class** statement **Syntax**

```
class ClassName(baseclass):  
    pass
```

To create a class, declare the class with the **class** statement. Any base classes may be specified in parentheses, but are not required.

Classes are conventionally named with UpperCamelCase (i.e., all words, including the first, are capitalized). This is also known as CapWords, StudlyCaps, etc. Modules conventionally have lower-case names. Thus, it is usual to have module `rocketengine` containing class `RocketEngine`.

All methods, including the constructor, are passed the object itself. This is conventionally named "self", and while this is not mandatory, most Python programmers expect it.

The basic layout is this:

```
class ClassName(baseclass):  
    classvar = value  
  
    def __init__(self,...):  
        self._attrib = instancevalue;  
        ClassName.attrib = classvalue;  
  
    def method1(self,...):  
        self._attrib = instancevalue  
  
    def method2(self,...):  
        x = self.method1()
```

## Example

### simple\_class.py

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

class Simple(): ①
    def __init__(self, message_text): ②
        self._message_text = message_text ③

    def text(self): ④
        return self._message_text

if __name__ == "__main__":
    msg1 = Simple('hello') ⑤
    print(msg1.text()) ⑥

    msg2 = Simple('hi there') ⑦
    print(msg2.text())
```

- ① default base class is **object**
- ② constructor
- ③ message text stored in instance object
- ④ instance method
- ⑤ instantiate an instance of Simple
- ⑥ call instance method
- ⑦ create 2nd instance of Simple

### simple\_class.py

```
hello
hi there
```



## Constructors

- Constructor is named ***`_init_`***
- AKA initializer
- Passed *self* plus any parameters

A class's constructor (also known as the initializer) is named *init*. It receives the object being created, and any parameters passed into the initializer in the code as part of instantiation.

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

It is also normal to name data elements (variables) of a class with a leading underscore to indicate (in a non-mandatory way) that the variable is private. Access to private variables should be provided via public access methods (AKA getters) or properties.

## Instance methods

- Expect the object as first parameter
- Object conventionally named *self*
- Otherwise like normal Python functions
- Use *self* to access instance attributes or methods
- Use class name to access class data

Instance methods are defined like normal functions, but like constructors, the object that the method is called from is passed in as a parameter. Like the constructor, it should be named *self*.

## Example

### animal.py

```
#!/usr/bin/env python
class Animal():
    count = 0 ①

    def __init__(self, species, name, sound):
        self._species = species
        self._name = name
        self._sound = sound
        Animal.count += 1

    @property
    def species(self):
        return self._species

    @classmethod
    def kill(cls):
        cls.count -= 1

    @property
    def name(self):
        return self._name

    def make_sound(self):
        print(self._sound)

    @classmethod
    def remove(cls):
        cls.count -= 1 ②

    @classmethod
    def zoo_size(cls): ③
        return cls.count

if __name__ == "__main__":
    leo = Animal("African lion", "Leo", "Roarrrrrrr")
    garfield = Animal("cat", "Garfield", "Meowwww")
    felix = Animal("cat", "Felix", "Meowwww")
```

```
print(leo.name, "is a", leo.species, "--", end=' ')
leo.make_sound()

print(garfield.name, "is a", garfield.species, "--", end=' ')
garfield.make_sound()

print(felix.name, "is a", felix.species, "--", end=' ')
felix.make_sound()
```

- ① class data
- ② update class data from instance
- ③ zoo\_size gets class object when called from instance or class

### ***animal.py***

```
Leo is a African lion -- Roarrrrrrrr
Garfield is a cat -- Meowwwwww
Felix is a cat -- Meowwwwww
```

# Properties

- Properties are managed attributes
- Create with `@property` decorator
- Create getter, setter, deleter, docstring
- Specify getter only for read-only property

An object can have properties, or managed attributes. When a property is evaluated, its corresponding getter method is invoked; when a property is assigned to, its corresponding setter method is invoked.

Properties can be created with the `@property` decorator and its derivatives. `@property` applied to a method causes it to be a "getter" method for a property with the same name as the method.

Using `@name.setter` on a method with the same name as the property creates a setter method, and `@name.deleter` on a method with the same name creates a deleter method.

## Example

### properties.py

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

class Person():

    def __init__(self, firstname=None, lastname=None):
        self._first_name = None
        self._last_name = None
        self.first_name = firstname ①
        self.last_name = lastname

    @property
    def first_name(self): ②
        return self._first_name

    @first_name.setter ③
    def first_name(self, value): ④
        if value is None or value.isalpha():
            self._first_name = value
        else:
            raise ValueError("First name may only contain letters")

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if value is None or value.isalpha():
            self._last_name = value
        else:
            raise ValueError("Last name may only contain letters")

if __name__ == '__main__':
    person1 = Person('Ferneater', 'Eulalia')

    person2 = Person()
    person2.last_name = 'Pepperpot' ⑤
    person2.first_name = 'Hortense'
```

```
print("{} {}".format(person1.first_name, person1.last_name))
print("{} {}".format(person2.first_name, person2.last_name))

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

- ① calls property
- ② getter property
- ③ decorator comes from getter property
- ④ setter property
- ⑤ access property

### ***properties.py***

```
Ferneater Eulalia  
Hortense Pepperpot  
First name may only contain letters
```



## Class methods and data

- Defined in the class, but outside of methods
- Defined as attribute of class name (similar to self)
- Define class methods with `@classmethod`
- Class methods get the class object as 1st parameter

Most classes need to store some data that is common to all objects created in the class. This is generally called class data.

Class attributes can be created by using the class name directly, or via class methods.

A class method is created by using the `@classmethod` decorator. Class methods are implicitly passed the class object.

Class methods can be called from the class object or from an instance of the class; in either case the method is passed the class object.

## Example

### class\_methods\_and\_data.py

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog" ①

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

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon)) ②

    @classmethod ③
    def get_location(cls): ④
        return cls.LOCATION ⑤

r = Rabbit("a nice cup of tea")
print(Rabbit.get_location()) ⑥
print(r.get_location()) ⑦
```

- ① class data (not duplicated in instances)
- ② instance method
- ③ the **@classmethod** decorator makes a function receive the class object, not the instance object
- ④ `*get_location()` is a *class* method
- ⑤ class methods can access class data via **cls**
- ⑥ call class method from class
- ⑦ call class method from instance

### class\_methods\_and\_data.py

```
the Cave of Caerbannog
the Cave of Caerbannog
```

## Static Methods

- Define with `@staticmethod`

A static method is a utility method that is included in the API of a class, but does not require either an instance or a class object. Static methods are not passed any implicit parameters.

Many classes do not need any static methods.

Define static methods with the `@staticmethod` decorator.

### Example

```
class Spam():  
  
    @staticmethod  
    def format_as_title(s): # no implicit parameters  
        return s.strip().title()
```

## Private methods

- Called by other methods in the class
- Not visible to users of the class
- Conventionally named with leading underscore

Private methods are those that are called only within the class. They are not part of the API – they are not visible to users of the class. Private methods may be instance, class, or static methods; if calling a private instance method, pass the instance variable (`self`) explicitly.

It is conventional to name a private method with a leading underscore. This does not protect it from use, but gives programmers a hint that it's for internal use only.

# Inheritance

- Specify base classes after class name
- Multiple inheritance OK
- Depth-first, left-to-right search for methods not in derived class

Classes may inherit methods and data from one or more other classes by specifying a parenthesized list of base classes after the class name in the class definition.

If a method or attribute is not found in the derived class, it is first sought in the first base class in the list. If not found, it is sought in the base class of that class, if any, and so on. This is usually called a depth-first search.

The derived class inherits all attributes of the base class. If the base class initializer takes the same arguments as the derived class, then no extra coding is needed. Otherwise, to explicitly call the initializer in the base class, use `super(ThisClass,self).init(args)`.

The simplest derived class would be:

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

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

## Example

### mammal.py

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

from animal import Animal

class Mammal(Animal): ①
    def __init__(self, species, name, sound, gestation):
        super(Mammal, self).__init__(species, name, sound)
        self._gestation = gestation

    @property
    def gestation(self): ②
        """Length of gestation period in days"""
        return self._gestation

if __name__ == "__main__":
    mammal1 = Mammal("African lion", "Bob", "Roarrrr", 120)
    print(mammal1.name, "is a", mammal1.species, "--", end=' ')
    mammal1.make_sound()

    print("Number of animals", mammal1.zoo_size())

    mammal2 = Mammal("Fruit bat", "Freddie", "Squeak!!", 180)
    print(mammal2.name, "is a", mammal2.species, "--", end=' ')
    mammal2.make_sound()

    print("Number of animals", mammal2.zoo_size())
    print("Number of animals", Mammal.zoo_size())

    mammal1.kill()
    print("Number of animals", Mammal.zoo_size())

    print("Gestation period of the", mammal1.species, "is", mammal1.gestation,
          "days")
    print("Gestation period of the", mammal2.species, "is", mammal2.gestation,
          "days")
```

- ① inherit from Animal
- ② add property to existing Animal properties

***mammal.py***

```
Bob is a African lion -- Roarrrr
Number of animals 1
Freddie is a Fruit bat -- Squeak!!
Number of animals 2
Number of animals 2
Number of animals 1
Gestation period of the African lion is 120 days
Gestation period of the Fruit bat is 180 days
```

## Untangling the nomenclature

There are many terms to distinguish the various parts of a Python program. This chart is an attempt to help you sort out what is what:

*Table 17. Objected-oriented Nomenclature*

attribute	A variable or method that is part of a class or object
base class	A class from which other classes inherit
child class	Same as derived class
class	A Python module from which objects may be created
class method	A function that expects the class object as its first parameter. Such a function can be called from either the class itself or an instance of the class. Created with <code>@classmethod</code> decorator.
derived class	A class which inherits from some other class
function	An executable subprogram.
instance method	A function that expects the instance object, conventionally named <code>self</code> , as its first parameter. See "method".
method	A function defined inside a class.
module	A file containing python code, and which is designed to be imported into Python scripts or other modules.
package	A folder containing one or more modules. Packages may be imported. There must be a file named <code>init.py</code> in the package folder.
parent class	Same as base class
property	A managed attribute (variable) of an instance of a class
script	A Python program. A script is an executable file containing Python commands.
static method	A function in a class that does not automatically receive any parameters; typically used for private utility functions. Created with <code>@staticmethod</code> decorator.
superclass	Same as base class



## Chapter 14 Exercises

### Exercise 14-1 (knight.py, knight\_info.py)

*Part 1:*

Create a module which defines a class named Knight.

The initializer for the class should expect the knight's name as a parameter. Get the information from the file knights.txt to initialize the object.

The object should have these (read-only) properties:

name  
title  
favorite\_color  
quest  
comment

#### Example

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

*Part 2:*

Create an application to use the Knight class created in part one. For each knight specified on the command line, create a knight object and print out the knight's name, favorite color, quest, and comment. Precede the name with the knight's title.

**Example output:**

```
kinfo.py Arthur Bedevere
Name: King Arthur
Favorite Color: blue
Quest: The Grail
Comment: King of the Britons

Name: Sir Bedevere
Favorite Color: red, no, blue!
Quest: The Grail
Comment: AARRRRRRRGGGGHH
```

# Appendix A: Python Bibliography

Title	Author	Publisher
<b>Data Science</b>		
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.
<b>Design Patterns</b>		

<b>Title</b>	<b>Author</b>	<b>Publisher</b>
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
<b>General Python development</b>		
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
<b>Misc</b>		
Python Geospatial Development	Erik Westra	Packt Publishing

<b>Title</b>	<b>Author</b>	<b>Publisher</b>
Python High Performance Programming	Gabriele Lanaro	Packt Publishing
<b>Networking</b>		
Python Network Programming Cookbook	Dr. M. O. Faruque Sarker	Packt Publishing
Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers	T J O'Connor	Syngress
Web Scraping with Python	Ryan Mitchell	O'Reilly & Assoc.
<b>Testing</b>		
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
<b>Web Development</b>		
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

<b>Title</b>	<b>Author</b>	<b>Publisher</b>
Flask Web Development	Miguel Grinberg	O'Reilly & Assoc.
Full Stack Python (e-book only)	Matt Makai	Gumroad (or free download)
Full Stack Python Guide to Deployments (e-book only)	Matt Makai	Gumroad (or free download)
High Performance Django	Peter Baumgartner, Yann Malet	Lincoln Loop
Instant Flask Web Development	Ron DuPlain	Packt Publishing
Learning Flask Framework	Matt Copperwaite, Charles O Leifer	Packt Publishing
Mastering Flask	Jack Stouffer	Packt Publishing
Two Scoops of Django: Best Practices for Django 1.11	Daniel Roy Greenfeld, Audrey Roy Greenfeld	Two Scoops Press
Web Development with Django Cookbook	Aidas Bendoraitis	Packt Publishing





# Index

@

@property, 301  
@staticmethod, 307  
\_\_pycache\_\_, 227

*A*

Amoeba, 10  
Anaconda bundle, 16  
Array types, 88

*B*

Boolean operators, 79  
break statement, 82  
builtin functions, 37  
    table, 38

*C*

class  
    constructor, 297  
    definition, 295  
    instance methods, 298  
    private method, 308  
    static method, 307  
class data, 305  
class method, 305  
command line parameters, 66  
conditional expression, 76  
continue statement, 82

*D*

Dagobah, 19  
date, 286  
dates and times, 286  
datetime, 286

*E*

Eclipse, 223  
Explicit execution, 27

editors and IDEs, 32  
email.mime, 276  
enumerate(), 108  
exceptions, 207  
    else, 212  
    finally, 214  
    generic, 210  
    ignoring, 211  
    list, 218  
    multiple, 209

*F*

file(), 137  
flow control, 72  
for loop, 97  
format, 61  
formatting, 61  
functions  
    definition, 166  
    keyword parameters, 174  
    keyword-only parameters, 173  
    named parameters, 174  
    optional parameters, 172  
    parameter types, 170  
    positional parameters, 171  
    returning values, 169

*G*

Guido van Rossum, 10, 17  
global statement, 182  
global variables, 178  
grabbing data from the web, 273

*H*

HTTP download, 273  
hello, world, 33  
help on Python, 18

*I*

- if statement, 74
- if-else, 76
- if/elif/else, 74
- implicit execution
  - Windows, 28
  - non-Windows, 29
- import statement, 226
- in, 111
- indentation, 73
- indexing, 94
- inheritance, 309
- instance methods, 298
- interpreted languages, 14
- interpreter attributes, 262
- iterable, 101
- iterating through a sequence, 97

*K*

- keywords, 37

*L*

- Linux, 16
- Luke Skywalker, 19
- lambda functions, 193
- launching external programs, 264
- legacy string formatting, 65
- len(), 105
- list methods
  - table, 91
- lists, 90
- literal Unicode characters, 44
- local variables, 178

*M*

- Mac OS X, 16
- MacOS, 16
- Math operators and expressions, 55
- Monty Python, 12
- math functions, 282
- math operators

- table, 57
- max(), 105
- min(), 105
- modules, 222
  - alias, 231
  - creating, 223
  - definition, 222
  - importing, 226
  - naming, 223
  - search path, 228

*N*

- None, 36
- nested sequences, 102
- numeric literals, 53

*O*

- OS X, 16
- object-oriented programming, 294
- os.path, 266
- os.system(), 264
- os.walk(), 270

*P*

- PATH variable
  - Linux, 24
  - OS X, 24
  - Unix, 24
  - Windows, 24
  - non-Windows, 24
- PYTHONPATH, 228
- Perl, 236
- PyCharm, 32, 223
- PyPI, 232
- Python 2.7 end-of-life, 17
- Python IDEs, 32
- Python Interpreter, 25
- Python Package Index, 232
- Python advantages, 15
- Python disadvantages, 15
- Python home page, 16

- Python scripts, 27
- Python timeline, 13
- Python version, 16
- package
  - alias, 231
- packages, 230
- pip, 232
- popen(), 264
- print() function, 59
- private method, 308
- properties, 301
- R*
- Reading from the keyboard, 67
- Regular Expression Metacharacters
  - table, 238
- Relational Operators, 77
- random, 283
- random.choice(), 283
- random.randint(), 283
- random.randrange(), 283
- random.sample(), 283
- random.shuffle(), 283
- raw strings, 43
- re.compile(), 242
- re.findall(), 239
- re.finditer(), 239
- re.search(), 239
- read(), 137
- readline(), 137
- readlines(), 137
- regular expressions, 236
  - about, 236
  - atoms, 237
  - branches, 237
  - compilation flags, 245-246
  - finding matches, 239
  - grouping, 249
  - re objects, 242
  - replacing text, 254
  - replacing text with callback, 256
  - special groups, 252
  - splitting text, 259
  - syntax overview, 237
- reversed(), 105
- S*
- Sequences, 88
- scripts vs. modules, 222
- self, 298
- sending email, 276
- sequence functions, 105
- sh, 264
- slicing, 94
- smtplib, 276
- sorted(), 105, 188
- sorting
  - custom key, 190
  - dictionaries, 200
  - in place, 203
  - nested data, 196
  - reverse, 202
- standard I/O, 263
- standard exception hierarchy, 218
- starting python, 22
- static method, 307
- stderr, 263
- stdin, 263
- stdio, 263
- stdout, 263
- string formatting, 61
- string literals, 41
- string methods, 47
- string operators, 47
- strings, 40
- subprocess, 264
- sum(), 105
- syntax errors, 206
- sys module, 262
- sys.executable, 262
- sys.modules, 262
- sys.path, 262

sys.platform, 262

sys.prefix, 262

sys.version, 262

## *T*

time, 286

timedelta, 286

triple-delimited strings, 42

triple-quoted strings, 42

tuple unpacking, 101

type conversions, 58

## *U*

unicode, 40

urllib module, 273

urlopen(), 273

using pydoc, 31

using try/except, 208

## *V*

variable scope, 178

variable typing, 39

variables, 36

## *W*

Windows, 16

walking directory trees, 270

while loop, 81

whitespace, 73

write(), 137

writelines(), 137

## *Y*

Yoda, 19

## *Z*

zip(), 105

zipfile, 289

zipped archives, 289