

# Python Programming Boot Camp

***TTPS4814***

John Strickler

Version 1.3, September 2024

# Table of Contents

About Python Programming Boot Camp: TTPS4814 .....	1
Course Outline .....	2
Student files .....	3
Examples .....	4
Appendices .....	5
Classroom etiquette .....	6
Chapter 1: The Python Environment .....	7
Starting Python .....	8
If the interpreter is not in your PATH .....	9
Using the interactive interpreter .....	10
Trying out a few commands .....	11
Running Python scripts .....	12
Using the <code>help()</code> function .....	13
Python Editors and IDEs .....	15
For more information .....	16
Chapter 2: Variables and values .....	18
Using variables .....	19
Keywords and builtin names .....	20
Variable typing .....	22
Strings .....	23
Single-delimited string literals .....	24
Triple-delimited string literals .....	25
Raw string literals .....	27
Unicode characters .....	29
String operators and methods .....	32
Numeric literals .....	38
Math operators and expressions .....	40
Converting among types .....	43
Chapter 3: Basic input and output .....	46
Writing to the screen .....	47
Format strings .....	49
Using the <code>.format()</code> method .....	53
Legacy String Formatting .....	55
<code>str()</code> vs <code>repr()</code> .....	58
f-string shortcut .....	59
Command line arguments .....	61

Reading from the keyboard .....	63
Chapter 4: Flow Control .....	65
About flow control .....	66
if and elif .....	67
White space .....	68
Determining Boolean values .....	69
The Conditional Expression .....	70
Relational Operators .....	71
Boolean operators .....	73
while loops .....	77
Loop control .....	78
Chapter 5: Array Types .....	82
About Array Types .....	83
Lists .....	85
Indexing and slicing .....	89
Iterating through a sequence .....	93
Tuples .....	96
Iterable Unpacking .....	98
Nested sequences .....	101
Operators and keywords for sequences .....	103
Functions for all sequences .....	105
Iterators .....	107
List comprehensions .....	112
Generator Expressions .....	115
Chapter 6: Working with Files .....	121
Text file I/O .....	122
Opening a text file .....	123
Reading a text file .....	125
Processing files from the command line .....	129
Writing to a text file .....	131
Modifying text files .....	132
Chapter 7: Dictionaries and Sets .....	135
About dictionaries .....	136
Creating dictionaries .....	137
Getting dictionary values .....	140
Iterating over a dictionary .....	143
Reading file data into a dictionary .....	145
Counting with dictionaries .....	147

About sets .....	149
Creating Sets .....	150
Working with sets .....	151
Chapter 8: Functions, modules, and packages .....	157
Functions .....	158
Function parameters .....	160
Default parameters .....	164
Every type of function parameter .....	165
Detailed Python Function parameter behavior (from PEP 570 and PEP 3102) .....	166
Name resolution (AKA Scope) .....	168
The global statement .....	170
Modules .....	171
Using import .....	172
How <code>import *</code> can be dangerous .....	177
Module search path .....	179
Executing modules as scripts .....	180
Packages .....	182
Configuring import with <code>__init__.py</code> .....	185
Documenting modules and packages .....	187
Python style .....	188
Chapter 9: Errors and Logging .....	190
Exceptions .....	191
Handling exceptions with <code>try</code> .....	192
Handling multiple exceptions .....	193
Handling generic exceptions .....	194
Ignoring exceptions .....	195
Using <code>else</code> .....	196
Cleaning up with <code>finally</code> .....	198
Simple Logging .....	202
Formatting log entries .....	204
Logging exception information .....	208
For more information .....	210
Chapter 10: Introduction to Python Classes .....	212
About object-oriented programming .....	213
Defining classes .....	214
Constructors .....	223
Instance methods .....	224
Properties .....	225

Class methods and data .....	228
Static Methods .....	230
Private methods .....	231
Inheritance .....	232
Untangling the nomenclature .....	235
Chapter 11: Metaprogramming .....	238
Metaprogramming .....	239
globals() and locals() .....	240
The inspect module .....	243
Working with attributes .....	247
Adding instance methods .....	249
Callable classes .....	252
Decorators .....	254
Applying decorators .....	256
Trivial Decorator .....	259
Decorator functions .....	260
Decorator Classes .....	263
Decorator arguments .....	266
Creating classes at runtime .....	268
Monkey Patching .....	271
Do you need a Metaclass? .....	274
About metaclasses .....	275
Mechanics of a metaclass .....	276
Singleton with a metaclass .....	280
Chapter 12: Type Hinting .....	285
Type Hinting .....	286
Static Analysis Tools .....	287
Runtime Analysis Tools .....	288
typing Module .....	290
Input Types .....	291
Variance .....	293
Union and Optional .....	295
multimethod and functools.singledispatch .....	297
Stub Type Hinting .....	299
Chapter 13: Developer Tools .....	301
Program development .....	302
Comments .....	303
pylint .....	304

Customizing pylint	305
Using pyreverse	306
The Python debugger	309
Starting debug mode	310
Profiling	313
Benchmarking	315
For more information	318
Chapter 14: Consuming RESTful Data	320
The REST API	321
When is REST not REST?	323
Consuming REST APIs	324
Printing JSON	329
Requests sessions	330
Authentication with requests	333
Posting data to a RESTful server	336
Other operations	339
Using Postman	343
Chapter 15: Database Access	345
The DB API	346
Connecting to a Server	348
Creating a Cursor	351
Querying data	352
Non-query statements	355
SQL Injection	358
Parameterized Statements	360
Metadata	366
Dictionary Cursors	369
Generic alternate cursors	373
Transactions	377
Object-relational Mappers	379
NoSQL	380
Chapter 16: Serializing Data	383
Which XML module to use?	384
Getting Started With ElementTree	385
How ElementTree Works	386
Elements	387
Creating a New XML Document	390
Parsing An XML Document	393

Navigating the XML Document .....	394
Using XPath .....	398
About JSON .....	401
Reading JSON .....	402
Writing JSON .....	405
Customizing JSON .....	407
Reading and writing YAML .....	410
Reading CSV data .....	415
Customizing CSV readers and writers .....	416
Using csv.DictReader .....	420
Writing CSV Data .....	422
Pickle .....	424
<b>Extra topics</b> .....	428
Chapter 17: Virtual Environments .....	429
Why do we need virtual environments? .....	430
What are virtual environments? .....	431
Preparing the virtual environment .....	432
Creating the environment .....	432
Activating the environment .....	433
Deactivating the environment .....	434
Freezing the environment .....	435
Duplicating an environment .....	436
The pipenv/conda/virtualenv/PyCharm swamp .....	437
Chapter 18: Effective Scripts .....	439
Using <b>glob</b> .....	440
Using shlex.split() .....	442
The subprocess module .....	443
subprocess convenience functions .....	444
Capturing stdout and stderr .....	447
Permissions .....	451
Using shutil .....	453
Creating a useful command line script .....	455
Creating filters .....	456
Parsing the command line .....	459
Simple Logging .....	464
Formatting log entries .....	466
Logging exception information .....	469
Logging to other destinations .....	471

Chapter 19: Regular Expressions .....	474
Regular expressions .....	475
RE syntax overview .....	476
Finding matches .....	478
RE objects .....	481
Compilation flags .....	484
Working with embedded newlines .....	488
Groups .....	491
Special groups .....	494
Replacing text .....	496
Replacing with backrefs .....	498
Replacing with a callback .....	501
Splitting a string .....	503
Appendix A: Where do I go from here? .....	506
Resources for learning Python .....	506
Appendix B: Field Guide to Python Expressions .....	508
Appendix C: String Formatting .....	509
Overview .....	509
Parameter Selectors .....	510
f-strings .....	512
Data types .....	513
Field Widths .....	516
Alignment .....	519
Fill characters .....	522
Signed numbers .....	524
Parameter Attributes .....	527
Formatting Dates .....	529
Run-time formatting .....	533
Miscellaneous tips and tricks .....	535
Appendix D: Python Bibliography .....	537
Index .....	540



# About Python Programming Boot Camp: TTPS4814

# Course Outline

## Day 1

**Chapter 1** [The Python Environment](#)

**Chapter 2** [Variables and values](#)

**Chapter 3** [Basic input and output](#)

**Chapter 4** [Flow Control](#)

## Day 2

**Chapter 5** [Array Types](#)

**Chapter 6** [Working with Files](#)

**Chapter 7** [Dictionaries and Sets](#)

**Chapter 8** [Functions, modules, and packages](#)

## Day 3

**Chapter 9** [Errors and Logging](#)

**Chapter 10** [Introduction to Python Classes](#)

**Chapter 11** [Metaprogramming](#)

**Chapter 12** [Type Hinting](#)

## Day 4

**Chapter 13** [Developer Tools](#)

**Chapter 14** [Consuming RESTful Data](#)

**Chapter 15** [Database Access](#)

**Chapter 16** [Serializing Data](#)

## Extra topics

**Chapter 17** [Virtual Environments](#)

**Chapter 18** [Effective Scripts](#)

**Chapter 19** [Regular Expressions](#)



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

## Student files

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

What's in the files?

**pyschwab** contains all files necessary for the class

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

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

**pyschwab/DATA/** contains data used in examples and answers

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

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

The following folders *may* be present:

**pyschwab/BIG\_DATA/** contains large data files used in examples and answers

**pyschwab/NOTEBOOKS/** Jupyter notebooks for use in class

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



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

# Examples

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

It will look like this:

## Example

### cmd\_line\_args.py

```
import sys    # Import the sys module

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

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

*cmd\_line\_args.py apple mango 123*

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

# Appendices

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

**Appendix B** [Field Guide to Python Expressions](#)

**Appendix C** [String Formatting](#)

**Appendix D** [Python Bibliography](#)

# Classroom etiquette

## Remote learning

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

## In-person learning

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

Please turn off cell phone ringers and other noisemakers.

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

We're all adults here; feel free to leave the 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.



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

# Chapter 1: The Python Environment

## Objectives

- Using the interactive interpreter
- Running scripts
- Getting help
- Learning about editors and IDEs

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

— Guido van Rossum

# Starting Python

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

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

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

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

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



# If the interpreter is not in your PATH

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

## Use the Anaconda Prompt

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

## Type the full path

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

## Add the directory to PATH temporarily

Windows (from a command prompt)

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

Linux/Mac (from a terminal window)

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

## Add the directory to PATH permanently

Windows

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

Linux/Mac

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

The command should look something like

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

## Using the interactive interpreter

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

The interactive prompt is `>>>`. You can type in any Python command or expression at this prompt.

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

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

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



The interpreter does autocomplete when you press `Tab`

Table 1. Mac/Linux command line editing

Key binding	Function
<code>^P</code>	Previous command
<code>^N</code>	Next command
<code>^F</code>	Forward 1 character
<code>^B</code>	Back 1 character
<code>^A</code>	Beginning of line
<code>^E</code>	End of line
<code>^D</code>	Delete character under cursor
<code>^K</code>	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()`. If you type an expression (variable or combination of variables, values, and operators), the interpreter will display its value.

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

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



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

## Running Python scripts

- Use Python interpreter
- Same for all platforms

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

```
python myscript.py
```

```
python myscript.py apple 123 banana
```

This will work on any platform.



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

## Using the `help()` function

From the Python interpreter

Type

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

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

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

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

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

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

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

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

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

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



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



if `pydoc` does not run from the command line, try `python -m pydoc`.

## From iPython

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

```
In [1]: len?  
Signature: len(obj, /)  
Docstring: Return the number of items in a container.  
Type:      builtin_function_or_method
```

## Python Editors and IDEs

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

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

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

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

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

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

## For more information

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



# Chapter 1 Exercises

## Exercise 1-1 (hello.py)

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

Run the script from the command line (or Ananconda prompt):

```
python hello.py
```

# Chapter 2: Variables and values

## Objectives

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

# Using variables

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

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

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

A variable *must* be assigned a value. A value of `None` (null) may be assigned if no particular value is needed. It is good practice to make variable names consistent. The Python style guide [Pep 8](#) suggests:

```
all_lower_case_words_spelled_out_with_underscores
```

## Example

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

## Keywords and builtin names

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

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

### Python keywords

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

### Builtin functions and classes

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



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

*Table 2. Python Builtin functions*

<code>abs()</code>	<code>aiter()</code>	<code>all()</code>
<code>any()</code>	<code>anext()</code>	<code>ascii()</code>
<code>bin()</code>	<code>breakpoint()</code>	<code>callable()</code>
<code>chr()</code>	<code>compile()</code>	<code>delattr()</code>
<code>dir()</code>	<code>divmod()</code>	<code>eval()</code>
<code>exec()</code>	<code>format()</code>	<code>getattr()</code>
<code>globals()</code>	<code>hasattr()</code>	<code>hash()</code>
<code>help()</code>	<code>hex()</code>	<code>id()</code>
<code>input()</code>	<code>isinstance()</code>	<code>issubclass()</code>
<code>iter()</code>	<code>len()</code>	<code>locals()</code>
<code>max()</code>	<code>min()</code>	<code>next()</code>
<code>oct()</code>	<code>open()</code>	<code>ord()</code>
<code>pow()</code>	<code>print()</code>	<code>repr()</code>
<code>round()</code>	<code>setattr()</code>	<code>sorted()</code>
<code>sum()</code>	<code>vars()</code>	<code>__import__()</code>

*Table 3. Python builtin classes*

<code>bool()</code>	<code>bytearray()</code>	<code>bytes()</code>
<code>classmethod()</code>	<code>complex()</code>	<code>dict()</code>
<code>enumerate()</code>	<code>filter()</code>	<code>float()</code>
<code>frozenset()</code>	<code>int()</code>	<code>list()</code>
<code>map()</code>	<code>memoryview()</code>	<code>object()</code>
<code>property()</code>	<code>range()</code>	<code>reversed()</code>
<code>set()</code>	<code>slice()</code>	<code>staticmethod()</code>
<code>str()</code>	<code>super()</code>	<code>tuple()</code>
<code>type()</code>	<code>zip()</code>	

# Variable typing

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

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

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

Certain operations are only valid with appropriate types.



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

## Example

### `variable_typing.py`

```
a = "123"
b = 456

result = a + b # raises error due to incompatible types

print(result)
```

### `variable_typing.py`

```
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/variable_typing.py", line 4,
in <module>
    result = a + b # raises error due to incompatible types
           ~~~~
TypeError: can only concatenate str (not "int") to str
```

# Strings

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

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

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

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

## Single-delimited string literals

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

Single-delimited (AKA "single-quoted") strings are enclosed in a pair of single or double quotes.

Escape codes, which start with a backslash, are interpreted specially. This makes it possible to include control characters such as *tab* (`\t`) and *newline* (`\n`) in a string.

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

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

### Example

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



# Triple-delimited string literals

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

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

Triple-delimited text is used for text containing quote characters as well as for documentation and boiler-plate text.

## Example

### string\_literals.py

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

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

query = """
select *
from customers
where state == "MT"
order by city
"""
```

***string\_literals.py***

Guido's the ex-BDFL of Python

Guido is the ex-"BDFL" of Python

Guido's the ex-"BDFL" of Python

## Raw string literals

- Start with `r`
- Do not interpret backslashes

If a string literal starts with `r` before the opening quotes, then it is a raw string literal. Backslashes are not interpreted.

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



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

## Example

### raw\_strings.py

```

regex = r"the\b\b\bend"
file_path = r"c:\temp"
message = r"please put a newline character (\n) after each line"

print("** raw strings **")
print(regex)
print(file_path)
print(message)
print()

regex = "the\b\b\bend"
file_path = "c:\temp"
message = "please put a newline character (\n) after each line"

print("** non-raw **")
print(regex)
print(file_path)
print(message)

```

### raw\_strings.py

```

** raw strings **
the\b\b\bend
c:\temp
please put a newline character (\n) after each line

** non-raw **
the\b\b\bend
c:  emp
please put a newline character (
) after each line

```

# Unicode characters

- Use `\uXXXX` to specify non-ASCII Unicode characters
  - `XXXX` is Unicode *codepoint* value in hex
- Use `\UXXXXXXXX` if codepoint > 0xFFFF

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

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

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



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

## Example

### unicode.py

```
print('26\u00B0') # Use \uXXXX where XXXX is the Unicode value in hex
print('26\N{DEGREE SIGN}') # use Unicode name
print(r'26\u00B0\n') # unicode is not evaluated in raw strings

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

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

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

### unicode.py

```
26°
26°
26\u00B0\n
we spent €1.23M for an original C  zanne
Romance in F   Major

             !
    !
             !
      !
               

unsorted: ['  ', '  ']
sorted: ['  ', '  ']
```

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 <code>name</code>
<code>\r</code>	Carriage Return
<code>\t</code>	TAB
<code>\uxxxx</code>	16-bit Unicode code point (must be padded with zeroes)
<code>\Uxxxxxxxx</code>	32-bit Unicode code point (must be padded with zeroes)
<code>\ooo</code>	Char with octal ASCII value <code>ooo</code>
<code>\xhh</code>	Character with hex ASCII value <code>hh</code>

# String operators and methods

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

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

## String operators

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

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

## String methods

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

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

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

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

String methods may be called on literal strings as well

## Builtin functions

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



## Example

### string\_methods.py

```
file_path = "Projects/alpha/src/utils/pdfstuff.py"

print(f"{file_path =}")
print(f"{len(file_path) =}")

print(f"{file_path.upper() =}")
print(f"{file_path.count('/') =}")
print(f"{file_path.count('p') =}")
print(f"{file_path.lower().count('p') =}")
print(f"{file_path.startswith('Projects') =}")
print(f"{file_path.endswith('.py') =}")
print(f"{file_path.removesuffix('.py') =}")
print(f"'alpha' in file_path =")
print(f"'beta' in file_path =")
print(f"{file_path.find('alpha') =}")
print(f"{file_path.find('beta') =}")
print(f"{file_path.replace('alpha', 'beta') =}")
print(f"{file_path.split('/') =}")
parts = file_path.split('/')
print(f"{':'.join(parts) =}")
print()

title = "    Why I love Python    "
print(f"title = [{title}]")
print(f"title.strip() = [{title.strip()}]")
print(f"title.lstrip() = [{title.lstrip()}]")
print(f"title.rstrip() = [{title.rstrip()}]")
```

***string\_methods.py***

```
file_path = 'Projects/alpha/src/utils/pdfstuff.py'
len(file_path) = 36
file_path.upper() = 'PROJECTS/ALPHA/SRC/UTILS/PDFSTUFF.PY'
file_path.count('/') = 4
file_path.count('p') = 3
file_path.lower().count('p') = 4
file_path.startswith('Projects') = True
file_path.endswith('.py') = True
file_path.removesuffix('.py') = 'Projects/alpha/src/utils/pdfstuff'
'alpha' in file_path = True
'beta' in file_path = False
file_path.find('alpha') = 9
file_path.find('beta') = -1
file_path.replace('alpha', 'beta') = 'Projects/beta/src/utils/pdfstuff.py'
file_path.split('/') = ['Projects', 'alpha', 'src', 'utils', 'pdfstuff.py']
':'.join(parts) = 'Projects:alpha:src:utils:pdfstuff.py'

title = [ Why I love Python ]
title.strip() = [Why I love Python]
title.lstrip() = [Why I love Python ]
title.rstrip() = [ Why I love Python]
```

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

Method	Description
<code>S.isprintable()</code>	Return True if all characters in S are considered printable in repr() or S is empty, False otherwise.
<code>S.isspace()</code>	Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.
<code>S.istitle()</code>	Return True if S is a title-cased string and there is at least one character in S, i.e. upper- and title-case characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise.
<code>S.isupper()</code>	Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.
<code>S.join(iterable)</code>	Return a string which is the concatenation of the strings in the iterable. The separator between elements is the string from which join() is called
<code>S.ljust(width[, fillchar])</code>	Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space).
<code>S.lower()</code>	Return a copy of the string S converted to lowercase.
<code>S.lstrip([chars])</code>	Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead.
<code>S.partition(sep)</code>	Search for the separator sep in S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
<code>S.removeprefix(prefix)</code>	Return copy of S with prefix removed from beginning. If the prefix is not found, return S.
<code>S.removesuffix(suffix)</code>	Return copy of S with suffix removed from end. If the suffix is not found, return S.
<code>S.replace(old, new[, count])</code>	Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.
<code>S.rfind(sub[, start[, end]])</code>	Return the highest index in S where substring sub is found, such that sub is contained within S[start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.
<code>S.rindex(sub[, start[, end]])</code>	Like rfind() but raise ValueError when the substring is not found.
<code>S.rjust(width[, fillchar])</code>	Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space).
<code>S.rpartition(sep)</code>	Search for the separator sep in S, starting at the end of S, and return the part before it, the separator itself, and the part after it. If the separator is not found, return two empty strings and the separator.

Method	Description
<code>S.rsplit(sep=None, maxsplit=-1)</code>	Return a list of the words in S, using sep as the delimiter string, starting at the end of the string and working to the front. If maxsplit is given, at most maxsplit splits are done. If sep is not specified, any whitespace string is a separator.
<code>S.rstrip([chars])</code>	Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead.
<code>S.split(sep=None, maxsplit=1)</code>	Return a list of the words in S, using sep as the delimiter string. If maxsplit is given, at most maxsplit splits are done. If sep is not specified or is None, any whitespace string is a separator and empty strings are removed from the result.
<code>S.splitlines([keepends])</code>	Return a list of the lines in S, breaking at line boundaries. Line breaks are not included in the resulting list unless keepends is given and true.
<code>S.startswith(prefix[, start[, end]])</code>	Return True if S starts with the specified prefix, False otherwise. With optional start, test S beginning at that position. With optional end, stop comparing S at that position. prefix can also be a tuple of strings to try.
<code>S.strip([chars])</code>	Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead.
<code>S.swapcase()</code>	Return a copy of S with uppercase characters converted to lowercase and vice versa.
<code>S.title()</code>	Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case.
<code>S.translate(table)</code>	Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted.
<code>S.upper()</code>	Return a copy of S converted to uppercase.
<code>S.zfill(width)</code>	Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated.

# Numeric literals

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

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

## Integers

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

## Floats

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



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



If a literal number starts with `0`, the next character *must* be `x`, `b`, or `o`.

## Example

### numeric.py

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

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

### numeric.py

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

# Math operators and expressions

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

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

Most of the operators should look familiar.

## Division /

Division, also called "true division", (/) always returns a float result.

## Floored Division //

Using the floored division operator //, the result is always rounded down to the nearest whole number. If both operands are ints, the result is an integer; otherwise, the result is a whole number float.

## Assignment-with-operation +=, \*=, -=, etc.

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

VARIABLE OP=VALUE      e.g. `x += 1`

is equivalent to

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

## Exponentiation \*\*

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

## Order of operations

**Please Excuse My Dear Aunt Sally!**

The order of operations is parentheses, exponents, multiplication or division, addition or subtraction, and left-to-right within the same type.

It's best practice to enforce the order of operations using parentheses for readability.



## Example

### math\_operators.py

```
a = 23
b = 7

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

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

print(a ** b) # exponentiation

print(a % b) # modulus (remainder)

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

print(f"{1 + 2 * 3 / 4 =}")
print(f"{(1 + 2) * (3 / 4) =}")
print(f"{(1 + (2 * 3)) / 4 =}")
print(f"{1 + ((2 * 3) / 4) =}") # same as example without parens
```

### math\_operators.py

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

1 + 2 * 3 / 4 = 2.5
(1 + 2) * (3 / 4) = 2.25
(1 + (2 * 3)) / 4 = 1.75
1 + ((2 * 3) / 4) = 2.5
```



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 type constructors

Python is dynamically typed; if you assign a number to a variable, it will raise an error if you use it with a string operator or function; likewise, if you assign a string, you can't use it with numeric operators. There is no automatic conversion between types.

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

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



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

Table 7. Builtin type constructors

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

## Example

### **converting\_types.py**

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

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

### **converting\_types.py**

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

## Chapter 2 Exercises

### Exercise 2-1 (string\_fun.py)

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

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

# Chapter 3: Basic input and output

## Objectives

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

## Writing to the screen

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

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

Two special named arguments can modify the default behavior. The `sep` argument specifies what is output between items, and `end` specifies what is written after all the arguments.

## Example

### **print\_examples.py**

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

print(city, temperature, hit_count, average)
print()

print(city, end=' ') # Print space instead of newline at the end
print(temperature)
print()

# Item separator is comma + space
print(city, temperature, hit_count, average, sep=', ')
print()

# Item separator is empty string
print(city, temperature, hit_count, average, sep='')
print()
```

### **print\_examples.py**

```
Orlando 85 5 3.4563892382

Orlando 85

Orlando,85,5,3.4563892382

Orlando8553.4563892382
```



## Format strings

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

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

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

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

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

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

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

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



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



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

## Example

### **f\_strings.py**

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

# variables inserted into string
print(f"It is {temperature}\u00B0 in {city}")
print()

# :03d means format (decimal) integer in 3 characters,
#     left-padded with zeros
# :.2f means round a float to 2 decimal points
print(f"hit count is {hit_count:03d} average is {average:.2f}")
print()

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

### **f\_strings.py**

```
It is 85° in Orlando

hit count is 005 average is 3.46

2 + 2 is 4
```

## Example

### f\_strings\_fun.py

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

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

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

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

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

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

*f\_strings\_fun.py*

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

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

result: $38,293,892

amount: 0.325482039
amount: 0.33
amount: 32.55%

length of 'name': 5
```

## Using the `.format()` method

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

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

Formatting codes still go after a colon.

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



`.format()` is useful for reusing the same format with different values.

### Example

#### `string_formatting.py`

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

# variables inserted into string
print("It is {}° in {}".format(temperature, city))
print()

# :03d means format (decimal) integer in 3 characters,
#     left-padded with zeros
# :.2f means round a float to 2 decimal points
print("hit count is {:03d} average is {:.2f}".format(hit_count, average))
print()

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

#### `string_formatting.py`

It is 85° in Orlando

hit count is 005 average is 3.46

2 + 2 is 4

## Legacy String Formatting

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

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

```
%f%lagW.Ptype
```

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

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

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

## Example

### string\_formatting\_legacy.py

```
city = 'Orlando'
temperature = 85
hit_count = 5
average = 3.4563892382

# variables inserted into string
print("It is %d\u00B0 in %s" % (temperature, city))
print()

# :03d means format (decimal) integer in 3 characters,
#     left-padded with zeros
# :.2f means round a float to 2 decimal points
print(f"hit count is %03d average is %.2f" % (hit_count, average))
print()

# any expression is OK
print(f"2 + 2 is %d" % (2 + 2))
```

### string\_formatting\_legacy.py

```
It is 85° in Orlando

hit count is 005 average is 3.46

2 + 2 is 4
```



Table 8. Legacy formatting types

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

Table 9. Legacy formatting flags

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

## str() vs repr()

- `str()` info for humans
- `repr()` how to create object

There are two ways to convert a Python object into a string. The first way is `str()`, which `print()` uses on each object passed to it. The string representation of an object should provide information about the object to a human looking at the output.

The second way is using `repr()`, which returns a string representing how to recreate the object (the "raw" view).

This is the default for many objects other than strings, such as lists, dictionaries, tuples, and sets.

### Example

#### `str_vs_repr.py`

```
from datetime import date

today = date.today()

print(today)    # uses str(today)
print()
print(repr(today)) # uses repr(today)
print()
print(f"{today =}") # also uses repr(today)
```

#### `str_vs_repr.py`

```
2024-09-27

datetime.date(2024, 9, 27)

today = datetime.date(2024, 9, 27)
```

## f-string shortcut

- Saves typing object name twice
- Put `=` after object name

A convenient shortcut is to put an equals sign after the object in an f-string field. This outputs the name of the object, then an equals sign, then the value of the object. This saves having to type the object name twice, as in

```
print("x = ", x)
```

This shortcut always uses `repr()` to display the object as a "raw" view.

You can still add formatting information to the field as usual.

## Example

### **f\_string\_shortcut.py**

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

print(f"{city = }")    # default is raw view
print(f"{city = !s}")  # forces normal str view
print(f"{temp = }")
print(f"{count = }")
print(f"{count = :05d}") # add formatting
print(f"{avg = :.2f}")   # add formatting
print(f"{flag = }")
```

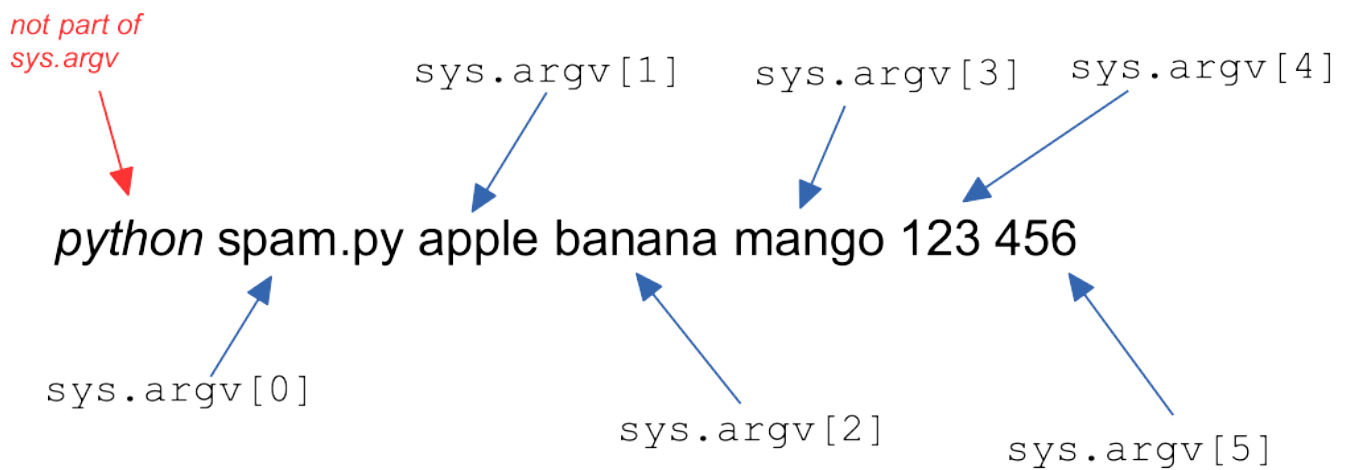
### **f\_string\_shortcut.py**

```
city = 'Orlando'
city = Orlando
temp = 85
count = 5
count = 00005
avg = 3.46
flag = True
```

## Command line arguments

- `argv` list in `sys` module
- `sys` must be imported
- Element 0 is script name itself

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



## Example

### **sys\_argv.py**

```
import sys

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

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

### **sys\_argv.py wombat**

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

animal: wombat
```



If you use an index for a non-existent argument, 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 argument is a prompt string that will be displayed on the console. `input()` returns the text that was entered as a string.

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



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

## Example

### `keyboard_input.py`

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

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

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

### `keyboard_input.py`

```
What is your name: Sir Lancelot
What is your quest? the Grail
Sir Lancelot seeks the Grail
Enter number: 5
2 times 5.0 is 10.0
```

## Chapter 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 your IDE)

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

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

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

### Exercise 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 your IDE)

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

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

Just for fun

Table 10. Temperature Scales

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



# 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

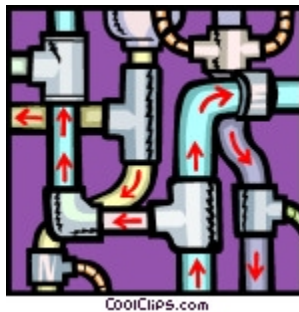
- Controls order of execution
- Conditionals and loops
- Uses Boolean logic

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`.



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



## if and elif

- Basic conditional statement is **if**
- **else** for alternatives
- **elif** provides nested if-else

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

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

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

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.

```
if EXPR:
    statement
    statement
    ...
elif EXPR:
    statement
    statement
    ...
else EXPR:
    statement
    statement
    ...
```

# White space

- Blocks defined by indenting
- No braces or special keywords
- Enforces what good programmers do anyway
- Be consistent
  - Suggested indent: 4 spaces

One of the first things that most programmers learn about Python is that whitespace is significant. This enforces what good programming practice by making Python code more readable. Nested blocks are always properly indented, or they won't run.

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

Blocks may be nested, as in any language. The nested block has more indentation. A block ends when the interpreter sees a line with less indentation than the current block.

The standard indentation for Python scripts is 4 spaces. There is very seldom any reason to vary this.



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

## Example

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

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

## Determining Boolean values

- `False`, `None`, numeric zero, and empty containers are false
- Everything else is true

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

The builtin objects `True` and `False` are objects of type `bool`, predefined to have values of 1 and 0, respectively. These are convenient for returning a Boolean value from a function, or for populating lists with true/false values.



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

Table 11. Boolean values

For any object X	
If X is	Boolean value is
Numeric, and equal to 0	<code>False</code>
Numeric, and NOT equal to 0	<code>True</code>
A collection, and <code>len(X)</code> is 0	<code>False</code>
A collection, and <code>len(X)</code> is > 0	<code>True</code>
<code>None</code>	<code>False</code>
<code>False</code>	<code>False</code>
<code>True</code>	<code>True</code>
<i>anything else</i>	<code>True</code>

# The Conditional Expression

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

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

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

This is a shortcut for

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

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

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

## Example

```
print(long_message if DEBUGGING else short_message)

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

file_mode = 'a' if APPEND_MODE else 'w'
```

# Relational Operators

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

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

==   !=   <   >   >=   <=



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



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

## Example

### if\_else.py

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

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

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

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

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

# 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.

## Example

### boolean\_ops.py

```
alpha = 15
beta = 10
gamma = 0
delta = 0

print(f"{alpha = } {bool(alpha) = }")
print(f"{beta = } {bool(beta) = }")
print(f"{gamma = } {bool(gamma) = }")
print(f"{delta = } {bool(delta) = }")
print()

print(f"{alpha and beta = }")
print(f"{bool(alpha and beta) = }")
print()

print(f"{beta and alpha = }")
print(f"{bool(beta and alpha) = }")
print()

print(f"{alpha and gamma = }")
print(f"{bool(alpha and gamma) = }")
print()

print(f"{alpha or gamma = }")
print(f"{bool(alpha or gamma) = }")
print()

print(f"{gamma or beta = }")
print(f"{bool(gamma or beta) = }")
print()

print(f"{gamma or delta = }")
print(f"{bool(gamma or delta) = }")
```

***boolean\_ops.py***

```
alpha = 15 bool(alpha) = True
beta = 10 bool(beta) = True
gamma = 0 bool(gamma) = False
delta = 0 bool(delta) = False
```

```
alpha and beta = 10
bool(alpha and beta) = True
```

```
beta and alpha = 15
bool(beta and alpha) = True
```

```
alpha and gamma = 0
bool(alpha and gamma) = False
```

```
alpha or gamma = 15
bool(alpha or gamma) = True
```

```
gamma or beta = 10
bool(gamma or beta) = True
```

```
gamma or delta = 0
bool(gamma or delta) = False
```

Table 12. 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. Unlike many languages, reading a file in Python uses a `for` loop.

# Loop control

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

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

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

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

## Example

### `while_loop_examples.py`

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

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

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

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

Welcome to ticket sales

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

sending 4 ticket(s)

Enter quantity to purchase (or q to quit):

Enter quantity to purchase (or q to quit): 2

sending 2 ticket(s)

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

goodbye!

## Chapter 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".



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

### Exercise 4-2

#### Part A (guess.py)

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

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



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

#### Part B (guessx.py)

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



## Pseudocode for **guess.py**

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

# Chapter 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, known as "containers". Some of these are array types. These hold values in a sequence, such that they can be retrieved by a numerical index. Such a type is also sometimes called an "ordered sequence".

A **str** is an array of characters. A **bytes** object is array of bytes. A **list** is an array of objects A **tuple** is an array of objects

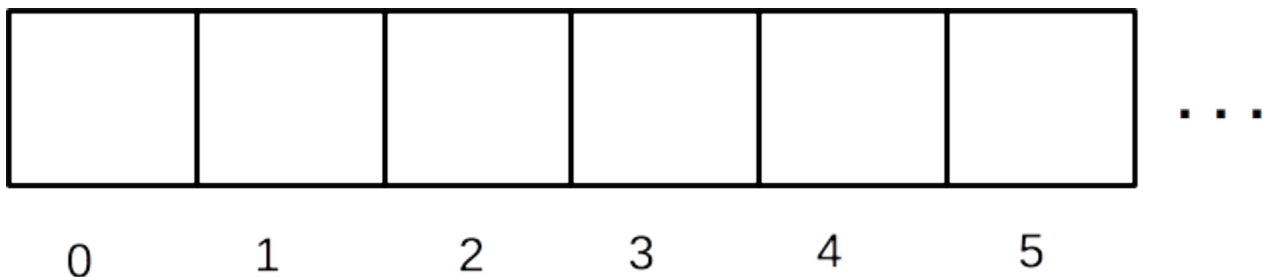
All array types may be indexed in the same way, retrieving a single item or a slice (multiple values) of the sequence.

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

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



Containers are sometimes called "collections"



## Example

### typical\_arrays.py

```
fruits = ['apple', 'cherry', 'orange', 'kiwi', 'banana', 'pear', 'fig'] # list
full_name = "Guido van Rossum" # str
place = "Durham", "NC" # tuple
data = b'wombat' # bytes

print(f"{fruits = }")
print(f"fruits[3]: {fruits[3]} len(fruits): {len(fruits)}\n")

print(f"{full_name = }")
print(f"full_name[2]: {full_name[2]} len(full_name): {len(full_name)}\n")

print(f"{place = }")
print(f"place[1]: {place[1]} len(place): {len(place)}\n")

print(f"{data = }")
print(f"data[4]: {data[4]} len(data): {len(data)}\n")
```

### typical\_arrays.py

```
fruits = ['apple', 'cherry', 'orange', 'kiwi', 'banana', 'pear', 'fig']
fruits[3]: kiwi len(fruits): 7

full_name = 'Guido van Rossum'
full_name[2]: i len(full_name): 16

place = ('Durham', 'NC')
place[1]: NC len(place): 2

data = b'wombat'
data[4]: 97 len(data): 6
```

# Lists

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

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

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

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

Table 13. List Methods and Keywords (note L represents a list)

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

## Example

### using\_lists.py

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

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

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

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

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

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

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

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

*using\_lists.py*

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

```
cities: ['Portland', 'Pittsburgh', 'Peoria', 'Miami', 'Montgomery']
```

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

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

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

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

```
city: Des Moines
```

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

```
city: Miami
```

```
cities: ['Boston', 'Portland', 'Pittsburgh', 'Montgomery', 'Detroit']
```



# Indexing and slicing

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

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

To get a single element, specify the index (0-based) of the element in square brackets. A negative index counts from the end — `x[-1]` is the last element of `x`.

```
foo = [ "apple", "banana", "cherry", "date", "elderberry",
        "fig", "grape" ]
```

`foo[1]` the 2nd element of list foo -- banana

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

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

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

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

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

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

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

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

The general syntax for a slice is

```
s[start-at:stop-before:count-by]
```

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

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

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



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

## Example

### indexing\_and\_slicing.py

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

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

phrase = "She turned me into a newt"

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

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

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

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

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

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

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

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

## Iterating through a sequence

- use a **for** loop
- works with lists, tuples, strings, or any other iterable
- Syntax

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

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

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

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



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

## Example

### iterating\_over\_arrays.py

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

for p in my_list: # Iterate over elements of list
    print(p)
print()

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

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

### iterating\_over\_arrays.py

```
Idle
Cleese
Chapman
Gilliam
Palin
Jones

Roger
Old Woman
Prince Herbert
Brother Maynard

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

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

```
Idle
Cleese
Chapman
Gilliam
Palin
Jones

Roger
Old Woman
Prince Herbert
Brother Maynard

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

# Tuples

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

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

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

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

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

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

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

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



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

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



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



## Example

### creating\_tuples.py

```
birth_date = 1901, 5, 5

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

latlon = 35.99, -72.390

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

### creating\_tuples.py

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



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

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

## Iterable Unpacking

- Copy elements to variables
- Works with any iterable
- More readable than numeric indexing

If you have a tuple like this:

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

You can access the elements with

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

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

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

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

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

## Example

### iterable\_unpacking.py

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

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

print(x, y, z)
print()

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

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

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

***iterable\_unpacking.py***

```
a b c
```

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

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

## Nested sequences

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

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

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

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

## Example

### nested\_sequences.py

```
people = [  
    ('Melinda', 'Gates', 'Gates Foundation'),  
    ('Steve', 'Jobs', 'Apple'),  
    ('Larry', 'Wall', 'Perl'),  
    ('Paul', 'Allen', 'Microsoft'),  
    ('Larry', 'Ellison', 'Oracle'),  
    ('Bill', 'Gates', 'Microsoft'),  
    ('Mark', 'Zuckerberg', 'Facebook'),  
    ('Sergey', 'Brin', 'Google'),  
    ('Larry', 'Page', 'Google'),  
    ('Linus', 'Torvalds', 'Linux'),  
]  
  
for person in people: # person is a tuple  
    print(person[0], person[1])  
print('-' * 60)  
  
for person in people:  
    first_name, last_name, product = person # unpack person into variables  
    print(first_name, last_name)  
print('-' * 60)  
  
for first_name, last_name, product in people: # if there is more than one variable in a  
for loop, each element is unpacked  
    print(first_name, last_name)  
print('-' * 60)
```

## Operators and keywords for sequences

- Operators `+` (plus) `*` (splat)
- Keywords `del` `in` `not` `in`

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

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

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

`+` adds one sequence to another

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

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

## Example

### *sequence\_operators.py*

```

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

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

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

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

del colors[4] # remove brown

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

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

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

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

print("sum of lists:", sum_of_lists)

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

print("product of lists:", product)

```

### *sequence\_operators.py*

```

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

```



# Functions for all sequences

- Many builtin functions expect a sequence
- Syntax

```
n = len(s)
n = min(s)
n = max(s)
n = sum(s)
s2 = sorted(s)
```

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

## len()

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

## min(), max(), sorted()

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

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



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

## sum()

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

## Example

### sequence\_functions.py

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

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

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

### sequence\_functions.py

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

sorted: black blue brown green red yellow
```

# Iterators

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

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

## `enumerate()`

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

When you iterate through `[x, y, z]` with `enumerate()`, you get a sequence of tuples: `[(0,x),(1,y),(2,z)]`



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

## `reversed()`

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

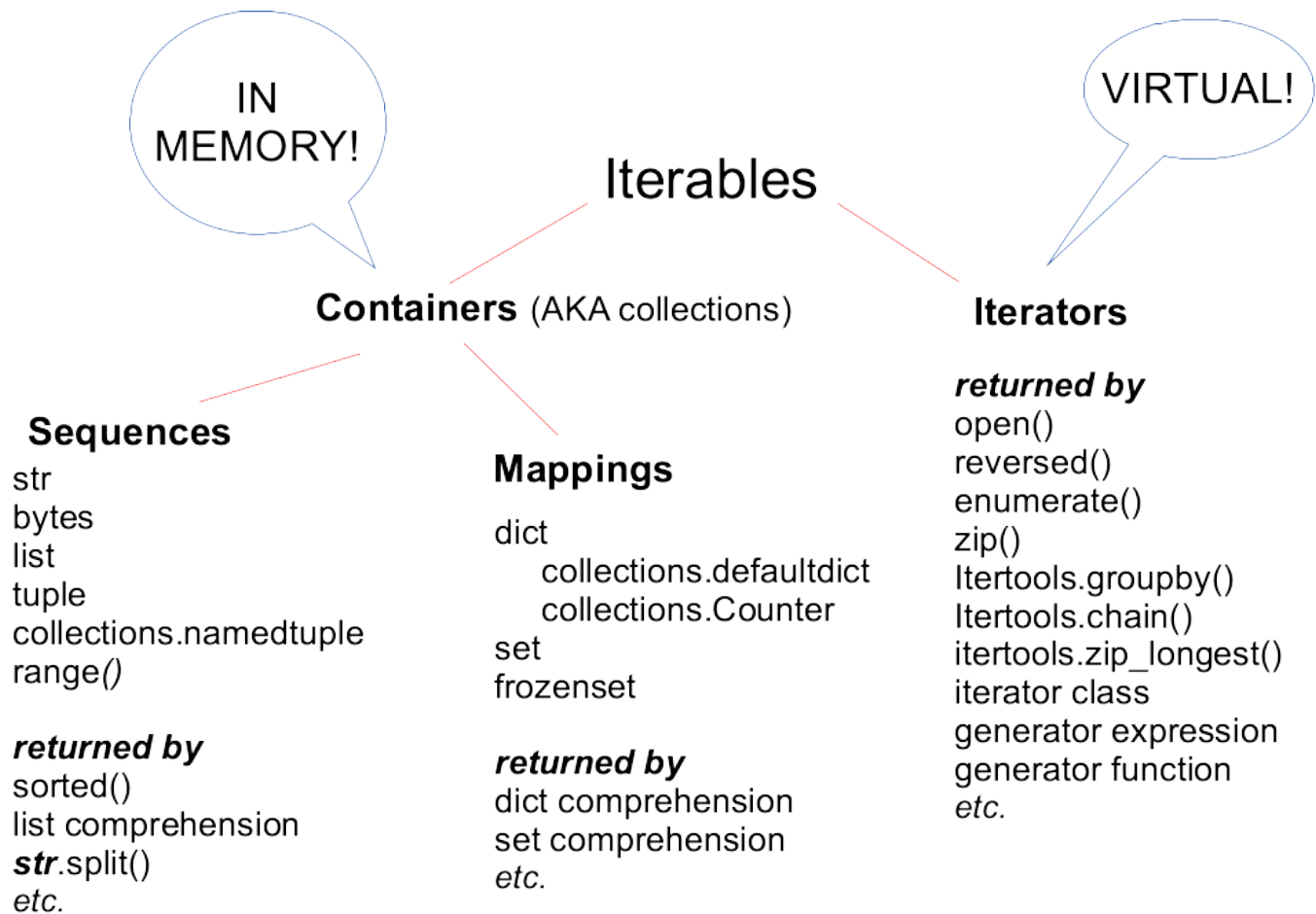
## `zip()`

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

## `range()`

`range()` returns a **range object**, that provides a sequence of integers. The parameters to `range()` are similar to the parameters for slicing (*start, stop, step*).

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



## Examples

### enumerate.py

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

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

colors_enum = enumerate(colors)
print(f"{colors_enum = }")

for i, color in colors_enum: # enumerate() returns iterable of (index, value) tuples
    print(i, color)

print()

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

### enumerate.py

```
colors_enum = <enumerate object at 0x109c4ab10>
0 red
1 blue
2 green
3 yellow
4 brown
5 black

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

**iterators.py**

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

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

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

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

**iterators.py**

```
man bites dog

full_names: <zip object at 0x105651d80>

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

**using\_ranges.py**

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

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

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

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

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

**using\_ranges.py**

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

## List comprehensions

- Shortcut for a **for** loop
- Optional **if** clause
- Returns list
- Syntax

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

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

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

can be rewritten as

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

A conditional **if** may be added:

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

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



## Example

### list\_comprehensions.py

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

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

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

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

doubles = [v * 2 for v in values]    # Any kind of data is OK

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

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

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

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

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

deck = [(rank, suit) for suit in suits for rank in ranks]    # More than one for is OK

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

***list\_comprehensions.py***

```
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
```

...

# Generator Expressions

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

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

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

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

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

## Example

### generator\_expressions.py

```

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

ufruits = (fruit.upper() for fruit in fruits) # These are all exactly like the list
comprehension example, but return generators rather than lists
afruits = (fruit.title() for fruit in fruits if fruit.startswith('a'))

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

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

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

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

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

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

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

```

***generator\_expressions.py***

```
ufruits: WATERMELON APPLE MANGO KIWI APRICOT LEMON GUAVA
```

```
afruits: Apple Apricot
```

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

```
2 42 18 92
```

```
>gronk< >pulaba< >floog<
```

```
1 1 1
```

```
2 4 8
```

```
3 9 27
```

```
4 16 64
```

```
5 25 125
```

```
6 36 216
```

```
7 49 343
```

```
8 64 512
```

```
9 81 729
```

```
10 100 1000
```

## Chapter 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.

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

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

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



For the next two exercises, 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 5-3 (sequences.py)

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

### Exercise 5-4 (sequences.py)

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

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

## Exercise 5-5 (sieve.py)

### FOR ADVANCED STUDENTS

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

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

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

Use two nested loops.

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

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

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



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



In this exercise, the *value* of the element is either **True** or **False** — the *index* is the number to 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
```

Follow the first link below for an animated graphic describing the sieve algorithm.



[Sieve of Eratosthenes](#)

[Eratosthenes bio](#)



# 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.

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



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

# Opening a text file

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

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

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

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

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

## Examples

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



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

Table 14. File Modes

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

# Reading a text file

## Read file line by line

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

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

## Read entire file into one string

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

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

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

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

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

## Read entire file into list of lines

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

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

## Read entire file into list of lines (without newlines)

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

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

## Read next line

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

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

## Example

### reading\_files.py

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

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

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

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

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

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

*reading\_files.py*

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

-----

NORMAL:

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

=====

RAW:

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

-----

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

-----

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



## Processing files from the command line

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

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

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



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

### Example

#### `looping_over_files.py`

```
import sys

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

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

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

## Example

### looping\_over\_files\_using\_first\_arg.py

```
import sys

first_arg = sys.argv.pop(1)

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

***looping\_over\_files\_using\_first\_arg.py spam ../DATA/alice.txt ../DATA/mary.txt***

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

## Writing to a text file

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

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

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

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

### Example

#### `write_file.py`

```
states = [  
    'Virginia',  
    'North Carolina',  
    'Washington',  
    'New York',  
    'Florida',  
    'Ohio',  
]  
  
with open("states.txt", "w") as states_out: # "w" opens for writing, "a" for append  
    for state in states:  
        states_out.write(state + "\n") # write() does not automatically add newline
```

#### `states.txt`

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



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

## Modifying text files

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

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

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

### Example

#### **modify\_text\_file.py**

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

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

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

Table 15. File Object 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 <code>n</code> 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 newline characters.

## Chapter 6 Exercises

### Exercise 6-1 (c2f\_file.py)

Write a program to read Celsius temperatures from the file `ctemps.txt` into a list. Loop over the list of Celsius temperatures and convert each one to Fahrenheit. Print out both temperatures.

### Exercise 6-2 (line\_no.py)

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



Use `enumerate()`.

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

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

Test with other files, as desired

### Exercise 6-3 (alt\_lines.py)

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

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

### Exercise 6-4 (count\_alice.py, count\_words.py)

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



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

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

### Exercise 6-5 (icount\_words.py)

*FOR ADVANCED STUDENTS*

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

# Chapter 7: Dictionaries and Sets

## Objectives

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



## About dictionaries

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

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

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

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

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

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

Dictionaries are also handy for counting. The keys are the things being counted, and the values are the counts for each thing.

For instance, a dictionary might

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



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



# Creating dictionaries

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

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

## The `dict()` constructor

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

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

## The `{}` literal dictionary constructor

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

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

## Adding elements

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

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

## Example

### creating\_dicts.py

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

airports = {'IAD': 'Dulles', 'SEA': 'Seattle-Tacoma', # initialize dict with literal
            'RDU': 'Raleigh-Durham', 'LAX': 'Los Angeles'}

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

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

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

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

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

### creating\_dicts.py

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

Table 16. 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] =</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 17. 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 a (key, value) pair. Pairs are removed in LIFO order.

## Getting dictionary values

- `d[key]` *missing key raises `KeyError`*
- `d.get(key, default-value)`
- `d.setdefault(key, default-value)`

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

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

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

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

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

## Example

### getting\_dict\_values.py

```
d1 = dict()

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

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

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

state_caps = dict(pairs)

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

airports['SLC'] = 'Salt Lake City'
airports['LAX'] = 'Lost Angels'
print(airports['SLC']) # print value where key is 'SLC'

code = 'PSP'

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

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

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

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

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

## Iterating over a dictionary

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

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

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



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

## Example

### iterating\_over\_dicts.py

```
airports = {'IAD': 'Dulles', 'SEA': 'Seattle-Tacoma', 'YCC': 'Calgary',  
            'RDU': 'Raleigh-Durham', 'LAX': 'Los Angeles'}  
  
for abbr, airport in airports.items(): # items() returns an iterable of key:value pairs  
    print(abbr, airport)  
  
print('-' * 60)  
  
for abbr, airport in sorted(airports.items()): # iterate, sorted by key  
    print(abbr, airport)
```

### iterating\_over\_dicts.py

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



## Reading file data into a dictionary

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

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



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

### Example

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

```
from pprint import pprint

knight_info = {} # create empty dict

with open("../DATA/knights.txt") as knights_in:
    for line in knights_in:
        name, title, color, quest, comment = line.rstrip('\n\r').split(":")
        knight_info[name] = title, color, quest, comment # create new dict element with
        name as key and a tuple of the other fields as the value

pprint(knight_info)
print()

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

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

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

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

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

Not Sure



See also **`read_into_dict_of_dicts.py`** and **`read_into_dict_of_named_tuples.py`** in the EXAMPLES folder.

# Counting with dictionaries

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

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

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



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

## Example

### `count_with_dict.py`

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

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

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

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

```
spam 34
Lucky Charms 4
eggs 3
oatmeal 1
sausage 4
upma 3
poha 4
ackee and saltfish 4
bacon 6
pancakes 2
idli 7
dosas 4
waffles 2
crumpets 1
```

**NOTE:**

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

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

See [count\\_with\\_dict\\_terse.py](#) for an example.

## About sets

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

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

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

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

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

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

## Creating Sets

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

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

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



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

## Working with sets

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

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

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

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

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

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

## Example

### set\_examples.py

```
set1data = ['red', 'blue', 'green', 'purple', 'green'] # create literal set
set2data = ['green', 'blue', 'yellow', 'orange']

set1 = set(set1data)
set2 = set(set2data)

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

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

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

print(f"{food =}")
print()

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



**set\_examples.py**

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

food = ['spam', 'spam', 'spam', 'Lucky Charms', 'spam', 'eggs', 'oatmeal', 'spam',
'spam', 'sausage', 'eggs', 'spam', 'Lucky Charms', 'upma', 'Lucky Charms', 'poha',
'spam', 'ackee and saltfish', 'bacon', 'pancakes', 'spam', 'sausage', 'spam', 'upma',
'spam', 'spam', 'spam', 'idli', 'idli', 'idli', 'dosas', 'waffles', 'dosas', 'dosas',
'spam', 'poha', 'bacon', 'bacon', 'spam', 'spam', 'ackee and saltfish', 'idli', 'ackee
and saltfish', 'spam', 'sausage', 'crumpets', 'spam', 'sausage', 'eggs', 'spam',
'waffles', 'spam', 'Lucky Charms', 'idli', 'spam', 'poha', 'spam', 'bacon', 'bacon',
'spam', 'spam', 'dosas', 'idli', 'spam', 'spam', 'bacon', 'spam', 'poha', 'spam', 'ackee
and saltfish', 'pancakes', 'spam', 'spam', 'spam', 'upma', 'spam', 'spam', 'spam',
'idli']

unique_food = {'poha', 'idli', 'dosas', 'waffles', 'Lucky Charms', 'pancakes', 'upma',
'eggs', 'spam', 'bacon', 'crumpets', 'sausage', 'ackee and saltfish', 'oatmeal'}
```



See also the file [set\\_examples\\_customers.py](#) in EXAMPLES.

Table 18. Set functions and methods

Function/Operator	Description (S=set, S2=other set, m=member)
<code>m in S</code>	True if S contains m
<code>m not in S</code>	True if S does not contain m
<code>len(s)</code>	the number of items in S
<code>S.add(m)</code>	Add m to S (if S already contains m do nothing)
<code>S.clear()</code>	remove all members from S
<code>S.copy()</code>	shallow copy of S
<code>S - S2</code> <code>S.difference(S2)</code>	Return the set of all elements in S that are not in S2
<code>S.difference_update(S2)</code>	Remove all members of S2 from S
<code>S.discard(m)</code>	Remove member m from S if it is a member. If m is not a member, do nothing.
<code>S &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 is S is a subset of S2
<code>S.issuperset(S2)</code>	Return True is S2 is a subset of S
<code>S.pop()</code>	Remove and return an arbitrary set element. Raises KeyError if the set is empty.
<code>S.remove(m)</code>	Remove member m from a set; it must be a member.
<code>S ^ S2</code> <code>S.symmetric_difference(S2)</code>	Return all members in S or S2 but not both.
<code>S.symmetric_difference_update(S2)</code>	Update a set with the symmetric difference of itself and another.
<code>S   S2</code> <code>S.union(S2)</code>	Return all members that are in S or S2
<code>S.update(S2)</code>	Update a set with the union of itself and S2

# Chapter 7 Exercises

## Exercise 7-1 (scores.py)

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



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

Table 19. Grading Scale

Score	Letter grade
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, modules, and packages

## Objectives

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

# Functions

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

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

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

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



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

## Example

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

```
# define function
def say_hello():
    print("Hello, world")

say_hello() # Call function
```

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

```
Hello, world
```

## Function parameters

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

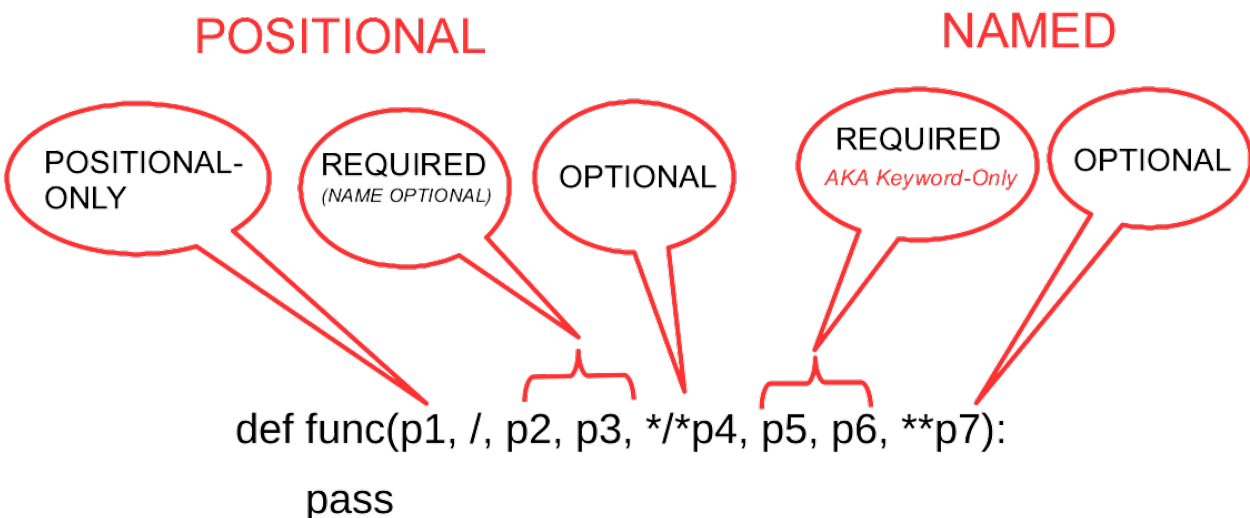
Functions can accept both positional and named parameters. Furthermore, parameters can be required or optional. They must be specified in the order presented here.

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

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

### Example

```
def some_function(pos_only1, pos_only2, /, pos_or_kw1, pos_or_kw2, *pos_varargs,  
                  kw_only1, kw_only2, **kw_varargs):  
    pass
```





## Example

### function\_parameters.py

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

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

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

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

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

fun_three()
fun_three(10)
print()

def fun_four(n, *opt): # one fixed, plus optional parameters
    print("fun_four():")
    print("n is ", n)
    print("opt is", opt)
    print('-' * 20)

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

def fun_five(*, spam=0, eggs=0): # keyword-only parameters
    print("fun_five():")
    print("spam is:", spam)
    print("eggs is:", eggs)
```

```
print()
```

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

```
def fun_six(**named_args): # keyword (named) parameters
    print("fun_six():")
    for name in named_args:
        print(f"{name} ==> {named_args[name]}")
```

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

***function\_parameters.py***

```
fun_one(): Hello, world

fun_two(5) is 25

spam spam spam
spam spam spam spam spam spam spam spam spam

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

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

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

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

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

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

## Default parameters

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

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

### Example

#### default\_parameters.py

```
def spam(greeting, whom='world'): # 'world' is default value for positional parameter
    whom
    print(f"{greeting}, {whom}")

spam("Hello", "Mom") # parameter supplied; default not used
spam("Hello") # parameter not supplied; default is used
print()

def ham(*, file_name, file_format='txt'): # 'world' is default value for named parameter
    format
    print(f"Processing {file_name} as {file_format}")

ham(file_name='eggs') # parameter format not supplied; default is used
ham(file_name='toast', file_format='csv')
```

#### default\_parameters.py

```
Hello, Mom
Hello, world

Processing eggs as txt
Processing toast as csv
```

# Every type of function parameter

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

There are five types of parameters. They must be specified in the following order:

## Positional-only

Can only be passed by position; used for emulating builtins. Specified before a lone `/` in the parameter list.

## Required positional

"normal" parameters; can be passed by name *or* position; may have default values.

## Optional positional

allow any number of arguments; parameter is a tuple of all non-required arguments.

## Required named

AKA keyword-only; must be passed by name; may have default values.

## Optional named

Allow any number of named (keyword) arguments; parameter is a dictionary of the arguments and their values.



This may seem very complicated, but most of the time you will just use required positional parameters. The others are to fine-tune how the function is used by calling code, to make using it easier.

# Detailed Python Function parameter behavior (from PEP 570 and PEP 3102)

**Positional arguments:** `spam(10, 20, 30)`

**Named arguments:** `spam(a=10, b=20, c=30)`

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

Table 20. Function Parameter Examples

Parameter definitions	Example arguments	Parameter values
<code>f(p1, p2)</code>	<code>f(10, 20)</code>	p1: 10 p2: 20
<code>f(p1, *p2)</code>	<code>f(10, 20, 30)</code>	p1: 10 p2: (20, 30)
<code>f(p1, *p2, p3)</code>	<code>f(10, 20, 30, p3=40)</code>	p1: 10 p2: (20, 30) p3: 40
<code>f(p1, *p2, p3, **p4)</code>	<code>f(10, 20, 30, p3=40, x=50, y=60)</code>	p1: 10 p2: (20, 30) p3: 40 p4: {'x': 50, 'y': 60}
<code>f(p1, *p2, p3=1, **p4)</code>	<code>f(10, 20, 30, x=40, y=50)</code>	p1: 10 p2: (20, 30) p3: 1 p4: {'x': 40, 'y': 50}
<code>f(*, p1, p2)</code>	<code>f(p1=10, p2=20)</code>	p1: 10 p2: 20
<code>f(*, p1, p2=100)</code>	<code>f(p1=10)</code>	p1: 10 p2: 100
<code>f(*p1, p2, p3)</code>	<code>f(10, 20, 30, p2=40, p3=50)</code>	p1: (10, 20, 30) p2: 40 p3: 50
<code>f(**p1)</code>	<code>f(x=10, y=20)</code>	p1: {'x': 10, 'y': 20}
<code>f(*, p1, **p2)</code>	<code>f(p1=10, x=20, y=30)</code>	p1: 10 p2: {'x': 20, 'y': 30}
<code>f(*args, **kwargs)</code>	<code>f(10, 20, 30, x=40, y=50)</code>	args: (10, 20, 30) kwargs: {'x': 40, 'y': 50}



The last entry in the table shows "universal" arguments. They will accept any number of positional arguments, and any number of named arguments.

## Name resolution (AKA Scope)

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

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

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

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

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

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

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



## Example

### scope\_examples.py

```
x = 42 # global variable

def function_a():
    y = 5 # local variable to function_a(), or nonlocal to function_b()

    def function_b():
        z = 32 # local variable
        print("function_b(): z is", z) # local scope
        print("function_b(): y is", y) # nested (nonlocal) scope
        print("function_b(): x is", x) # global scope
        print("function_b(): type(x) is", type(x)) # builtin scope

    return function_b

f = function_a() # calling function_a, which returns function_b
f() # calling function_b
```

### scope\_examples.py

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

## The global statement

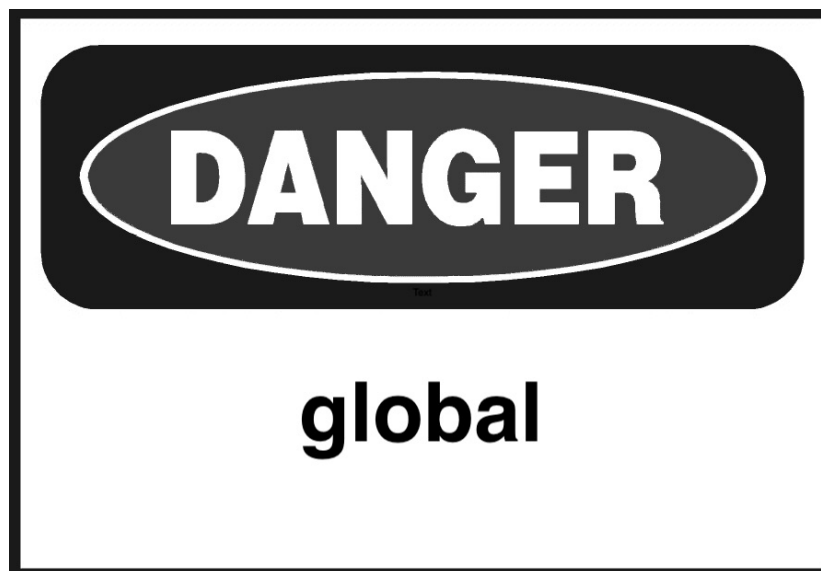
- global statement allows function to change globals
- nonlocal statement allows function to change nonlocals

The **global** keyword allows a function to modify a global variable. This is universally acknowledged as a BAD IDEA. Modifying global data can lead to all sorts of hard-to-diagnose bugs, because a function might change a global that affects some other part of the program.

The better approach is to pass data into functions as parameters and return data as needed. Mutable objects, such as lists, sets, and dictionaries can be modified in-place.



The **nonlocal** keyword can be used like **global** to make nonlocal variables in an outer function writable. *But don't.*



# Modules

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

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

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

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

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

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

Modules and packages should be documented with *docstrings*.

# Using import

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

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

## Variation 1

`import module`

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

## Variation 2

`from module import function, ...`

imports only the function(s) specified into the current namespace. Other functions are not available (even though they are loaded into memory).

## Variation 3

`from module import *`

loads the module, and imports all functions that do not start with an underscore into the current namespace. This should be used with caution, as it can pollute the current namespace and possibly overwrite builtin attributes or attributes from a different module.



The first time a module is loaded, the interpreter creates a version compiled for faster loading. This version has platform information embedded in the name, and has the extension `.pyc`. These `.pyc` files are put in a folder named `__pycache__`.

## Example

### geometry.py

```

"""
General geometry-related functions

Syntax:

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

import math    # load math.py

PI = math.pi

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

    :param diameter: Diameter of circle
    :return: Area of circle
    """
    radius = diameter / 2
    return PI * (radius ** 2)

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

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

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

    :param side: Length of one side
    :return: Area of square
    """
    return side ** 2

if __name__ == "__main__":

```

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

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

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

**use\_geometry\_1.py**

```
import geometry

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

**use\_geometry\_1.py**

```
50.26548245743669 120 62.410000000000004
```

**use\_geometry\_2.py**

```
import geometry as g

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

**use\_geometry\_2.py**

```
50.26548245743669 120 62.410000000000004
```

**use\_geometry\_3.py**

```
from geometry import circle_area, rectangle_area, square_area

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

**use\_geometry\_3.py**

```
50.26548245743669 120 62.410000000000004
```

**use\_geometry\_4.py**

```
from geometry import *

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

***use\_geometry\_4.py***

```
50.26548245743669 120 62.410000000000004
```

***use\_geometry\_5.py***

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

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

***use\_geometry\_5.py***

```
50.26548245743669 120 62.410000000000004
```



## How `import *` can be dangerous

- Imported names may overwrite existing names
- Be careful to read the documentation

Using `import *` to import all public names from a module has a bit of a risk. While generally harmless, there is the chance that you will unknowingly import a module that overwrites some previously-imported module.

To be 100% certain, always import the entire module, or else import names explicitly.

### Examples

#### **electrical.py**

```
default_amps = 10
default_voltage = 110
default_current = 'AC'

def amps():
    return default_amps

def voltage():
    return default_voltage

def current():
    return default_current
```

#### **navigation.py**

```
current_types = 'slow medium fast'.split()

def current():
    return current_types[0]
```

### why\_import\_star\_is\_bad.py

```
from electrical import current, voltage, amps # import current _explicitly_ from
electrical
from navigation import * # import current _implicitly_ from navigation

print(current()) # calls navigation.current(), not electrical.current()
print(voltage())
print(amps())
```

### why\_import\_star\_is\_bad.py

```
slow
110
10
```

### how\_to\_avoid\_import\_star.py

```
import electrical as e # import module electrical
import navigation as n # import module navigation

print(e.current()) # use the current() function in electrical
print(n.current()) # use the current() function in navigation
```

### how\_to\_avoid\_import\_star.py

```
AC
slow
```

## Module search path

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

When you specify a module to load with the import statement, it first looks in the current directory, and then searches the directories listed in `sys.path`.

```
>>> import sys
>>> sys.path
```

To add locations, put one or more directories to search in the PYTHONPATH environment variable. Separate multiple paths by semicolons for Windows, or colons for Unix/Linux. This will add them to **`sys.path`**, after the current folder, but before the predefined locations.

### Windows

```
set PYTHONPATH=C:\Users\bob\Documents and Settings\Python
```

### Linux/OS X

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



Do not append to `sys.path` in your scripts. This can result in non-portable scripts, and scripts that will fail if the location of the imported modules changes.

## Executing modules as scripts

- `__name__` is name of current module.
  - set to `"__main__"` if run as script
  - set to `"module_name"` if imported
- test with `if __name__ == "__main__"`
- Module can be both run directly and imported

It is sometimes convenient to have a module also be a runnable script. This is handy for testing and debugging, and for providing modules that also can be used as standalone utilities.

Since the interpreter defines its own name as `__main__`, you can test the current namespace's `__name__` attribute.

If it is `__main__`, then you are at the main (top) level of the interpreter, and your file is being run as a script; all of the file's code, including the code under `if __name__ == "__main__"`, is executed.

If it is not `__main__`, then you are importing the file, and the code under `if __name__ == "__main__"` will not be executed.

### script

*(from the command line)*

```
python filename.py
```

`__name__` is set to `"__main__"`

### module

*(in a python script or module)*

```
import module
```

`__name__` is set to `"module"`

## Example

### using\_main.py

```
import sys

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

# constants (AKA global variables)

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

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

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

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

# Packages

- Package is folder containing modules or packages
- Startup code goes in `__init__.py` (optional)

A package is a group of related modules or subpackages. The grouping is physical – a package is a folder that contains one or more modules. It is a way of giving a hierarchical structure to the module namespace so that all modules do not live in the same folder.

A package may have an initialization script named `__init__.py`. If present, this script is executed when the package or any of its contents are loaded. (In Python 2, `__init__.py` was required).

Modules in packages are accessed by prefixing the module with the package name, using the dot notation used to access module attributes.

Thus, if Module **eggs** is in package **spam**, to call the **scramble()** function in **eggs**, you would say `spam.eggs.scramble()`.

By default, importing a package name by itself has no effect; you must explicitly load the modules in the packages. You should usually import the module using its package name, like `from spam import eggs`, to import the eggs module from the spam package.

Packages can be nested.

## Example

```
sound/                Top-level package
  __init__.py         Initialize the sound package (optional)
  formats/            Subpackage for file formats
    __init__.py       Initialize the formats package (optional)
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/            Subpackage for sound effects
    __init__.py       Initialize the formats package (optional)
    echo.py
    surround.py
    reverse.py
    ...
  filters/            Subpackage for filters
    __init__.py       Initialize the formats package (optional)
    equalizer.py
```

```
from sound.formats import aiffread
from sound.effects.surround import dolby
import sound.filters.equalizer as eq
```

## Example: Core Django packages

django	django.contrib.staticfiles
django.apps	django.contrib.syndication
django.conf.urls	django.core.checks
django.contrib.admin	django.core.files
django.contrib.admindocs	django.core.mail
django.contrib.auth	django.core.management
django.contrib.contenttypes	django.db
django.contrib.flatpages	django.db.backends
django.contrib.gis	django.db.migrations
django.contrib.gis.admin	django.db.migrations.operations
django.contrib.gis.db.backends	django.db.models
django.contrib.gis.db.models	django.db.models.fields
django.contrib.gis.forms	django.db.models.functions
django.contrib.gis.gdal	django.dispatch
django.contrib.gis.geoip2	django.forms
django.contrib.gis.geos	django.http
django.contrib.gis.utils	django.middleware
django.contrib.humanize	django.template
django.contrib.messages	django.template.backends
django.contrib.postgres	django.test
django.contrib.postgres.aggregates	django.urls
django.contrib.redirects	django.utils
django.contrib.sessions	django.utils.translation
django.contrib.sitemaps	django.views
django.contrib.sites	



## Configuring import with `__init__.py`

- Provide package documentation
- Load modules into package's namespace for convenience
  - Specify modules to load when `*` is used
- Execute startup code

The docstring in `__init__.py` is used to document the package itself. This is used by IDEs as well as **pydoc**.

For convenience, you can put import statements in a package's `__init__.py` to autoload the modules into the package namespace, so that `import PKG` imports all the (or just selected) modules in the package.

If the variable `__all__` in `__init__.py` is set to a list of module names, then only these modules will be loaded when using `import *`

`__init__.py` can also be used to setup data or other resources that will be used by multiple modules within a package.

```
from PKG import *
```

Given the following package and module layout, the table on the next page describes how `__init__.py` affects imports.

```
my_package
|-----__init__.py
|-----module_a.py
|           function_a()
|-----module_b.py
|           function_b()
|-----module_c.py
|           function_c()
```

Import statement	What it does
If <code>__init__.py</code> is empty	
<code>import my_package</code>	Imports <code>my_package</code> only, but not contents. No modules are imported. This is not useful.
<code>import my_package.module_a</code>	Imports <code>module_a</code> into <code>my_package</code> namespace. Objects in <code>module_a</code> must be prefixed with <code>my_package.module_a</code>
<code>from my_package import module_a</code>	Imports <code>module_a</code> into main namespace. Objects in <code>module_a</code> must be prefixed with <code>module_a</code>
<code>from my_package import module_a, module_b</code>	Imports <code>module_a</code> and <code>module_b</code> into main namespace.
<code>from my_package import *</code>	Does not import anything!
<code>from my_package.module_a import *</code>	Imports all contents of <code>module_a</code> (that do not start with an underscore) into main namespace. Not generally recommended.
If <code>__init__.py</code> contains: <code>all = ['module_a', 'module_b']</code>	
<code>import my_package</code>	Imports <code>my_package</code> only, but not contents. No modules are imported. This is still not useful.
<code>from my_package import module_a</code>	As before, imports <code>module_a</code> into main namespace. Objects in <code>module_a</code> must be prefixed with <code>module_a</code>
<code>from my_package import *</code>	Imports <code>module_a</code> and <code>module_b</code> , but not <code>module_c</code> into main namespace.
If <code>__init__.py</code> contains: <code>all = ['module_a', 'module_b']</code> <code>import module_a</code> <code>import module_b</code>	
<code>import my_package</code>	Imports <code>module_a</code> and <code>module_b</code> into the <code>my_package</code> namespace. Objects in <code>module_a</code> must be prefixed with <code>my_package.module_a</code> . <i>Now this is useful.</i>
<code>from my_package import module_a</code>	Imports <code>module_a</code> into main namespace. Objects in <code>module_a</code> must be prefixed with <code>module_a</code>
<code>from my_package import *</code>	Only imports <code>module_a</code> and <code>module_b</code> into main namespace.
<code>from my_package import module_c</code>	Imports <code>module_c</code> into the main namespace.

## Documenting modules and packages

- Use docstrings
- Described in PEP 257
- Generate docs with Sphinx (optional)

In addition to comments, which are for the *maintainer* of your code, you should add docstrings, which provide documentation for the *user* of your code.

If the first statement in a module, function, or class is an unassigned string, it is assigned as the *docstring* of that object. It is stored in the special attribute `_doc_`, and so is available to code.

The docstring can use any form of literal string, but triple double quotes are preferred, for consistency.

See **PEP 257** for a detailed guide on docstring conventions.

Tools such as pydoc, and many IDEs will use the information in docstrings. In addition, the Sphinx tool will gather docstrings from an entire project and format them as a single HTML, PDF, or EPUB document.

## Python style

- Code is read more often than it is written!
- Style guides enforce consistency and readability

- Indent 4 spaces (do not use tabs)
- Keep lines  $\leq$  79 characters
- Imports at top of script, and on separate lines
- Surround operators with space
- Comment thoroughly to explain why and how code works when not obvious
- Use docstrings to explain how to use modules, classes, methods, and functions
- Use lower\_case\_with\_underscores for functions, methods, and attributes
- Use UPPER\_CASE\_WITH\_UNDERSCORES for globals
- Use StudlyCaps (mixed-case) for class names
- Use `_leading_underscore` for internal (non-API) attributes

Guido van Rossum, Python's BDFL (Benevolent Dictator For Life), once said that code is read much more often than it is written. This means that once code is written, it may be read by the original developer, users, subsequent developers who inherit your code. Do them a favor and make your code readable. This in turn makes your code more maintainable.

To make your code readable, it is import to write your code in a consistent manner. There are several Python style guides available, including PEP (Python Enhancement Proposal) 8, Style Guide for Python Code, and PEP 257, Docstring Conventions.

If you are part of a development team, it is a good practice to put together a style guide for the team. The team will save time not having to figure out each other's style.

## Chapter 8 Exercises

### Exercise 8-1 (potus.py, potus\_main.py)

Create a module named **potus** (potus.py) to provide information from the presidents.txt file. It should provide the following function:

```
get_info(term#) -> dict provide dictionary of info for a specified president
```

Write a script to use the module.

### *For the ambitious* (potus\_amb.py, potus\_amb\_main.py)

Add the following functions to the module

```
get_oldest() -> string return the name of oldest president  
get_youngest()-> string return the name of youngest president
```

'youngest' and 'oldest' refer to age at beginning of first term and age at end of last term.

# Chapter 9: Errors and Logging

## Objectives

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

# Exceptions

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

Even if code is syntactically correct, run-time errors can occur once a script is launched. A common error is to attempt to open a non-existent file. Such errors are also called *exceptions*, and cause the interpreter to stop with an error message.

Python has a hierarchy of builtin exceptions; handling an exception higher in the tree will handle any children of that exception.

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



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

## Example

### exception\_unhandled.py

```
x = 5
y = "cheese"

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

### exception\_unhandled.py 2>&1

```
Traceback (most recent call last):
  File "/Users/jstrick/curr/courses/python/common/examples/exception_unhandled.py", line
  5, in <module>
    z = x + y # Adding a string to an int raises TypeError
      ~~~~
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

## Handling exceptions with try

- Use try/except clauses
- Specify expected exception

To handle an exception, put the code in a `try` block. After the `try` block, you must specify an `except` block with the expected exception. If an exception is raised in the `try` block, execution stops and the interpreter checks to see if the exception matches the `except` block. If it matches, it executes the `except` block and execution continues; otherwise, the exception is treated as fatal (the default behavior) and the interpreter exits.

### Example

#### `exception_simple.py`

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

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

print("After try-except") # Get here whether or not exception occurred
```

#### `exception_simple.py`

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



## Handling multiple exceptions

- Use a tuple of exception names, but with single argument

If your try clause might generate more than one kind of exception, you can specify a tuple of exception types, then the variable which will hold the exception object.

### Example

#### **exception\_multiple.py**

```
try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except (IOError, TypeError) as err: # Use a tuple of 2 or more exception types
    print("Naughty programmer! ", err)
```

#### **exception\_multiple.py**

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

## Handling generic exceptions

- Use **Exception**
- Specify except with no exception list
- Clean up any uncaught exceptions

As a shortcut, you can specify **Exception** or an empty exception list. This will handle any exception that occurs in the try block.

### Example

#### **exception\_generic.py**

```
try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except Exception as err: # Will catch _any_ exception
    print("Naughty programmer! ", err)
```

#### **exception\_generic.py**

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

# Ignoring exceptions

- Use `pass`
- Not usually recommended

Use the `pass` statement to do nothing when an exception occurs

Because the `except` clause must contain some code, the `pass` statement fulfills the syntax without doing anything.

## Example

### `exception_ignore.py`

```
try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except(TypeError, IOError): # Catch exceptions, and do nothing
    pass
```

### `exception_ignore.py`

no output



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

## Using else

- executed if no exceptions were raised
- not required
- can make code easier to read

The last `except` block can be followed by an `else` block. The code in the `else` block is executed only if there were no exceptions raised in the `try` block. Exceptions in the `else` block are not handled by the preceding `except` blocks.

The `else` block lets you make sure that some code related to the `try` clause (and before the `finally` clause) is only run if there's no exception, without trapping the exception specified in the `except` clause.

```
try:
    something_that_can_throw_ioerror()
except IOError as e:
    handle_the_IO_exception()
else:
    # we don't want to catch this IOError if it's raised
    something_else_that_throws_ioerror()
finally:
    something_we_always_need_to_do()
```

## Example

### **exception\_else.py**

```
numpairs = [(5, 1), (1, 5), (5, 0), (0, 5)]

total = 0

for x, y in numpairs:
    try:
        quotient = x / y
    except Exception as err:
        print(f"uh-oh, when y = {y}, {err}")
    else:
        total += quotient # Only if no exceptions were raised
print(total)
```

### **exception\_else.py**

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

## Cleaning up with finally

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

A `finally` block can be used in addition to, or instead of, an `except` block. The code in a `finally` block is executed whether or not an exception occurs. The `finally` block is executed after the `try`, `except`, and `else` blocks.

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

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

## Example

### exception\_finally.py

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

print()

try:
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")
except TypeError as err:
    print("Caught exception:", err)
finally:
    print("Still don't care whether we had an exception")
```

### exception\_finally.py

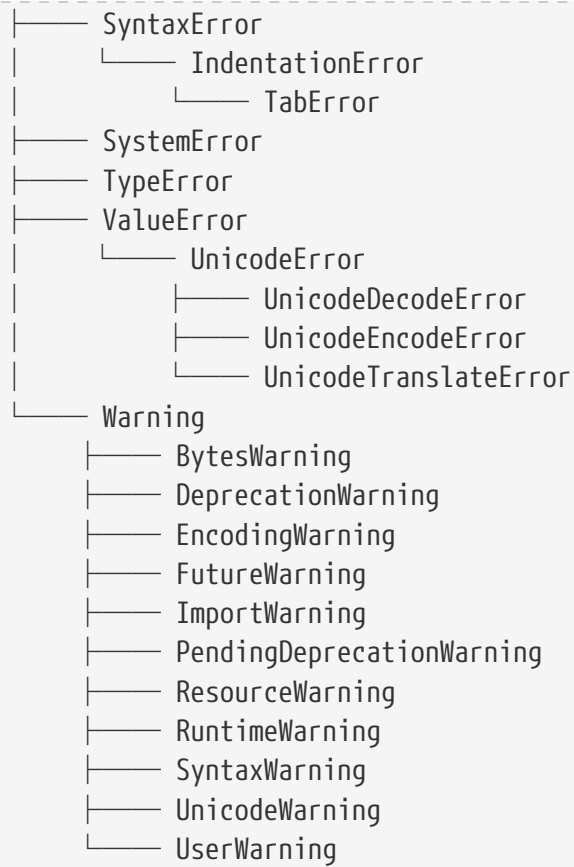
```
z is 42
Don't care whether we had an exception

Caught exception: unsupported operand type(s) for +: 'int' and 'str'
Still don't care whether we had an exception
```

## The Standard Exception Hierarchy (Python 3.11)

```
BaseException
├── BaseExceptionGroup
├── GeneratorExit
├── KeyboardInterrupt
├── SystemExit
├── Exception
│   ├── ArithmeticError
│   │   ├── FloatingPointError
│   │   ├── OverflowError
│   │   └── ZeroDivisionError
│   ├── AssertionError
│   ├── AttributeError
│   ├── BufferError
│   ├── EOFError
│   ├── ExceptionGroup [BaseExceptionGroup]
│   ├── ImportError
│   │   └── ModuleNotFoundError
│   ├── LookupError
│   │   ├── IndexError
│   │   └── KeyError
│   ├── MemoryError
│   ├── NameError
│   │   └── UnboundLocalError
│   ├── OSError
│   │   ├── BlockingIOError
│   │   ├── ChildProcessError
│   │   ├── ConnectionError
│   │   │   ├── BrokenPipeError
│   │   │   ├── ConnectionAbortedError
│   │   │   ├── ConnectionRefusedError
│   │   │   └── ConnectionResetError
│   │   ├── FileExistsError
│   │   ├── FileNotFoundError
│   │   ├── InterruptedError
│   │   ├── IsADirectoryError
│   │   ├── NotADirectoryError
│   │   ├── PermissionError
│   │   ├── ProcessLookupError
│   │   └── TimeoutError
│   ├── ReferenceError
│   ├── RuntimeError
│   │   ├── NotImplementedError
│   │   └── RecursionError
│   ├── StopAsyncIteration
│   └── StopIteration
```





## Simple Logging

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

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

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

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

Table 21. Logging Levels

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

## Example

### logging\_simple.py

```
import logging

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

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

### ../LOGS/simple.log

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

...

## Formatting log entries

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

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

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

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

### Example

#### `logging_formatted.py`

```
import logging

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

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

**../LOGS/formatted.log**

```
INFO root 09/27/24-10:22:20 logging_formatted.py 11 this is information
WARNING root 09/27/24-10:22:20 logging_formatted.py 12 this is a warning
ERROR root 09/27/24-10:22:20 logging_formatted.py 13 this is an ERROR
ERROR root 09/27/24-10:22:20 logging_formatted.py 15 Invalid value 38.7
INFO root 09/27/24-10:22:20 logging_formatted.py 16 this is information
CRITICAL root 09/27/24-10:22:20 logging_formatted.py 17 this is critical
```

Table 22. Log entry formatting directives

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

Table 23. Date Format Directives

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

## Logging exception information

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

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

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

### Example

#### `logging_exception.py`

```
import logging

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

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



**../LOGS/exception.log**

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

## For more information

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

## Chapter 9 Exercises

### Exercise 9-1 (c2f\_loop\_safe.py)

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

### Exercise 9-2 (c2f\_batch\_safe.py)

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

# Chapter 10: Introduction to Python Classes

## Objectives

- Understanding the big picture of OO programming
- Defining a class and its constructor
- Creating object methods
- Adding attributes and properties to a class
- Using inheritance for code reuse
- Adding class data and methods

## About object-oriented programming

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

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

Objects contain both data and the methods (functions) that operate on the data. Each object created has its own personal data, called *instance data*. It can also have data that is shared with all the objects created from its class, called *class data*.

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

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



Methods are not polymorphic; i.e., you can't define multiple versions of a method, with different signatures, and have the corresponding method selected at runtime. However, because Python has dynamic typing, this is seldom needed.

## Defining classes

- Use the **class** statement
- **Syntax**

```
class ClassName(baseclass):  
    pass
```

To create a class, declare the class with the **class** statement. Any base classes may be specified in parentheses, but are not required. If no base classes are needed, you can skip the parentheses.

Classes are conventionally named with UpperCamelCase (i.e., all words, including the first, are capitalized). This is also known as CapWords, StudlyCaps, etc. Modules conventionally have lower-case names. Thus, it is usual to have module **rocketengine** containing class **RocketEngine**.

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

Basic layout of a class:

```
class ClassName(baseclass):  
    classvar = value  
  
    def __init__(self,...):  
        self._attrib = instancevalue;  
        ClassName.attrib = classvalue;  
  
    def method1(self,...):  
        self._attrib = instancevalue  
  
    def method2(self,...):  
        x = self.method1()  
  
    @property  
    def some_property(self, ...):  
        return self.some_attribute
```

## Example

### **simple\_class.py**

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

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

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

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

### **simple\_class.py**

```
hello
hi there
```

## Example

### card.py

```
class Card:
    def __init__(self, rank, suit):
        self._rank = rank
        self._suit = suit

    @property
    def rank(self):
        return self._rank

    @property
    def suit(self):
        return self._suit

    @rank.setter
    def rank(self, value):
        pass

    def __repr__(self):
        return f"Card('{self._rank}', '{self._suit}')"

    def __str__(self):
        return f"{self.rank}-{self.suit}"
```



## Example

### carddeck.py

```

"""
Provide a CardDeck object and some utility methods

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

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

CardList = list[Card]

class CardDeck:
    """
    A deck of 52 cards for playing standard card games
    """
    # def __new__(): pass
    RANKS = '2 3 4 5 6 7 8 9 10 J Q K A'.split()
    CLUB = '\u2663\uFE0F' # FE0F converts chars to emoji
    DIAMOND = '\u2666\uFE0F'
    HEART = '\u2665\uFE0F'
    SPADE = '\u2660\uFE0F'
    SUITS = CLUB, DIAMOND, HEART, SPADE
    DEALER_NAMES = []

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

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

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

```

```

def draw(self) -> Card:
    """
    Retrieve next available card from deck.

    Raises IndexError when deck is empty.

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

def draw_n(self, n) -> CardList:
    """
    Convenience method to draw N cards
    """
    cards: CardList = []
    for _ in range(n):
        cards.append(self.draw())
    return cards

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

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

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

    If invalid type is assigned, raises TypeError

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

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

```

```

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

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

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

    :return:
    """
    return cls.RANKS

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

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

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

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

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

if __name__ == '__main__':
    d = CardDeck("Mary")

```

```
d.shuffle()  
print(d.cards)
```

**carddeck.py**

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

## Example

### play\_cards.py

```
from carddeck import CardDeck

deck = CardDeck("Mary")

deck.shuffle()

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

print(deck)

print(deck.cards)
```

### play\_cards.py

```
3-♠
2-♠
7-♠
J-♠
3-♠
K-♠
Q-♠
Q-♠
4-♠
K-♠

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

...

# Constructors

- Constructor is named `__init__`
- AKA initializer
- Passed *self* plus any parameters

A class's constructor (also known as the initializer) is named `__init__`. It receives the object being created, and any parameters passed into the class when its being created.

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

## Private attributes

Name attributes (variables and methods) of a class with a leading underscore to indicate (in a non-mandatory way) that the variable is *private*. Access to private variables should be provided via *properties*.

## Instance methods

- Expect the object as first parameter
- Object conventionally named *self*
- Otherwise like normal Python functions
- Use *self* to access instance attributes or methods
- Use class name to access class data

Instance methods are defined like normal functions, but like constructors, the object that the method is called from is passed in as the first parameter. As with the constructor, the parameter should be named *self*.



# Properties

- Properties are managed attributes
- Create with `@property` decorator
- Create getter, setter, deleter, docstring
- Specify getter only for read-only property

An object can have properties, or managed attributes. When a property is evaluated, its corresponding getter method is invoked; when a property is assigned to, its corresponding setter method is invoked.

Properties can be created with the `@property` decorator and its derivatives. `@property` applied to a method causes it to be a "getter" method for a property with the same name as the method.

Using `@name.setter` on a method with the same name as the property creates a setter method, and `@name.deleter` on a method with the same name creates a deleter method.

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

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

you can say

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

## properties.py

```

class Person():

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

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

    @first_name.setter # decorator comes from getter property
    def first_name(self, value): # setter property
        if value is None or value.isalpha():
            self._first_name = value
        else:
            raise ValueError("ERROR: First name may only contain letters")

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if value is None or value.isalpha():
            self._last_name = value
        else:
            raise ValueError("Last name may only contain letters")

if __name__ == '__main__':
    person1 = Person('Ferneater', 'Eulalia')
    person2 = Person()
    person2.last_name = 'Pepperpot' # assign to property
    person2.first_name = 'Hortense'

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

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

```

***properties.py***

```
Ferneater Eulalia  
Hortense Pepperpot  
ERROR: First name may only contain letters
```

## Class methods and data

- Defined in the class, but outside of methods
- Defined as attribute of class name (similar to self)
- Define class methods with @classmethod
- Class methods get the class object as 1st parameter

Most classes need to store some data that is common to all objects created in the class. This is generally called class data.

Class attributes can be created by using the class name directly, or via class methods.

A class method is created by using the `@classmethod` decorator. Class methods are implicitly passed the class object.

Class methods can be called from the class object or from an instance of the class; in either case the method is passed the class object.

## Example

### class\_methods\_and\_data.py

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

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

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

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

r = Rabbit("a nice cup of tea")
print(Rabbit.get_location()) # call class method from class
print(r.get_location()) # call class method from instance
```

### class\_methods\_and\_data.py

```
the Cave of Caerbannog
the Cave of Caerbannog
```

## Static Methods

- Define with @staticmethod

A static method is a utility method that is included in the API of a class, but does not require either an instance or a class object. Static methods are not passed any implicit parameters.

Many classes do not need any static methods.

Define static methods with the `@staticmethod` decorator.

### Example

```
class Spam():  
  
    @staticmethod  
    def format_as_title(s): # no implicit parameters  
        return s.strip().title()
```

## Private methods

- Called by other methods in the class
- Should not be used outside class
- Named with leading underscore

"Private" methods are those that are called only within the class. They are not part of the API – they are not visible to users of the class. Private methods may be instance, class, or static methods.

Name private methods with a leading underscore. This does not protect them from use, but gives programmers a hint that they are for internal use only.

# Inheritance

- Specify base classes after class name
- Multiple inheritance OK
- Depth-first, left-to-right search for methods not in derived class

Classes may inherit methods and data. Specify a parenthesized list of base classes after the class name in the class definition.

If a method or attribute is not found in the derived class, it is first looked for in the first base class in the list. If not found, it is sought in the parent of that class, if any, and so on. This is usually called a depth-first search.

The derived class inherits all attributes of the base class. If the base class constructor takes the same arguments as the derived class, then no extra coding is needed. Otherwise, to explicitly call the initializer in the base class, use `super().__init__(args)`.

The simplest derived class would be:

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

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



## Example

### jokerdeck.py

```
from carddeck import CardDeck, Card

JOKER = '\U0001F0CF'

class JokerDeck(CardDeck):

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

## Example

### play\_cards\_jokers.py

```
from jokerdeck import JokerDeck

deck = JokerDeck("Ozzy")

deck.shuffle()

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

print(deck)

print(deck.cards)
```

*play\_cards\_jokers.py*

```

8-♣
10-♣
A-♣
4-♣
10-♣
4-♣
2-♣
10-♣
9-♣
A-♣

JokerDeck(Ozzy,44)
(Card('K', '♠'), Card('3', '♠'), Card('K', '♠'), Card('8', '♠'), Card('♠', '♠'),
Card('6', '♠'), Card('2', '♠'), Card('7', '♠'), Card('J', '♠'), Card('Q', '♠'),
Card('5', '♠'), Card('6', '♠'), Card('Q', '♠'), Card('6', '♠'), Card('K', '♠'),
Card('3', '♠'), Card('7', '♠'), Card('7', '♠'), Card('7', '♠'), Card('♠', '♠'),
Card('9', '♠'), Card('A', '♠'), Card('J', '♠'), Card('10', '♠'), Card('J', '♠'),
Card('4', '♠'), Card('5', '♠'), Card('Q', '♠'), Card('J', '♠'), Card('5', '♠'),
Card('8', '♠'), Card('2', '♠'), Card('9', '♠'), Card('6', '♠'), Card('Q', '♠'),
Card('8', '♠'), Card('2', '♠'), Card('A', '♠'), Card('K', '♠'), Card('9', '♠'),
Card('5', '♠'), Card('3', '♠'), Card('4', '♠'), Card('3', '♠'))

```

## 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 24. Objected-oriented Nomenclature*

Term	Description
attribute	A variable or method that is part of a class or object
base class	A class from which other classes inherit
child class	Same as derived class
class	A Python module from which objects may be created
class method	A function that expects the class object as its first parameter. Such a function can be called from either the class itself or an instance of the class. Created with @classmethod decorator.
constructor	A method that initializes an instance. The constructor receives any arguments passed into the class name.
derived class	A class which inherits from some other class
function	An executable subprogram.
instance method	A function that expects the instance object, conventionally named self, as its first parameter. See "method".
method	A function defined inside a class.
module	A file containing python code, and which is designed to be imported into Python scripts or other modules.
package	A folder containing one or more modules. Packages may be imported. There must be a file named __init__.py in the package folder.
parent class	Same as base class
property	A managed attribute (variable) of an instance of a class
script	A Python program. A script is an executable file containing Python commands.
static method	A function in a class that does not automatically receive any parameters; typically used for private utility functions. Created with @staticmethod decorator.
superclass	Same as base class

## Chapter 10 Exercises

### Exercise 10-1 (knight.py, knight\_info.py)

#### Part A

Create a module which defines a class named **Knight**.

The initializer for the class should expect the knight's name as a parameter. Get the data from **knights.txt** to initialize the object.

The object should have the following read-only properties:

**name**  
**title**  
**favorite\_color**  
**quest**  
**comment**

#### Part B

Create an application to use the **Knight** class created in part one. For each knight specified on the command line, create a **knight** object and print out the knight's name, favorite color, quest, and comment. Precede the name with the knight's title.

## Example

```
python knight_info.py Arthur Bedevere
```

```
Name: King Arthur  
Favorite Color: blue  
Quest: The Grail  
Comment: King of the Britons
```

```
Name: Sir Bedevere  
Favorite Color: red, no, blue!  
Quest: The Grail  
Comment: AARRRRRRRRGGGGHH
```

## Part C

*FOR ADVANCED STUDENTS*

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

## Example

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

# Chapter 11: Metaprogramming

## Objectives

- Learn what metaprogramming means
- Access local and global variables by name
- Inspect the details of any object
- Use attribute functions to manipulate an object
- Design decorators for classes and functions
- Define classes with the `type()` function
- Create metaclasses

# Metaprogramming

- Writing code that writes (or at least modifies) code
- Can simplify some kinds of programs
- Not as hard as you think!
- Considered deep magic in other languages

Metaprogramming is writing code that generates or modifies other code. It includes fetching, changing, or deleting attributes, and writing functions that return functions (AKA factories).

Metaprogramming is easier in Python than many other languages. Python provides explicit access to objects, even the parts that are hidden or restricted in other languages.

For instance, you can easily replace one method with another in a Python class, or even in an object instance. In Java, this would be deep magic requiring many lines of code.

## globals() and locals()

- Contain all variables in a namespace
- `globals()` returns all global objects
- `locals()` returns all local variables

The **`globals()`** builtin function returns a dictionary of all global objects. The keys are the object names, and the values are the objects values. The dictionary is "live" — changes to the dictionary affect global variables.

The **`locals()`** builtin returns a dictionary of all objects in local scope.



## Example

### globals\_locals.py

```
from pprint import pprint # import prettyprint function

spam = 42 # global variable
ham = 'Smithfield'

def eggs(fruit): # function parameters are local
    name = 'Lancelot' # local variable
    idiom = 'swashbuckling' # local variable
    print("Globals:")
    pprint(globals()) # globals() returns dict of all globals
    print()
    print("Locals:")
    pprint(locals()) # locals() returns dict of all locals

eggs('mango')
```

### globals\_locals.py

```
Globals:
{'__annotations__': {},
 '__builtins__': <module 'builtins' (built-in)>,
 '__cached__': None,
 '__doc__': None,
 '__file__': '/Users/jstrick/curr/courses/python/common/examples/globals_locals.py',
 '__loader__': <frozen_importlib_external.SourceFileLoader object at 0x100ce92d0>,
 '__name__': '__main__',
 '__package__': None,
 '__spec__': None,
 'eggs': <function eggs at 0x100d5a840>,
 'ham': 'Smithfield',
 'pprint': <function pprint at 0x100dad9e0>,
 'spam': 42}

Locals:
{'fruit': 'mango', 'idiom': 'swashbuckling', 'name': 'Lancelot'}
```

## Example

### **../DATA/sample.cfg**

```
animal=wombat  
color=ecru  
detective=Wolfe  
language=Python
```

### **config\_to\_variables.py**

```
CONFIG_FILE_PATH = "../DATA/sample.cfg"  
  
g = globals() # get access to dictionary of global names  
  
with open(CONFIG_FILE_PATH) as config_in:  
    for raw_line in config_in:  
        line = raw_line.rstrip()  
        name, value = line.split('=')  
        g[name] = value # create a new global variable  
  
print(animal, color, detective, language)
```

### **config\_to\_variables.py**

```
wombat ecru Wolfe Python
```

# The inspect module

- Simplifies access to metadata
- Easily get metadata

The `inspect` module provides user-friendly functions for accessing the metadata of objects.

This can be useful for troubleshooting, as well as run-time type checking.

For instance, if you have a function that expects a callback function as a parameter, you can use `inspect.getfullargspec()` to make sure the callback has the correct signature (number of parameters).

## Example

### inspect\_ex.py

```
import inspect
import geometry
from carddeck import CardDeck

deck = CardDeck("Leonard")

things = (
    geometry,
    geometry.circle_area,
    CardDeck,
    CardDeck.get_ranks,
    deck,
    deck.shuffle,
)

print("Name      Module?  Function?  Class?  Method?")
for thing in things:
    try:
        thing_name = thing.__name__
    except AttributeError:
        thing_name = type(thing).__name__ + " instance"
    print("{:18s} {!s:6s}  {!s:6s}  {!s:6s}  {!s:6s}".format(
        thing_name,
        inspect.ismodule(thing), # test for module
        inspect.isfunction(thing), # test for function
        inspect.isclass(thing), # test for class
        inspect.ismethod(thing),
    ))

print()
def spam(p1, p2='a', *p3, p4, p5='b', **p6): # define a function
    print(p1, p2, p3, p4, p5, p6)

# get argument specifications for a function
print("Function spec for Ham:", inspect.getfullargspec(spam))
print()

# get frame (function call stack) info
print("Current frame:", inspect.getframeinfo(inspect.currentframe()))
```

***inspect\_ex.py***

Name	Module?	Function?	Class?	Method?
geometry	True	False	False	False
circle_area	False	True	False	False
CardDeck	False	False	True	False
get_ranks	False	False	False	True
CardDeck instance	False	False	False	False
shuffle	False	False	False	True

Function spec for Ham: FullArgSpec(args=['p1', 'p2'], varargs='p3', varkw='p6', defaults=('a',), kwonlyargs=['p4', 'p5'], kwonlydefaults={'p5': 'b'}, annotations={})

Current frame:

Traceback(filename='/Users/jstrick/curr/courses/python/common/examples/inspect\_ex.py',  
lineno=40, function='<module>', code\_context=['print("Current frame:",  
inspect.getframeinfo(inspect.currentframe()))\n'], index=0,  
positions=Positions(lineno=40, end\_lineno=40, col\_offset=24, end\_col\_offset=68))

Table 25. inspect module convenience functions

Function(s)	Description
<code>ismodule(), isclass(), ismethod(), isfunction(), isgeneratorfunction(), isgenerator(), istraceback(), isframe(), iscode(), isbuiltin(), isroutine()</code>	check object types
<code>getmembers()</code>	get members of an object that satisfy a given condition
<code>getfile(), getsourcefile(), getsource()</code>	find an object's source code
<code>getdoc(), getcomments()</code>	get documentation on an object
<code>getmodule()</code>	determine the module that an object came from
<code>get_annotations()</code>	return dictionary of object's annotations
<code>getclasstree()</code>	arrange classes so as to represent their hierarchy
<code>getargspec(), getfullargspec(), getargvalues()</code>	get info about function arguments, return values, and annotations
<code>formatargspec(), formatargvalues()</code>	format an argument spec
<code>getouterframes(), getinnerframes()</code>	get info about frames
<code>currentframe()</code>	get the current stack frame
<code>stack(), trace()</code>	get info about frames on the stack or in a traceback

## Working with attributes

- Objects are dictionaries of attributes
- Special functions can be used to access attributes
- Attributes specified as strings
- Syntax

```
getattr(object, attribute [,defaultvalue] )  
hasattr(object, attribute)  
setattr(object, attribute, value)  
delattr(object, attribute)
```

All Python objects are essentially dictionaries of attributes. There are four special builtin functions for managing attributes. These may be used to programmatically access attributes when you have the name as a string.

**getattr()** returns the value of a specified attribute, or raises an error if the object does not have that attribute. `getattr(a, 'spam')` is the same as `a.spam`. An optional third argument to `getattr()` provides a default value for nonexistent attributes (and does not raise an error).

**hasattr()** returns the value of a specified attribute, or `None` if the object does not have that attribute.

**setattr()** an attribute to a specified value.

**delattr()** deletes an attribute and its corresponding value.

## Example

### attributes.py

```
class Spam():

    def eggs(self, msg): # create attribute
        print("eggs!", msg)

s = Spam()

s.eggs("fried")

print("hasattr()", hasattr(s, 'eggs')) # check whether attribute exists

e = getattr(s, 'eggs') # retrieve attribute
e("scrambled")

def toast(self, msg):
    print("toast!", msg)

setattr(Spam, 'eggs', toast) # set (or overwrite) attribute

s.eggs("battered!")

delattr(Spam, 'eggs') # remove attribute

try:
    s.eggs("shirred")
except AttributeError as err: # missing attribute raises error
    print(err)
```

### attributes.py

```
eggs! fried
hasattr() True
eggs! scrambled
toast! battered!
'Spam' object has no attribute 'eggs'
```



## Adding instance methods

- Use `setattr()`
- Add instance method to class
- Add instance method to instance

Using `setattr()`, it is easy to add instance methods to classes. Just add a function object to the class. Because it is part of the class itself, it will automatically be bound to the instance. Remember that an instance method expects `self` as the first parameter. In fact, this is the meaning of a bound instance — it is "bound" to the instance, and therefore when called, it is passed the instance as the first parameter.

Once added, the method may be called from any existing *or new* instance of the class.

To add an instance method to an *instance* takes a little more effort. Because it's not being added to the class, it is not automatically bound. The function needs to know what instance it should be bound to. This can be accomplished with the `types.MethodType()` function.

Pass the function and the instance to `MethodType()`.

## Example

### adding\_instance\_methods.py

```
from types import MethodType

class Dog(): # Define Dog type
    pass

d1 = Dog() # Create instance of Dog

def bark(self): # Define (unbound) function
    print("Woof! woof!")

setattr(Dog, "bark", bark) # Add function to class (which binds it as an instance
method)

d2 = Dog() # Define another instance of Dog

d1.bark() # New function can be called from either instance
d2.bark()

def wag(self): # Create another unbound function
    print("Wagging...")

setattr(d1, "wag", MethodType(wag, d1)) # Add function to instance after passing it
through MethodType()

d1.wag() # Call instance method
try:
    d2.wag() # Instance method not available - only bound to d1
except AttributeError as err:
    print(err)
```

***adding\_instance\_methods.py***

```
Woof! woof!  
Woof! woof!  
Wagging...  
'Dog' object has no attribute 'wag'
```

## Callable classes

- Convenient for one-method classes
- Really "callable instances"
- Implement `__call__`
- Convenient for one-method classes
- Useful for decorators

Any class instance may be made callable by implementing the special method `call`. This means that rather than saying:

```
sc = SomeClass()  
sc.some_method()
```

you can say

```
sc = SomeClass()  
sc()
```

You can think of a callable class as a function that can also keep some state. As with many object-oriented features, its main purpose is to simplify the user interface.

What's the advantage? In addition to saving state between calls, it saves having to call a method from the instance. Thus, one good use case is for classes that only have one method.

One good use of callable classes is for implementing decorators as classes, rather than functions.



See `memorychecker.py` in the EXAMPLES folder for another example of a callable class to make it easy to check the current memory footprint.

## Example

### incrementer.py

```
class Incrementer:
    def __init__(self):
        self._value = 0

    def __call__(self):
        self._value += 1

    @property
    def value(self):
        return self._value

if __name__ == "__main__":
    inc = Incrementer()
    inc()
    inc()
    inc()
    print(f"inc.value: {inc.value}")
```

### incrementer.py

```
inc.value: 3
```

# Decorators

- Classic design pattern
- Built into Python
- Implemented via functions or classes
- Can decorate functions or classes
- Can take arguments (but not required to)
- `functools.wraps()` preserves function's properties

In Python, many decorators are provided by the standard library, such as `property()` or `classmethod()`

A decorator is a component that modifies some other component. The purpose is typically to add functionality, but there are no real restrictions on what a decorator can do. Many decorators register a component with some other component. For instance, the `@app.route()` decorator in Flask maps a URL to a view function.

As another example, **unittest** provides decorators to skip tests. A very common decorator is **@property**, which converts a class method into a property object.

A decorator can be any *callable*, which means it can be a normal function, a class method, or a class which implements the `__call__()` method (AKA callable class, as discussed earlier).

A simple decorator expects the item being decorated as its parameter, and returns a replacement. Typically, the replacement is a new function, but there is no restriction on what is returned. If the decorator itself needs arguments, then the decorator returns a wrapper function that expects the item being decorated, and then returns the replacement.

Table 26. Decorators in the standard library

Decorator	Description
<code>@abc.abstractmethod</code>	Indicate abstract method (must be implemented).
<code>@abc.abstractproperty</code>	Indicate abstract property (must be implemented). <b>DEPRECATED</b>
<code>@asyncio.coroutine</code>	Mark generator-based coroutine.
<code>@atexit.register</code>	Register function to be executed when interpreter (script) exits.
<code>@classmethod</code>	Indicate class method (receives class object, not instance object)
<code>@contextlib.contextmanager</code>	Define factory function for <b>with</b> statement context managers (no need to create <code>__enter__()</code> and <code>__exit__()</code> methods)
<code>@dataclass</code>	Mark a class as a dataclass from the <code>dataclass</code> module
<code>@functools.cache</code> <code>@functools.lru_cache</code>	Wrap a function with a memoizing callable
<code>@functools.singledispatch</code>	Transform function into a single-dispatch generic function.
<code>@basefunction.register(function)</code>	Register function for single dispatch
<code>@functools.total_ordering</code>	Supply all other comparison methods if class defines at least one.
<code>@functools.wraps</code>	Invoke <code>update_wrapper()</code> so decorator's replacement function keeps original function's name and other properties.
<code>@property</code>	Indicate a class property.
<code>@staticmethod</code>	Indicate static method (passed neither instance nor class object).
<code>@types.coroutine</code>	Transform generator function into a coroutine function.
<code>@unittest.mock.patch</code>	Patch target with a new object. When the function/with statement exits patch is undone.
<code>@unittest.mock.patch.dict</code>	Patch dictionary (or dictionary-like object), then restore to original state after test.
<code>@unittest.mock.patch.multiple</code>	Perform multiple patches in one call.
<code>@unittest.mock.patch.object</code>	Patch object attribute with mock object.
<code>@unittest.skip()</code>	Skip test unconditionally
<code>@unittest.skipIf()</code>	Skip test if condition is true
<code>@unittest.skipUnless()</code>	Skip test unless condition is true
<code>@unittest.expectedFailure()</code>	Mark Test as expected failure
<code>@unittest.removeHandler()</code>	Remove Control-C handler

## Applying decorators

- Use @ symbol
- Applied to *next* item only
- Multiple decorators OK

The @ sign is used to apply a decorator to a function or class. A decorator only applies to the next definition in the script.

For simple decorators, you can also call the decorator, passing in the decorated function, and assigning the result to the original function name.

This leads to the most important thing to remember about the decorators:

```
@spam  
def ham():  
    pass
```

is exactly the same as

```
ham = spam(ham)
```

and

```
@spam(a, b, c)  
def ham():  
    pass
```

is exactly the same as

```
ham = spam(a, b, c)(ham)
```

Once you understand this, then creating decorators is just a matter of writing functions or classes and having them return the appropriate thing.

The last example could also be written as

```
wrapper = spam(a, b, c) # pass args to decorator  
ham = wrapper(ham) # decorator returns wrapper, which wraps the original function
```





A decorator can be applied to a function imported from some other module using the assignment method.

## Example

### **deco\_post\_import.py**

```
from functools import cache
from random import randint
from geometry import circle_area

circle_area = cache(circle_area)
for _ in range(10000): # call circle_area() 10000 times
    result = circle_area(randint(1, 50)) # call with argument in range 1-50

print(circle_area.cache_info()) # show cache hits and misses
```

### **deco\_post\_import.py**

```
CacheInfo(hits=9950, misses=50, maxsize=None, currsize=50)
```

Table 27. Implementing Decorators

Implemented as	Decorates	Takes arguments	How to do it
function	function	N	Decorator function returns replacement function
function	function	Y	Decorator function accept params and returns function that returns replacement function
class	function	N	<code>__init__</code> accepts original function; <code>__call__()</code> is replacement function
class	function	Y	<code>__init__()</code> accepts params; <code>__call__()</code> accepts original function and <i>returns</i> replacement function
function	class	N	Decorator function modifies and returns replacement class
function	class	Y	Decorator function accepts params and <i>returns</i> function that modifies and returns replacement class
class	class	N	<code>__new__()</code> accepts new_class and original class, returns replacement class (which is usually same as original class)
class	class	Y	<code>__init__</code> accepts params; <code>__call__()</code> accepts original class, modifies original class, and then returns replacement class (which is normally also the original class)

# Trivial Decorator

- Decorator can return anything
- Not very useful, usually

A decorator does not have to be elaborate. It can return anything, though typically decorators return the same type of object they are decorating.

In this example, the decorator returns the integer value 42. This is not particularly useful, but illustrates that the decorator always replaces the object being decorated with *something*.

## Example

### **deco\_trivial.py**

```
def void(thing_being_decorated):
    return 42 # replace function with 42

name = "Guido"
x = void(name) # decorate 'name', which is now 42, not a string

@void # decorate hello() function
def hello():
    print("Hello, world")

@void # decorate howdy() function
def howdy():
    print("Howdy, world")

print(hello, type(hello)) # hello is now the integer 42, not a function
print(howdy, type(howdy)) # howdy is now the integer 42, not a function
print(x, type(x))
```

### **deco\_trivial.py**

```
42 <class 'int'>
42 <class 'int'>
42 <class 'int'>
```

## Decorator functions

- Provide a wrapper around a function
- Purposes
  - Add functionality
  - Register
  - ??? (open-ended)
- Optional arguments

A decorator function acts as a wrapper around some object (usually function or class). It allows you to add features to a function without changing the function itself. For instance, the `@property`, `@classmethod`, and `@staticmethod` decorators are used in classes.

A simple decorator function expects only one argument – the function to be modified. It should return a new function, which will replace the original. The replacement function typically calls the original function as well as some new code. More complex decorators expect arguments to the decorator itself. In this case the decorator returns a function that expects the original function, and returns the replacement function.

The new function should be defined with generic arguments (`*args`, `**kwargs`) so it can handle any combination of arguments for the original function.

The **`wraps`** decorator from the `functools` module in the standard library should be used with the function that returns the replacement function. This makes sure the replacement function keeps the same properties (especially the name) as the original (target) function. Otherwise, the replacement function keeps all of its own attributes.

## Example

### deco\_debug.py

```

from functools import wraps

def debugger(old_func): # decorator function -- expects decorated (original) function as
    a parameter

    @wraps(old_func) # @wraps preserves name of original function after decoration
    def new_func(*args, **kwargs): # replacement function; takes generic parameters
        print("*" * 40) # new functionality added by decorator
        print("** function", old_func.__name__, "**") # new functionality added by
decorator

        if args: # new functionality added by decorator
            print(f"\targs are {args}")
        if kwargs: # new functionality added by decorator
            print(f"\tkwargs are {kwargs}")

        print("*" * 40) # new functionality added by decorator

        return old_func(*args, **kwargs) # call the original function

    return new_func # return the new function object

@debugger # apply the decorator to a function
def hello(greeting, whom='world'):
    print(f"{greeting}, {whom}")

hello('hello', 'world') # call new function
print()

hello('hi', 'Earth')
print()

hello('greetings')

```

***deco\_debug.py***

```
*****
** function hello **
   args are ('hello', 'world')
*****
hello, world

*****
** function hello **
   args are ('hi', 'Earth')
*****
hi, Earth

*****
** function hello **
   args are ('greetings',)
*****
greetings, world
```

## Decorator Classes

- Same purpose as decorator functions
- Two ways to implement
  - With arguments
  - Without arguments
- Decorator can keep state

A class can also be used to implement a decorator. The advantage of using a class for a decorator is that a class can keep state, so that the replacement function can update information stored at the class level.

Implementation depends on whether the decorator needs arguments.

If the decorator does *not* need arguments, the class must implement two methods: `__init__()` is passed the original function, and can perform any setup needed. The `__call__` method *replaces* the original function. In other word, after the function is decorated, calling the function is the same as calling `CLASS.__call__`.

If the decorator *does* need arguments, `__init__()` is passed the arguments, and `__call__()` is passed the original function, and must *return* the replacement function.

A good use for a decorator class is to log how many times a function has been called, or even keep track of the arguments it is called with (see example for this).

## Example

### deco\_debug\_class.py

```
class debugger(): # class implementing decorator

    function_calls = []

    def __init__(self, func): # original function passed into decorator's constructor
        self._func = func

    def __call__(self, *args, **kwargs): # __call__() is replacement function

        result = self._func(*args, **kwargs) # call the original function

        self.function_calls.append( # add function name and arguments to saved list
            (self._func.__name__, args, kwargs, result)
        )

        return result # return result of calling original function

    @classmethod
    def get_calls(cls): # define method to get saved function call information
        return cls.function_calls

@debugger # apply debugger to function
def hello(greeting, whom="world"):
    print(f"{greeting}, {whom}")
    return greeting

@debugger # apply debugger to function
def bark(bark_word, *, repeat=2):
    print(f"{bark_word * repeat}!")
    return bark_word

hello('hello', 'world') # call replacement function
print()

hello('hi', 'Earth')
print()

hello('greetings')
```



```

bark("woof", repeat=3)
bark("yip", repeat=4)
bark("arf")

hello('hey', 'you')

print('-' * 60)

for i, info in enumerate(debugger.get_calls(), 1): # display function call info from
class
    print(f"{i:2d}. {info[0]:10s} {info[1]:!s:20s} {info[2]:!s:20s} {info[3]:!s}")

```

### ***deco\_debug\_class.py***

```

hello, world

hi, Earth

greetings, world
woofwoofwoof!
yipypipyip!
arfarf!
hey, you
-----
1. hello      ('hello', 'world')  {}          hello
2. hello      ('hi', 'Earth')  {}          hi
3. hello      ('greetings',)  {}          greetings
4. bark       ('woof',)      {'repeat': 3} woof
5. bark       ('yip',)      {'repeat': 4} yip
6. bark       ('arf',)      {}          arf
7. hello      ('hey', 'you') {}          hey

```

## Decorator arguments

- Decorator functions require two nested functions
- Method `__call__()` returns replacement function in classes

A decorator can be passed arguments. This requires a little extra work.

For decorators implemented as functions, the decorator itself is passed the arguments; it contains a nested function that is passed the decorated function (the target), and it returns the replacement function.

For decorators implemented as classes, `init` is passed the arguments, `__call__()` is passed the decorated function (the target), and `__call__` returns the replacement function.

There are many combinations of decorators (8 total, to be exact). This is because decorators can be implemented as either functions or classes, they may take arguments, or not, and they can decorate either functions or classes.



For an example of all 8 approaches, see the file `decorama.py` in the EXAMPLES folder.



For an example of a decorator with an optional argument, see `deco_params_optional.py` in the EXAMPLES folder.

## Example

### deco\_params.py

```
#

from functools import wraps # wrapper to preserve properties of original function

def multiply(multiplier): # actual decorator -- receives decorator parameters

    def deco(old_func): # "inner decorator" -- receives function being decorated

        @wraps(old_func) # retain name, etc. of original function
        def new_func(*args, **kwargs): # replacement function -- this is called instead
of original
            result = old_func(*args, **kwargs) # call original function and get return
value
            return result * multiplier # multiple result of original function by
multiplier

        return new_func # deco() returns new_function

    return deco # multiply returns deco


@multiply(4)
def spam():
    return 5


@multiply(10)
def ham():
    return 8


a = spam()
b = ham()
print(a, b)
```

### deco\_params.py

```
20 80
```

## Creating classes at runtime

- Use the **type()** function
- Provide dictionary of attributes

A class can be created programmatically, without the use of the class statement. The syntax is

```
type("name", (base_class, ...), {attributes})
```

The first argument is the name of the class, the second is a tuple of base classes (use object if you are not inheriting from a specific class), and the third is a dictionary of the class's attributes.



Instead of `type`, any other *metaclass* can be used.

## Example

### creating\_classes.py

```
def function_1(self): # create method (not inside a class -- could be a lambda)
    print("Hello from f1()")

def function_2(self): # create method (not inside a class -- could be a lambda)
    print("Hello from f2()")

NewClass = type("NewClass", (), { # create class using type() -- parameters are class
    name, base classes, dictionary of attributes
    'hello1': function_1,
    'hello2': function_2,
    'color': 'red',
    'state': 'Ohio',
})

n1 = NewClass() # create instance of new class

n1.hello1() # call instance method
n1.hello2()
print(n1.color) # access class data
print()

SubClass = type("SubClass", (NewClass,), {'fruit': 'banana'}) # create subclass of first
class
s1 = SubClass() # create instance of subclass
s1.hello1() # call method on subclass
print(s1.color) # access class data
print(s1.fruit)
```

***creating\_classes.py***

```
Hello from f1()  
Hello from f2()  
red
```

```
Hello from f1()  
red  
banana
```

# Monkey Patching

- Modify existing class or object
- Useful for enabling/disabling behavior
- Can cause problems

"Monkey patching" refers to technique of changing the behavior of an object by adding, replacing, or deleting attributes from outside the object's class definition.

It can be used for:

- Replacing methods, attributes, or functions
- Modifying a third-party object for which you do not have access
- Adding behavior to objects in memory

If you are not careful when creating monkey patches, some hard-to-debug problems can arise

- If the object being patched changes after a software upgrade, the monkey patch can fail in unexpected ways.
- Conflicts may occur if two different modules monkey-patch the same object.
- Users of a monkey-patched object may not realize which behavior is original and which comes from the monkey patch.

Monkey patching defeats object encapsulation, and so should be used sparingly.

Decorators are a convenient way to monkey-patch a class. The decorator can just add a method to the decorated class.

## Example

### meta\_monkey.py

```
class Spam(): # create normal class

    def __init__(self, name):
        self._name = name

    def eggs(self): # add normal method
        print(f"Good morning, {self._name}. Here are your delicious fried eggs.")

s = Spam('Mrs. Higgenbotham') # create instance of class
s.eggs() # call method

def scrambled(self): # define new method outside of class
    print(f"Hello, {self._name}. Enjoy your scrambled eggs")

setattr(Spam, "eggs", scrambled) # monkey patch the class with the new method

s.eggs() # call the monkey-patched method from the instance
```



***meta\_monkey.py***

Good morning, Mrs. Higgenbotham. Here are your delicious fried eggs.  
Hello, Mrs. Higgenbotham. Enjoy your scrambled eggs

## Do you need a Metaclass?

- Deep magic
- Used in frameworks such as Django
- YAGNI (You Ain't Gonna Need It)

Before we cover the details of metaclasses, a disclaimer: you will probably never need to use a metaclass. When you think you might need a metaclass, consider using inheritance or a class decorator. However, metaclasses may be a more elegant approach to certain kinds of tasks, such as registering classes when they are defined.

There are two use cases where metaclasses are always an appropriate solution, because they must be done before the class is created:

- Modifying the class name
- Modifying the the list of base classes.

Several popular frameworks use metaclasses, Django in particular. In Django they are used for models, forms, form fields, form widgets, and admin media.

Remember that metaclasses can be a more elegant way to accomplish things that can also be done with inheritance, composition, decorators, and other techniques that are less "magic".

# About metaclasses

- Metaclass:Class::Class:Object

Just as a class is used to create an instance, a *metaclass* is used to create a class.

The primary reason for a metaclass is to provide extra functionality at *class creation* time, not *instance creation* time. Just as a class can share state and actions across many instances, a metaclass can share (or provide) data and state across many *classes*.

The metaclass might modify the list of base classes, or register the class for later retrieval.

The builtin metaclass that Python provides is **type**.

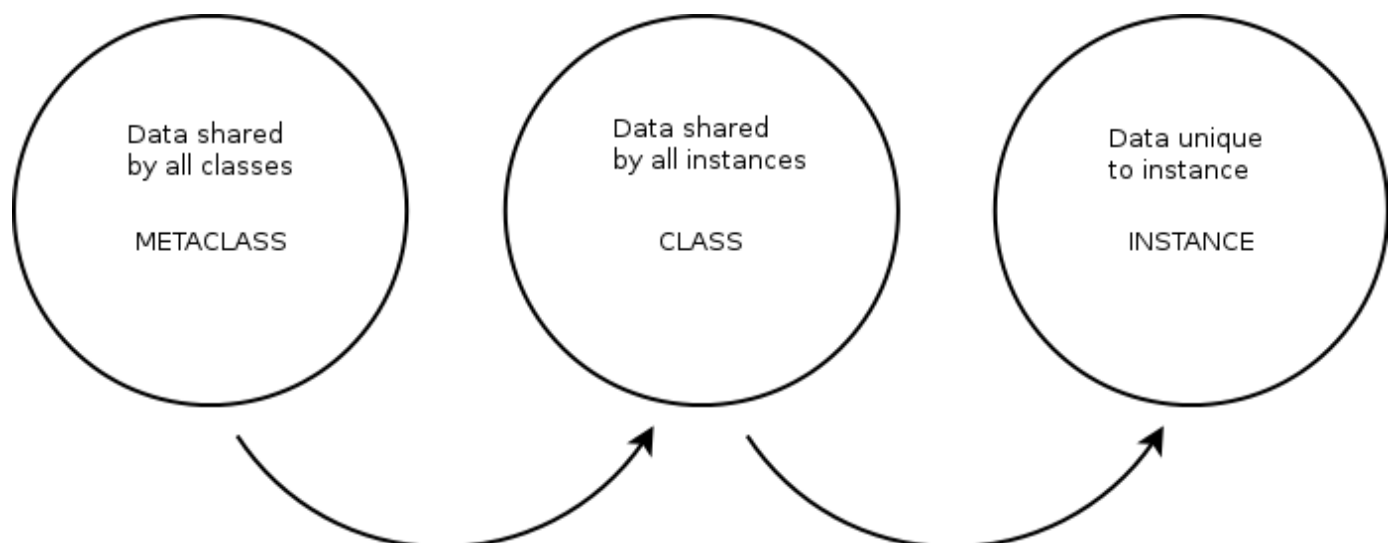
As we saw earlier ,you can create a class from a metaclass by passing in the new class's name, a tuple of base classes (which can be empty), and a dictionary of class attributes (which also can be empty).

```
class Spam(Ham):  
    id = 1
```

is exactly equivalent to

```
Spam = type('Spam', (Ham,), {"id": 1})
```

Replacing "type" with the name of any other metaclass works the same.



## Mechanics of a metaclass

- Like normal class
- *Should* implement `__new__`
- *Can* implement
  - `__init__`
  - `__prepare__`
  - `__call__`

To create a metaclass, define a normal class. Most metaclasses implement the `__new__` method. This method is called with the type, name, base classes, and attribute dictionary (if any) of the new class. It should return a new class, typically using `super().__new__()`, which is very similar to how normal classes create instances. This is one place you can modify the class being created. You can add or change attributes, methods, or properties.

For instance, the Django framework uses metaclasses for Models. When you create an instance of a Model, the metaclass code automatically creates methods for the fields in the model. This is called "declarative programming", and is also used in SQLAlchemy's declarative model, in a way pretty similar to Django.

When you execute the following code:

```
class SomeClass(metaclass=SomeMeta):  
    pass
```

`META(name, bases, attrs)` is executed, where `META` is the metaclass (normally `type()`). Then,

1. The `__prepare__` method of the metaclass is called
2. The `__new__` method of the metaclass is called
3. The `__init__` method of the metaclass is called.

Next, after the following code runs:

```
obj = SomeClass()
```

`SomeMeta.call()` is called. It returns whatever `SomeMeta.__new__()` returned.

`__prepare__()` `__new__()` `__init__()` `__call__()`

## Example

### metaclass\_generic.py

```
class Meta(type):

    def __prepare__(class_name, bases):
        """
        "Prepare" the new class. Here you can update the base classes.

        :param name: Name of new class as a string
        :param bases: Tuple of base classes
        :return: Dictionary that initializes the namespace for the new class (must be a
        dict)
        """
        print(f"in metaclass (class={class_name}) __prepare__()", end=' ==> ')
        print(f"params: name={class_name}, bases={bases}")
        return {'animal': 'wombat', 'id': 100}

    def __new__(metatype, name, bases, attrs):
        """
        Create the new class. Called after __prepare__(). Note this is only called when
        classes

        :param metatype: The metaclass itself
        :param name: The name of the class being created
        :param bases: bases of class being created (may be empty)
        :param attrs: Initial attributes of the class being created
        :return:
        """
        print(f"in metaclass (class={name}) __new__()", end=' ==> ')
        print(f"params: type={metatype} name={name} bases={bases} attrs={attrs}")
        return super().__new__(metatype, name, bases, attrs)

    def __init__(cls, *args):
        """
        :param cls: The class being created (compare with 'self' in normal class)
        :param args: Any arguments to the class
        """
        print(f"in metaclass (class={cls.__name__}) __init__()", end=' ==> ')
        print(f"params: cls={cls}, args={args}")

        super().__init__(cls)

    def __call__(self, *args, **kwargs):
        """
```

Function called when the metaclass is called, as in `NewClass = Meta(...)`

```

:param args:
:param args:
:param kwargs:
:return:
"""
    print(f"in metaclass (class={self.__name__})__call__()")

```

```

class MyBase():
    pass

```

```
print('=' * 60)
```

```

class A(MyBase, metaclass=Meta):
    id = 5

    def __init__(self):
        print("In class A __init__()")

```

```
print('=' * 60)
```

```

class B(MyBase, metaclass=Meta):
    animal = 'wombat'

    def __init__(self):
        print("In class B __init__()")

```

```

print('=' * 60)
m1 = A()
print('=' * 60)
m2 = B()
print('=' * 60)
m3 = A()
print('=' * 60)
m4 = B()
print('=' * 60)
print(f"animal: {A.animal} id: {B.id}")

```

**metaclass\_generic.py**

```

=====
in metaclass (class=A) __prepare__() ==> params: name=A, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=A) __new__() ==> params: type=<class '__main__.Meta'> name=A
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 5, '__module__':
'__main__', '__qualname__': 'A', '__init__': <function A.__init__ at 0x102305bc0>}
in metaclass (class=A) __init__() ==> params: cls=<class '__main__.A'>, args=('A',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 5, '__module__': '__main__',
'__qualname__': 'A', '__init__': <function A.__init__ at 0x102305bc0>})
=====
in metaclass (class=B) __prepare__() ==> params: name=B, bases=(<class
'__main__.MyBase'>,)
in metaclass (class=B) __new__() ==> params: type=<class '__main__.Meta'> name=B
bases=(<class '__main__.MyBase'>,) attrs={'animal': 'wombat', 'id': 100, '__module__':
'__main__', '__qualname__': 'B', '__init__': <function B.__init__ at 0x102305d00>}
in metaclass (class=B) __init__() ==> params: cls=<class '__main__.B'>, args=('B',
(<class '__main__.MyBase'>,), {'animal': 'wombat', 'id': 100, '__module__': '__main__',
'__qualname__': 'B', '__init__': <function B.__init__ at 0x102305d00>})
-----
in metaclass (class=A) __call__()
-----
in metaclass (class=B) __call__()
-----
in metaclass (class=A) __call__()
-----
in metaclass (class=B) __call__()
-----
animal: wombat id: 100

```

## Singleton with a metaclass

- Classic example
- Simple to implement
- Works with inheritance

One of the classic use cases for a metaclass in Python is to create a *singleton* class. A singleton is a class that only has one actual instance, no matter how many times it is instantiated. Singletons are used for loggers, config data, and database connections, for instance.

To create a single, implement a metaclass by defining a class that inherits from **type**. The class should have a class-level dictionary to store each class's instance. When a new instance of a class is created, check to see if that class already has an instance. If it does not, call `__call__` to create the new instance, and add the instance to the dictionary.

In either case, then return the instance where the key is the class object.



## Example

### metaclass\_singleton.py

```
class Singleton(type): # use type as base class of a metaclass
    _instances = {} # dictionary to keep track of instances

    def __new__(typ, *junk):
        # print("__new__()")
        return super().__new__(typ, *junk)

    def __call__(cls, *args, **kwargs): # __call__ is passed the new class plus its
        # parameters
        # print("__call__()")
        if cls not in cls._instances: # check to see if the new class has already been
            # instantiated
            cls._instances[cls] = super().__call__(*args, **kwargs) # if not, create the
            # (single) class instance and add to dictionary

        return cls._instances[cls] # return the (single) class instance

class ThingA(metaclass=Singleton): # Define two different classes which use Singleton
    def __init__(self, value):
        self.value = value

class ThingB(metaclass=Singleton): # Define two different classes which use Singleton
    def __init__(self, value):
        self.value = value

ta1 = ThingA(1) # Create instances of ThingA and ThingB
ta2 = ThingA(2)
ta3 = ThingA(3)

tb1 = ThingB(4)
tb2 = ThingB(5)
tb3 = ThingB(6)

for thing in ta1, ta2, ta3, tb1, tb2, tb3:
    print(type(thing).__name__, id(thing), thing.value) # Print the type, name, and ID
    # of each thing -- only one instance is ever created for each class
```

***metaclass\_singleton.py***

```
ThingA 4530415504 1
ThingA 4530415504 1
ThingA 4530415504 1
ThingB 4530415824 4
ThingB 4530415824 4
ThingB 4530415824 4
```

## Chapter 11 Exercises

### Exercise 11-1 (pres\_attr.py)

Instantiate the President class. Get the first name, last name, and party attributes using `getattr()`.

### Exercise 11-2 (pres\_monkey.py, pres\_monkey\_amb.py)

Monkey-patch the President class to add a method `get_full_name` which returns a single string consisting of the first name and the last name, separated by a space.



Instead of a method, make `full_name` a property.

### Exercise 11-3 (sillystring.py)

Without using the **class** statement, create a class named `SillyString`, which is initialized with any string. Include an instance method called `every_other` which returns every other character of the string.

Instantiate your string and print the result of calling the **every\_other()** method. Your test code should look like this:

```
ss = SillyString('this is a test')
print(ss.every_other())
```

It should output

```
ti sats
```

### Exercise 11-4 (doubledeco.py)

Write a decorator to double the return value of any function. If a function returns 5, after decoration it should return 10. If it returns "spam", after decoration it should return "spamsam", etc.

### Exercise 11-5 (word\_actions.py)

Write a decorator, implemented as a class, to register functions that will process a list of words. The decorated functions will take one parameter — a string — and return the modified string.

The decorator itself takes two arguments — minimum length and maximum length. The class will store the min/max lengths as the key, and the functions as values, as class data.

The class will also provide a method named **process\_words**, which will open **DATA/words.txt** and read it line by line. Each line contains a word.

For every registered function, if the length of the current word is within the min/max lengths, call all the functions whose key is that min/max pair.

In other words, if the registry key is (5, 8), and the value is [func1, func2], when the current word is within range, call func1(*w*) and func2(*w*), where *w* is the current word.

Example of class usage:

```
word_select = WordSelect() # create callable instance

@word_select(16, 18) # register function for length 16-18, inclusive
def make_upper(s):
    return s.upper()

word_select.process_words() # loop over words, call functions if selected
```

Suggested functions to decorate:

- make the word upper-case
- put stars before or around the word
- reverse the word

Remember all the decorated functions take one argument, which is one of the strings in the word list, and return the modified word.

# Chapter 12: Type Hinting

## Objectives

- Learn how to annotate variables with their type
- Understand what the type hints do **not** provide
- Employ the `typing` module to annotate collections
- Use the `Union` and `Optional` types correctly
- Write stub interfaces using type hints

## Type Hinting

Python supports optional type hinting of variables, functions, and parameters. **This is not enforced by the Python interpreter.**

Types may be specified with the declaration of a variable:

```
count: int = 0
```

Once again, this is type **hinting** only; the Python interpreter does not check the types in any way:

```
# Valid Python, incorrect intent  
valid: dict = (3, 'hello')
```

Python functions use a `->` to indicate a return type; function parameters are annotated with type information in the same way as any variable.

```
def shout(word: str, times: int = 1) -> str:  
    return word.upper() * times
```

Argument lists and keyword argument lists may be type-hinted, the values are then expected to uniformly be of that type. For keyword arguments, the keys are still strings; only the values get the type hint.

```
def shout_various(*, **kwargs: int) -> None:  
    for word in kwargs:  
        print(word.upper() * kwargs[word])
```

## Static Analysis Tools

If these type hints are not used by the Python interpreter, how are they useful? While Python (currently) does not use the type hints in any way, static analysis tools do. The most common tool for static type analysis is the `mypy` module. This is not part of the standard distribution and is a separate `pip` install.

```
> pip install mypy
> python3 -m mypy multishout.py
```

The `mypy` module will scan the code (technically, AST of the code) and try to determine, at "compile" time, whether the types expected and used match up correctly. It will emit errors when it detects static typing problems in the code.

`mypy` even supports scanning the inline arbitrary code that may be present in a format-string literal.

```
word: str = 'hello'
# mypy will report an error on the next line
print(f'{word + 3}')
```

`mypy` will emit errors, warnings, and notes of what it finds. While the output is quite configurable, most projects would benefit from fixing any and all issues found by `mypy`.

## Runtime Analysis Tools

In theory, Python's type hints can be checked at runtime. Unfortunately, such checking involves a very high performance cost.

### `__annotations__`

The mechanism behind how type hints are tracked is through a dictionary called `__annotations__`. There is one at the global level, for any global variables declared with type hinting. There also exist `__annotations__` dictionary attributes on both functions and classes. (Notably, while such annotations may be made on local variables, Python does not track these programmatically; there is no local `__annotations__` dictionary. Static tools such as `mypy` are expected to parse these annotations themselves, rather than having the runtime cost of the interpreter parsing them and filling in another dictionary.)

This `__annotations__` dictionary actually just stores key: value pairs of the string version of the variable name, and the type specified. Interestingly, the "type" could be any Python value!

```
def weird(word: len, times: 0 = 1) -> 'unk':  
    return word * times
```

This is perfectly valid Python code. The `__annotations__` dictionary stores the value of each annotation.

```
>>> weird.__annotations__  
{'word': <built-in function len>, 'times': 0, 'return': 'unk'}
```

The return type of the function is stored in the key `return`. This ensures that no parameter name will conflict with the return type (as `return` is a keyword).



## Forward References

Not all types may be available at the time that a given piece of Python code is compiled to bytecode. In other words, **forward references**, where a type is referred to before it is defined, are needed. The standard way to do this in Python is by using strings; tools are expected to handle this forward reference.

```
class First:
    ...
    # The type Second is not yet available
    # to python, so it must be
    # forward-declared using a string
    def process(self, item: 'Second') -> str:
        ...

class Second:
    ...
    # The type First is available to
    # python, so it can just reference
    # the First symbol directly
    def create(self, data: First) -> str:
        ...
```

Other languages use similar concepts for declaring a type without defining it. The `mypy` tool deals correctly with these forward references.

Forward references are also how various operator overloads may need to be written, to refer to the current class.

```
class Matrix:
    def __matmul__(self, obj: 'Matrix') -> 'Matrix':
        ...
```

Forward references can also just be part of a type hint, they need not "gobble up" the entire type hint.

```
class Tree:
    def leaves(self) -> List['Tree']:
        ...
```

## typing Module

The `typing` module makes it easier to refer to containers as a kind of type in Python code. To declare that a tuple has specific types of fields:

```
from typing import Tuple

# Expects a three-tuple with types of str, int, and float
def process(record: Tuple[str, int, float]) -> None:
    ...
```

The `typing` module makes available the many type-wrapper classes. See <https://docs.python.org/3/library/typing.html> for the complete list.

## Input Types

Most input types should not be of type `List`, but rather how the list is used, so either an `Iterable` or a `Collection`. Most of the more-specific containers (such as `Dict`, `Set`, and `List`) should generally only find use as return values.

`Tuple` objects generally specify exactly which type each positional value is, such as `Tuple[str, int, str]`. Tuples of arbitrary length (but the same type throughout) may be specified using the `Ellipsis` object.

```
def ziptuple(words: Tuple[str, ...], times: Tuple[int, ...]) -> Generator[str, None, None]:
    for s, i in zip(words, times):
        yield s * i
```

The `typing.Generator` type takes exactly three types for its template: the type yielded, the type sent, and the type returned by the generator. If any of those types are not used, they should be set to `None`. Unusually, the send-type for a generator is contravariant (because a generator is a function).

```
def primes() -> Generator[int, None, None]:
    yield 2
    yield 3
    yield 5

def process() -> Generator[int, Manager, None]:
    value = 0
    while True:
        sent = yield value
        value = sent.employee_id

results = process();
for c in results:
    x = lookup_employee(c.boss)
    # Valid,
    results.send(x)
```

## Creating Types

Creating new type aliases in Python follows the same process as creating any other kind of variable in Python.

```
FrequencyDict = Dict[str, int]
LangMap = Dict[str, FrequencyDict]

def determine_language(d: LangMap) -> None:
    ...
```

When working with generic types, it can be important to signal that a given type is maintained throughout a function call. Consider a function that filters a value from a container:

```
from typing import TypeVar, Sequence

Choice = TypeVar('Choice')

def third(coll: Sequence[Choice]) -> Choice:
    return coll[2]
```

In this case, a generic type is needed (rather than inheriting from a specific type), so a new one is created with `TypeVar`. This class can restrict itself to a specific set of types:

```
from typing import TypeVar, Sequence

Numeric = TypeVar('Numeric', int, float)

def first(coll: Sequence[Numeric]) -> Numeric:
    return coll[0]
```

In addition, `TypeVar` also allows for fine-grained control of the variance of a given container's type.

## Variance

When dealing with types and inheritance, certain interactions between things that contain a type need to be made explicit. For instance, given a relationship of `Cat`, `Mammal`, `Animal`, and `Dog`, consider the following method:

```
def brush_hair(a: Mammal): ...
```

It makes logical sense that any subtype of `Mammal` would be able to have its hair brushed. So, passing a subtype to `brush_hair` should be acceptable to the type system. This kind of relationship between an argument's type and its inheritance is known as **covariance**: the subtype of an argument can be used in the place of the argument's type. This is also known as the **Liskov Substitution Principle**. But a function with a different signature involves subtle traps.

```
def brush_hair_all(a: List[Mammal]) -> None: ...

def add_cat(a: List[Mammal]) -> None: ...
```

If a function existed that would brush the hair of a whole collection of `Mammals`, it makes sense that there might be both `Cat` and `Dog` objects in the collection, all of which get their hair brushed. Similarly, a `List[Cat]` collection could be passed to the function. This function is covariant.

But, suppose a function with an identical signature existed, `add_cat`. It takes a collection of `Mammal` objects and adds a cat, which at first glance seems acceptable. The problem is that this function could have been passed a `List[Dog]` collection, and adding a `Cat` to it would violate its type! Some languages handle this at runtime, even though it can be discovered statically.

On the other hand, if a `List[Mammal]` or `List[Animal]` collection was passed, the `add_cat` function would work perfectly well. Note that these are both supertypes, rather than subtypes. That means that the `add_cat` function is **contravariant**: the supertype of an argument can be used in place of the argument's type. This form of variance can be noted expressly in Python's type-hinting system.

```
T_co = TypeVar('T_co', covariant=True)
T_contra = TypeVar('T_contra', contravariant=True)

class ListOrMoreSpecific(List[T_co]): pass
class ListOrLessSpecific(List[T_contra]): pass

def brush_hair_all(a: ListOrMoreSpecific[Mammal]) -> None:
    for x in a:
        print(f'Pet the {x.hair()}')

def add_cat(a: ListOrLessSpecific[Cat]) -> None:
```

```
a.append(Cat())
```

The default variance of the collections in the `typing` module are **invariant**, which means that only the exact type specified is permitted.

# Union and Optional

Python's type system is actually quite robust from a theoretical perspective, allowing a form of tagged unions and optional types.

## Union Types

A **Union** type is allowed to be one of a number of possible types. For instance, a value that may be either a **float** or a **str** may be written as:

```
apartment: Union[int, str]
if apartment.isinstance(int):
    print(f'Apartment #{apartment}')
else:
    print(f'Apartment {apartment}')
```

**Union**s may specify any number of valid types. It is the job of the calling code to tease out the correct type if they must be treated differently.

```
def destroy(junk: Union[Car, Refrigerator, ACUnit]) -> None:
    if junk.isinstance(Refrigerator):
        junk.remove_door()
    elif junk.isinstance(Car):
        crush(junk)
    else:
        junk.drain_freon()
```

## Optional Types

An important specific case of **Union** types is the **Optional** type. An **Optional** type is one that is either **None**, or a specified type.

```
def get_record(id: int) -> Optional[Record]:
    row = db.query('WHERE id = ?', id)
    if not row:
        return None
    else:
        return row
```

With Python's support for exceptions, this may seem unusual. But, there are cases where a value may be present, or absent. The **Optional** type is excellent for type-checking these cases, as **mypy** will detect if the wrapped type is being used without a branch for checking the **None** possibility.

```
def annoy_cat(times: Optional[int]) -> str:
    # This line generates the mypy output:
    #note: Right operand is of type "Optional[int]"
    return 'meow' * times
```

This allows for dealing with detectable default values in a type-safe way.

```
# print the phrase some number of times, unless the number is not specified
def print_times(phrase: str, times: Optional[int] = None) -> None:
    if times is None:
        print(phrase)
    else:
        print(phrase * times)
```



## multimethod and functools.singledispatch

One of the utility of types in a programming language is the ability to pick a specific function or method based on its arguments' types. Baked into the standard library is the `functools.singledispatch` decorator. This involves decorating a function, and then defining multiple implementations with different types for the **first** argument.

```
@functools.singledispatch
def wound(x):
    print(f'Hit the {x}')

@wound.register
def _(x: Twistable):
    print(f'Wind the {x}')
```

Note that any additional registered functions must **not** share the same function or method name as the originally decorated function, unlike the `@property` decorator.

The functions are typed **only** on the basis of the first argument. Having a different number of arguments, or differently-typed later arguments, will not be used by `@singledispatch`.

### multimethod

`multimethod` is a third-party library that provides even greater support for multiple dispatch functions. It provides a decorator, `@multimethod`, that takes into account the types of all function arguments.

```
@multimethod
def drop_bass(loc: Location, m: Music):
    print('WUBWUBWUBWUBWUBWUB DRRRRRR')
    ...

@multimethod
def drop_bass(loc: Location, f: Fish):
    print('Sploosh')
    ...
```

Further, the `@multimethod` decorator may be passed the keyword `strict`, which will raise an exception if no function signature can be found with the passed-in types.

```
@multimethod(strict=True)
def drop_bass(loc: Location, f: Fish):
    print('Sploosh')
    ...
```

```
b = StringBass()  
# Raises TypeError  
drop_bass(concert, b)
```

## Stub Type Hinting

While type hinting can be a useful tool, it can be limiting when integrating with other source code. If a third-party library does not use type hints, then the type errors generated by that module may not be correctable in the current project.

The `mypy` utility can actually read a separate file to find out the type interface of a different module.

```
$ export MYPYPATH=$VIRTUAL_ENV/stubs
```

This path-setting may be best in a given virtual environment, so that different versions of a given library can have appropriate type hinting. In the stub directory, create a Python interface (`.pyi`) with the same name as the module to annotate. Then fill the Python interface file with an outline of the public types of the module to annotate.

```
class Card:
    suit: str
    rank: int
    def __repr__(self) -> str: ...

    def deal(deck: 'Deck', players: int = ...) -> Tuple[Deck]: ...
```

In this code sample, the ellipses are **literal** ellipses. The `Ellipsis` singleton is used by `mypy` to allow Python to parse the interface without running any code.

Now, `mypy` is able to parse this interface file to determine the correct annotated types, and can perform static analysis on code using the referenced module.

## Chapter 12 Exercises

### Exercise 12-1 (grep\_sed.py)

Write type hints for the following code, rewriting the code as necessary:

```
def grep(x):  
    '''Only emits lines of text that contain x'''  
    while True:  
        coll = yield  
        if x in coll:  
            yield x  
  
def sed(pattern, replacement):  
    '''Replace any lines containing pattern with replacement'''  
    while True:  
        value = yield  
        if pattern in value:  
            yield replacement
```

### Exercise 12-2 (sow.py)

Write an overloaded function `sow` that behaves differently when it is passed a `Pig` object versus a `Seed` object.

### Exercise 12-3 (append\_42.py)

Write and annotate a function `append_42` that appends a `42` to the argument, a list. If no list is supplied, a new one should be created and returned.

### Exercise 12-4 (double\_arg\_deco.py)

Write a decorator `@double` that will double all numerically-annotated arguments passed to the function it decorates.

# Chapter 13: Developer Tools

## Objectives

- Run pylint to check source code
- Debug scripts
- Find speed bottlenecks in code
- Compare algorithms to see which is faster

# Program development

- More than just coding
  - Design first
  - Consistent style
  - Comments
  - Debugging
  - Testing
  - Documentation

# Comments

- Keep comments up-to-date
- Use complete sentences
- Block comments describe a section of code
- Inline comments describe a line
- Don't state the obvious

Comments that contradict the code are worse than no comments. Always make a priority of keeping the comments up-to-date when the code changes!

Comments should be complete sentences. If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code. Each line of a block comment starts with a # and a single space (unless it is indented text inside the comment).

Paragraphs inside a block comment are separated by a line containing a single #.

Use inline comments sparingly. Inline comments should be separated by at least two spaces from the statement; they should start with a # and a single space.

Inline comments are unnecessary and in fact distracting if they state the obvious. Don't do this:

```
x = x + 1      # Increment x
```

Only use an inline comment if the reason for the statement is not obvious:

```
x = x + 1      # Add one so range() does the right thing
```

*The above was adapted from PEP 8*

# pylint

- Checks many aspects of code
- Finds mistakes
- Rates your code for standards compliance
- Don't worry if your code has a lower rating!
- Can be highly customized

*pylint is a Python source code analyzer which looks for programming errors, helps enforcing a coding standard and sniffs for some code smells (as defined in Martin Fowler's Refactoring book)*

*from the pylint documentation*

**pylint** can be very helpful in identifying errors and pointing out where your code does not follow standard coding conventions. It was developed by Python coders at Logilab <http://www.logilab.fr>.

It has very verbose output, which can be modified via command line options.

pylint can be customized to reflect local coding conventions.

pylint usage:

```
pylint filename(s) or directory
pylint -ry filename(s) or directory
```

The **-ry** option says to generate a full detailed report.

Most Python IDEs have pylint, or the equivalent, built in.

Other tools for analyzing Python code:

- pyflakes
- pychecker



# Customizing pylint

- Use `pylint --generate-rcfile`
- Redirect to file
- Edit as needed
- Knowledge of regular expressions useful
- Use `-rcfile` file to specify custom file on Windows
- Name file `~/.pylintrc` on Linux/Unix/OS X

To customize pylint, run pylint with only the `-generate-rcfile` option. This will output a well-commented configuration file to STDOUT, so redirect it to a file.

Edit the file as needed. The comments describe what each part does. You can change the allowed names of variables, functions, classes, and pretty much everything else. You can even change the rating algorithm.

## Windows

Put the file in a convenient location (name it something like `pylintrc`). Invoke pylint with the `-rcfile` option to specify the location of the file.

pylint will also find a file named `pylintrc` in the current directory, without needing the `-rcfile` option.

## Non-Windows systems

On Unix-like systems (Mac, Linux, etc.), `/etc/pylintrc` and `~/.pylintrc` will be automatically loaded, in that order.

See [docs.pylint.org](https://docs.pylint.org) for more details.

## Using pyreverse

- Source analyzer
- Reverse engineers Python code
- Part of `pylint`
- Generates UML diagrams

**pyreverse** is a Python source code analyzer. It reads a script, and the modules it depends on, and generates UML diagrams. It is installed as part of the `pylint` package.

There are many options to control what it analyzes and what kind of output it produces.

Use `-A` to search all ancestors, `-p` to specify the project name, `-o` to specify output type (e.g., **pdf**, **png**, **jpg**). See the output of `pyreverse -h` for all options.

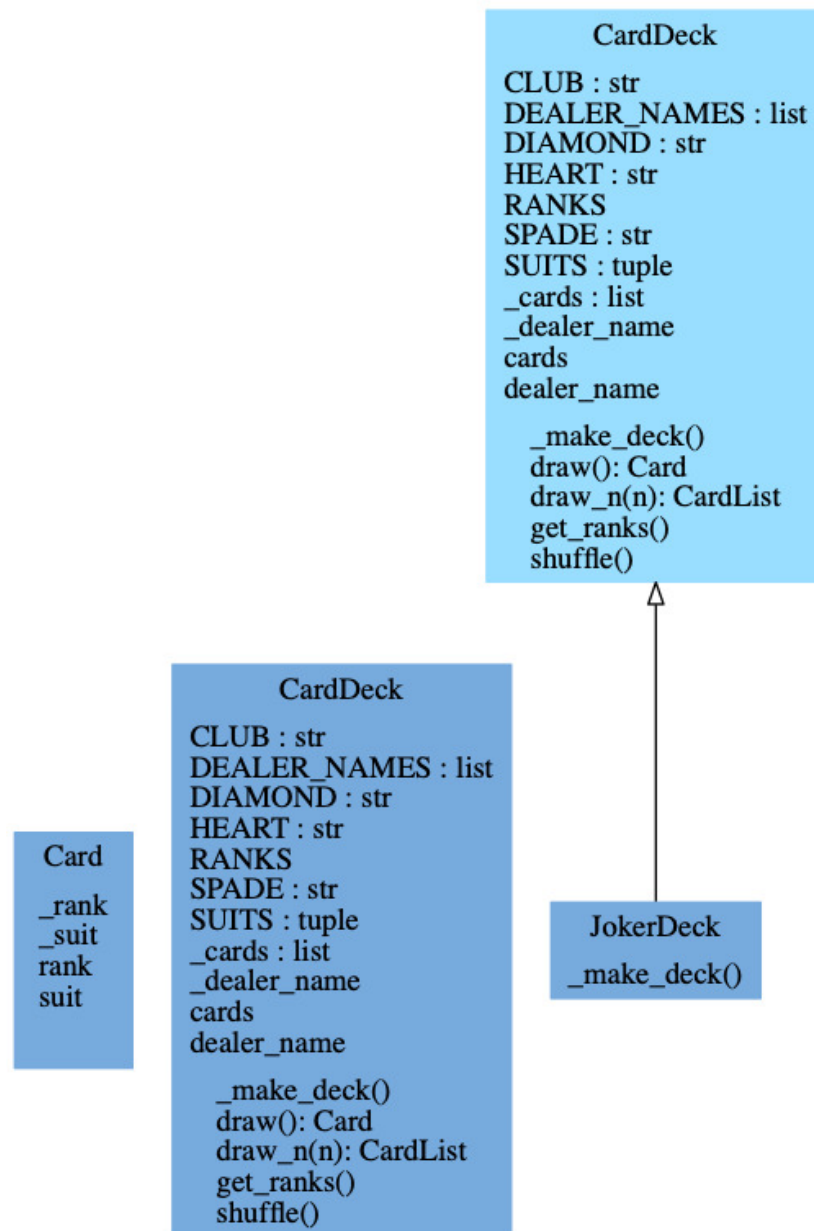


`pyreverse` requires **Graphviz**, a graphics tool that must be installed separately from Python

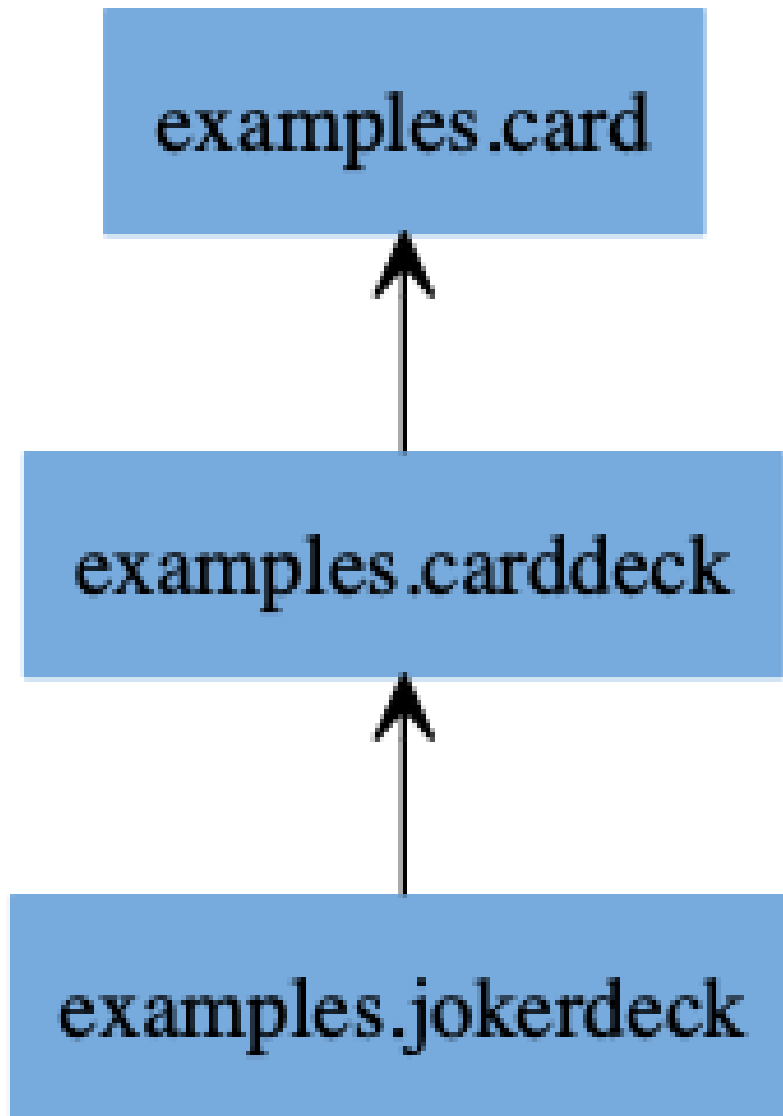
### Example

```
$ pyreverse -d ../images -p carddeck -f SPECIAL -S -A -o png --colorized card carddeck  
jokerdeck
```

classes\_carddeck.png



packages\_carddeck.png



# The Python debugger

- Implemented via **pdb** module
- Supports breakpoints and single stepping
- Based on **gdb**

While most IDEs have an integrated debugger, it is good to know how to debug from the command line. The **pdb** module provides debugging facilities for Python.

The usual way to use **pdb** is from the command line:

```
python -mpdb script_to_be_debugged.py
```

Once the program starts, it will pause at the first executable line of code and provide a prompt, similar to the interactive Python prompt. There is a large set of debugging commands you can enter at the prompt to step through your program, set breakpoints, and display the values of variables.

Since you are in the Python interpreter as well, you can enter any valid Python expression.

You can also start debugging mode from within a program.

## Starting debug mode

- `python -m pdb script`

pdb is usually invoked as a script to debug other scripts. For example:

```
python -m pdb myscript.py
```

Table 28. Common debugging commands

Command	Description
<code>n</code>	execute next line, stepping over functions
<code>s</code>	execute next line, stepping over functions
<code>b n</code>	set breakpoint at line n
<code>b function</code>	set breakpoint at function
<code>c</code>	continue to next breakpoint
<code>r</code>	return from function
<code>h</code>	show help

To get more help, type `h` at the debugger prompt.

## Typical usage

```
python -m pdb play_cards.py
> /Users/jstrick/curr/courses/python/common/examples/play_cards.py(1)<module>()
➔ from carddeck import CardDeck
(Pdb) l
1 ➔ from carddeck import CardDeck
2
3     deck = CardDeck("Mary")
4
5     deck.shuffle()
6
7     for _ in range(10):
8         card = deck.draw()
9         print(card)
10    print()
11
(Pdb) b 5
(Pdb) b CardDeck.draw
Breakpoint 1 at /Users/jstrick/curr/courses/python/common/examples/play_cards.py:5
(Pdb) c
> /Users/jstrick/curr/courses/python/common/examples/play_cards.py(5)<module>()
➔ deck.shuffle()
(Pdb) s
--Call--
> /Users/jstrick/curr/courses/python/common/examples/carddeck.py(51)shuffle()
➔ def shuffle(self):
(Pdb) n
> /Users/jstrick/curr/courses/python/common/examples/carddeck.py(57)shuffle()
➔ random.shuffle(self._cards)
(Pdb) p self._cards
```

## Stepping through a program

- **s** single-step, stepping into functions
- **n** single-step, stepping over functions
- **r** return from function
- **c** run to next breakpoint or end

The debugger provides several commands for stepping through a program. Use **s** to step through one line at a time, stepping into functions.

Use **n** to step over functions; use **r** to return from a function; use **c** to continue to next breakpoint or end of program.

Pressing **ENTER** repeats most commands; if the previous command was **l** (list), the debugger lists the next set of lines.

## Setting breakpoints

```
b b linenumber (, condition) b file:linenumber (, condition) b function name (, condition)
```

Breakpoints can be set with the **b** command. Specify a line number, or a function name, optionally preceded by the filename that contains it.

Any of the above can be followed by an expression (use comma to separate) to create a conditional breakpoint.

The **tbreak** command creates a one-time breakpoint that is deleted after it is hit the first time.



# Profiling

- Use the `profile` module from the command line
- Shows where program spends the most time
- Output can be tweaked via options

Profiling is the technique of discovering the part of your code where your application spends the most time. It can help you find bottlenecks in your code that might be candidates for revision or refactoring.

To use the profiler, execute the following at the command line:

```
python -m profile scriptname.py
```

This will output a simple report to STDOUT.

You can also specify an output file with the `-o` option, which can be examined with the `???` utility.

To order by a specific column, use the `-s` option with any of the column names.

See <https://docs.python.org/3/library/profile.html> for more information.



The `pycallgraph2` module (third-party module) will create a graphical representation of an application's profile, indicating visually where the application is spending the most time.

## Example

```
python -m profile count_with_dict.py
...script output...
19 function calls in 0.000 seconds
```

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
14	0.000	0.000	0.000	0.000	:0(get)
1	0.000	0.000	0.000	0.000	:0(items)
1	0.000	0.000	0.000	0.000	:0(open)
1	0.000	0.000	0.000	0.000	:0(setprofile)
1	0.000	0.000	0.000	0.000	count_with_dict.py:3(<module>)
1	0.000	0.000	0.000	0.000	profile:0(<code object <module> at 0xb74c36e0, file "count_with_dict.py", line 3>)
0	0.000		0.000		profile:0(profiler)

# Benchmarking

- Use the `timeit` module
- Create a `Timer` object with specified # of repetitions

Use the `timeit` module to benchmark two or more code snippets. To time code, create a `Timer` object, which takes two strings of code. The first is the code to test; the second is setup code, that is only run once per timer .

Call the `.timeit()` method with the number of times to call the test code, or call the `.repeat()` method which repeats `.timeit()` a specified number of times.

You can also use `timeit` from the command line. Use `-s` to specify startup code:

```
python -m timeit -s 'startup code...' 'code...'.
```

## Example

### bm\_range\_vs\_while.py

```
from timeit import Timer

REPEATS = 10000

setup = """
values = []
""" # setup code is only executed once

code_snippets = [
    '''
for i in range(10000):
    values.append(i)
values.clear()
''', # code fragment executed many times
    '''
i = 0
while i < 10000:
    values.append(i)
    i += 1
values.clear()
''', # code fragment executed many times
]

for code_snippet in code_snippets:
    t = Timer(code_snippet, setup)
    print(f"{code_snippet:80.80s}{t.timeit(REPEATS)}")
    print('-' * 60)
```

***bm\_range\_vs\_while.py***

```
for i in range(10000):  
    values.append(i)  
values.clear()  
2.7564368689781986  
-----
```

```
i = 0  
while i < 10000:  
    values.append(i)  
    i += 1  
values.clear()  
3.743688105023466  
-----
```

## For more information

- <https://pylint.readthedocs.io/en/latest/index.html>
- <https://graphviz.org/>
- <https://mermaid.js.org/>
- <https://docs.python.org/3/library/profile.html>
- <https://web.archive.org/web/20170318204046/http://lanyrd.com/2013/pycon/scdywg/>

## Chapter 13 Exercises

### Exercise 13-1

Pick several of your scripts (from class, or from real life) and run pylint on them.

### Exercise 13-2

Use the builtin debugger or one included with your IDE to step through any of the scripts you have written so far.

# Chapter 14: Consuming RESTful Data

## Objectives

- Writing REST clients
  - Opening URLs and downloading data
  - Fetching data from RESTful servers
- Use the requests module to simplify authentication and proxies
- Uploading data to a server



# The REST API

- Based on HTTP verbs
  - GET get all objects or details on one
  - POST add a new object
  - PUT update an object (all fields)
  - PATCH update an object (some fields)
  - DELETE delete an object

REST stands for **R**epresentational **S**tate **T**ransfer, first described (and named) by Roy Fielding in 2000. It is not a protocol or structure, but rather an architectural style resulting from a set of guidelines. It provides for loosely-coupled resource management over HTTP. REST does not enforce a particular implementation.

A RESTful site provides *resources*, which contain *records*. The same API typically contains more than one resource; each has a different *endpoint* (URL). For instance, <https://sandbox-api.brewerydb.com/v2/> has resources **beer**, **brewery**, and **ingredient**.

A RESTful API uses HTTP verbs to manipulate records. The same endpoint can be used for all access; what happens depends on a combination of which HTTP verb is used, plus whether there is more information on the URL.

If it is just the endpoint (e.g. [www.wombats.com/api/v1/wombat](http://www.wombats.com/api/v1/wombat)):

- GET retrieves a list of all resources. Query strings can be used to sort or filter the list.
- POST adds a new resource

If it is the endpoint plus more information, typically a primary key (e.g. [www.wombats.com/api/v1/wombat/1](http://www.wombats.com/api/v1/wombat/1)):

- GET retrieves details for that resource
- PUT updates the resource (replaces all fields)
- PATCH updates the resource (replaces some fields)
- DELETE removes the resource

A list of public RESTful APIs is located here: [http://www.programmableweb.com/category/all/apis?data\\_format=21190](http://www.programmableweb.com/category/all/apis?data_format=21190)

Table 29. RESTful requests

Verb	URL	Description
GET	/API/wombat	Get list of all wombats
POST	/API/wombat	Create a new wombat
GET	/API/wombat/ <i>ID</i>	Get details of wombat by id
PUT	/API/wombat/ <i>ID</i>	Update wombat by id
DELETE	/API/wombat/ <i>ID</i>	Delete wombat by id



see <https://restfulapi.net/resource-naming/> for more information on designing a RESTful interface

## When is REST not REST?

- REST is guidelines, not protocol
- Implementers are not consistent
- YMMV (your mileage may vary)

REST is a set of guidelines, not a specific protocol. Because of this, REST implementations vary widely. For instance, many APIs use more than one endpoint for the same resource. Many APIs do not return a list of links on a GET request to the endpoint, but the details for every resource. Many APIs misuse (from the strict REST point of view) extended URLs.

It is thus important to read the docs for each individual API to see exactly what they expect, and what they provide.

The good news is that the variations are mostly in the URLs you need to construct — making the requests and parsing out the data are generally about the same for most APIs.

## Consuming REST APIs

- Use `requests.method()`
- Specify parameters, headers, etc. as dictionary
- Use `response.content` to get raw data
- Use `response.json()` to convert JSON to Python

The **requests** module is used to get data from RESTful APIs.

To add GET parameters, pass a dictionary of key-value parameters with the **params** keyword argument:

```
payload = { 'key1': 'value1', ... }  
requests.get(URL, params=payload)
```

Likewise, for POST data, pass a dictionary with the **post** argument.

To use a proxy, pass a dictionary of protocol/url parameters with the **proxies** keyword

```
proxies={'http':'http://proxy.something.com:1234'}  
requests.get(URL, proxies=proxies)
```

```
proxies={'http':'http://proxy.something.com:1234'}
```

To convert a JSON response into a Python data structure, use the `json()` method on the **response** object.



See the quickstart guide for requests at: <http://www.python-requests.org/en/latest/user/quickstart/>

## Example

### rest\_consumer\_omdb.py

```
import requests
from pprint import pprint

with open('omdbapikey.txt') as api_in:
    OMDB_API_KEY = api_in.read().rstrip()

OMDB_URL = "http://www.omdbapi.com"

def main():
    requests_params = {'t': 'Black Panther', "apikey": OMDB_API_KEY}
    response = requests.get(OMDB_URL, params=requests_params)
    if response.status_code == requests.codes.OK:
        raw_data = response.json()

        print(f"raw_data['Title']: {raw_data['Title']}")
        print(f"raw_data['Director']: {raw_data['Director']}")
        print(f"raw_data['Year']: {raw_data['Year']}")
        print(f"raw_data['Runtime']: {raw_data['Runtime']}")
        print()

        print('-' * 60)

        print("raw DATA:")
        pprint(response.json())
    else:
        print(f"response.status_code: {response.status_code}")

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

### rest\_consumer\_omdb.py

```
raw_data['Title']: Black Panther
raw_data['Director']: Ryan Coogler
raw_data['Year']: 2018
raw_data['Runtime']: 134 min

-----

raw DATA:
{'Actors': "Chadwick Boseman, Michael B. Jordan, Lupita Nyong'o",
 'Awards': 'Won 3 Oscars. 124 wins & 290 nominations total',
 'BoxOffice': '$700,426,566',
```

```
'Country': 'United States',
'DVD': 'N/A',
'Director': 'Ryan Coogler',
'Genre': 'Action, Adventure, Sci-Fi',
'Language': 'English, Swahili, Nama, Xhosa, Korean',
'Metascore': '88',
'Plot': "T'Challa, heir to the hidden but advanced kingdom of Wakanda, must "
        'step forward to lead his people into a new future and must confront '
        "a challenger from his country's past.",
'Poster': 'https://m.media-
amazon.com/images/M/MV5BMTg1MTY2MjYzNV5BMl5BanBnXkFtZTgwMTc4NTMwNDI@._V1_SX300.jpg',
'Production': 'N/A',
'Rated': 'PG-13',
'Ratings': [{'Source': 'Internet Movie Database', 'Value': '7.3/10'},
             {'Source': 'Rotten Tomatoes', 'Value': '96%'},
             {'Source': 'Metacritic', 'Value': '88/100'}],
'Released': '16 Feb 2018',
'Response': 'True',
'Runtime': '134 min',
'Title': 'Black Panther',
'Type': 'movie',
'Website': 'N/A',
'Writer': 'Ryan Coogler, Joe Robert Cole, Stan Lee',
'Year': '2018',
'imdbID': 'tt1825683',
'imdbRating': '7.3',
'imdbVotes': '850,236'}
```

Table 30. Keyword Parameters for **requests** methods

Option	Data Type	Description
<code>allow_redirects</code>	<code>bool</code>	set to True if PUT/POST/DELETE redirect following is allowed
<code>auth</code>	<code>tuple</code>	authentication pair (user/token,password/key)
<code>cert</code>	<code>str</code> or <code>tuple</code>	path to cert file or ('cert', 'key') tuple
<code>cookies</code>	<code>dict</code> or <code>CookieJar</code>	cookies to send with request
<code>data</code>	<code>dict</code>	parameters for a POST or PUT request
<code>files</code>	<code>dict</code>	files for multipart upload
<code>headers</code>	<code>dict</code>	HTTP headers
<code>json</code>	<code>str</code>	JSON data to send in request body
<code>params</code>	<code>dict</code>	parameters for a GET request
<code>proxies</code>	<code>dict</code>	map protocol to proxy URL
<code>stream</code>	<code>bool</code>	if False, immediately download content
<code>timeout</code>	<code>float</code> or <code>tuple</code>	timeout in seconds or (connect timeout, read timeout) tuple
<code>verify</code>	<code>bool</code>	if True, then verify SSL cert



These can be used with any of the HTTP request types, as appropriate.

Table 31. `requests.Response` methods and attributes

Method/attribute	Definition
<code>apparent_encoding</code>	Returns the apparent encoding
<code>close()</code>	Closes the connection to the server
<code>content</code>	Content of the response, in bytes
<code>cookies</code>	A Cookiejar object with the cookies sent back from the server
<code>elapsed</code>	A timedelta object with the time elapsed from sending the request to the arrival of the response
<code>encoding</code>	The encoding used to decode <code>r.text</code>
<code>headers</code>	A dictionary of response headers
<code>history</code>	A list of response objects holding the history of request (url)
<code>is_permanent_redirect</code>	True if the response is the permanent redirected url, otherwise False
<code>is_redirect</code>	True if the response was redirected, otherwise False
<code>iter_content()</code>	Iterates over the response
<code>iter_lines()</code>	Iterates over the lines of the response
<code>json()</code>	Converts JSON content to Python data structure (if the result was written in JSON format, if not it raises an error)
<code>links</code>	The header links
<code>next</code>	A PreparedRequest object for the next request in a redirection
<code>ok</code>	True if <code>status_code</code> is less than 400, otherwise False
<code>raise_for_status()</code>	If an error occur, this method a <code>HTTPError</code> object
<code>reason</code>	A text corresponding to the status code
<code>request</code>	The request object that requested this response
<code>status_code</code>	A number that indicates the status (200 is OK, 404 is Not Found)
<code>text</code>	The content of the response, in unicode
<code>url</code>	The URL of the response



# Printing JSON

- Default output of JSON is ugly
- `pprint` makes structures human friendly
- Use `pprint.pprint()`

When debugging JSON data, the `print` command is not so helpful. The Python data structure parsed from JSON is just printed out all jammed together, one element after another, and is hard to read.

The `pprint` (pretty print) module will analyze a structure and print it out with indenting, to make it much easier to read.

Before Python 3.6, dictionaries used to store keys in indeterminate order. Because of this, the default is for `pprint()` to display dictionary keys in sorted order. When viewing JSON, it usually makes more sense to view the keys in the same order they were in the JSON file. To do this, add the `sort_dicts=True` parameter to `pprint()`.

You can customize the output even more with other parameters: `indent` (default 1) specifies how many spaces to indent nested structures; `width` (default 80) constrains the width of the output; `depth` (default unlimited) says how many levels to print – levels beyond depth are shown with '...'.

## Requests sessions

- Share configuration across requests
- Can be faster
- Instance of `requests.Session`
- Supports context manager (*with* statement)

To make it more convenient to share HTTP information across multiple requests, **requests** provides *sessions*. You can use a session by creating an instance of **requests.Session**. Once the session is created, it contains attributes containing the named arguments to request methods, such as `session.params` or `session.headers`.

A Session object implements the context manager protocol, so it can be used with the **with** statement. This will automatically close the session.

These are normal Python dictionaries, so you can use the *update* method to add information:

```
with requests.Session() as session:
    session.params.update({'token': 'MY_TOKEN'})
    session.headers.update({'accept': 'application/xml'})

    response1 = session.get(...)
    response2 = session.get(...)
```

Then you can call the HTTP methods (**get**, **post**, etc.) from the session object. Any parameters passed to an HTTP method will overwrite those that already exist in the session, but do not update the session.



For more examples of using REST in a real-world situation, see `consume_omdb_main.py` in the **EXAMPLES** folder. This script imports various modules that connect to the OMDB API using various multitasking approaches.

## Example

### rest\_consumer\_omdb\_sessions.py

```
import requests
from pprint import pprint

with open('omdbapikey.txt') as api_in:
    OMDB_API_KEY = api_in.read().rstrip()

OMDB_URL = "http://www.omdbapi.com"

MOVIE_TITLES = [
    'Black Panther',
    'Frozen',
    'Top Gun: Maverick',
    'Bullet Train',
    'Death on the Nile',
    'Casablanca',
]

def main():
    with requests.Session() as session:
        session.params.update({"apikey": OMDB_API_KEY})
        for movie_title in MOVIE_TITLES:
            params = {'t': movie_title}
            response = session.get(OMDB_URL, params=params)
            if response.status_code == requests.codes.OK:
                raw_data = response.json()
                print(f"raw_data['Title']: {raw_data['Title']}")
                print(f"raw_data['Director']: {raw_data['Director']}")
                print(f"raw_data['Year']: {raw_data['Year']}")
                print(f"raw_data['Runtime']: {raw_data['Runtime']}")
                print()

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

***rest\_consumer\_omdb\_sessions.py***

```
raw_data['Title']: Black Panther
raw_data['Director']: Ryan Coogler
raw_data['Year']: 2018
raw_data['Runtime']: 134 min

raw_data['Title']: Frozen
raw_data['Director']: Chris Buck, Jennifer Lee
raw_data['Year']: 2013
raw_data['Runtime']: 102 min

raw_data['Title']: Top Gun: Maverick
raw_data['Director']: Joseph Kosinski
raw_data['Year']: 2022
raw_data['Runtime']: 130 min

raw_data['Title']: Bullet Train
raw_data['Director']: David Leitch
raw_data['Year']: 2022
raw_data['Runtime']: 127 min

raw_data['Title']: Death on the Nile
raw_data['Director']: Kenneth Branagh
raw_data['Year']: 2022
raw_data['Runtime']: 127 min

raw_data['Title']: Casablanca
raw_data['Director']: Michael Curtiz
raw_data['Year']: 1942
raw_data['Runtime']: 102 min
```

## Authentication with requests

- Options
  - Basic-Auth
  - Digest
  - Custom
- Use **auth** argument

**requests** makes it easy to provide basic authentication to a web site.

In the simplest case, create a `requests.auth.HTTPBasicAuth` object with the username and password, then pass that to requests with the `auth` argument. Since this is a common use case, you can also just pass a `(user, password)` tuple to the `auth` parameter.

For digest authentication, use `requests.auth.HTTPDigestAuth` with the username and password.

For custom authentication, you can create your own auth class by inheriting from `requests.auth.AuthBase`.

For OAuth 1, OAuth 2, and OpenID, install `requests-oauthlib`. This additional module provides auth objects that can be passed in with the `auth` parameter, as above.

See <https://docs.python-requests.org/en/latest/user/authentication/> for more details.

## Example

### http\_basic\_auth.py

```
import requests
from requests.auth import HTTPBasicAuth, HTTPDigestAuth

# base URL for httpbin
BASE_URL = 'https://httpbin.org'

# formats for httpbin
BASIC_AUTH_FMT = "/basic-auth/{}/{}"
DIGEST_AUTH_FMT = "/digest-auth/{}/{}/{}"

USERNAME = "spam"
PASSWORD = "ham"
BAD_PASSWORD = "toast"

REPORT_FMT = "{:35s} {}"

def main():
    basic_auth()
    digest()

def basic_auth():
    auth = HTTPBasicAuth(USERNAME, PASSWORD)
    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth good password", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, PASSWORD),
        auth=(USERNAME, PASSWORD),
    )
    print(REPORT_FMT.format("Basic auth good password (shortcut)", response))

    response = requests.get(
        BASE_URL + BASIC_AUTH_FMT.format(USERNAME, BAD_PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Basic auth bad password", response))

def digest():
    auth = HTTPDigestAuth(USERNAME, PASSWORD)
    response = requests.get(
```

```
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth good password", response))

    auth = HTTPDigestAuth(USERNAME, BAD_PASSWORD)
    response = requests.get(
        BASE_URL + DIGEST_AUTH_FMT.format('WOMBAT', USERNAME, PASSWORD),
        auth=auth,
    )
    print(REPORT_FMT.format("Digest auth bad password", response))

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

### ***http\_basic\_auth.py***

Basic auth good password	<Response [200]>
Basic auth good password (shortcut)	<Response [200]>
Basic auth bad password	<Response [401]>
Digest auth good password	<Response [200]>
Digest auth bad password	<Response [401]>

## Posting data to a RESTful server

- use `requests.post(url, data=dict)`

The **POST** operation adds a new record to a resource using the endpoint.

To post to a RESTful service, use the **data** argument to **request's post** function. It takes a dictionary of parameters, which will be URL-encoded automatically.

If the POST is successful, the server should return response data with a link to the newly created record, with an HTTP response code of 201 ("created") rather than 200 ("OK").



## Example

### post\_to\_rest.py

```
from datetime import datetime
import time
import requests

URL = 'http://httpbin.org/post'

for i in range(3):
    response = requests.post( # POST data to server
        URL,
        data={'date': datetime.now(),
              'label': 'test_' + str(i)
            },
        cookies={'python': 'testing'},
        headers={'X-Python': 'Guido van Rossum'},
    )
    if response.status_code in (requests.codes.OK, requests.codes.created):
        print(response.status_code)
        print(response.text)
        print()
        time.sleep(2)
```

***post\_to\_rest.py***

```
200
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "date": "2024-09-27 10:22:32.683265",
    "label": "test_0"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "48",
    "Content-Type": "application/x-www-form-urlencoded",
    "Cookie": "python=testing",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-66f6bfa8-27c4a25542fac3c235824c9f",
    "X-Python": "Guido van Rossum"
  },
  "json": null,
  "origin": "69.218.218.146",
  "url": "http://httpbin.org/post"
}

200
{
  "args": {},
  "data": "",
```

...

## Other operations

- Use **requests** functions
  - put
  - delete
  - patch
  - head
  - options

For other HTTP operations, **requests** provides appropriately named functions.

## Example

### other\_web\_service\_ops.py

```
import requests

print('PUT:')
r = requests.put("http://httpbin.org/put", data={'spam': 'ham'}) # send data via HTTP PUT
request
print(r.status_code, r.text)
print('-' * 60)

print('DELETE:')
r = requests.delete("http://httpbin.org/delete") # send HTTP DELETE request
print(r.status_code, r.text)
print('-' * 60)

print('HEAD:')
r = requests.head("http://httpbin.org/get") # get HTTP headers via HEAD request
print(r.status_code, r.text)
print(r.headers)
print('-' * 60)

print('OPTIONS:')
r = requests.options("http://httpbin.org/get") # get negotiated HTTP options
print(r.status_code, r.text)
print('-' * 60)
```

**other\_web\_service\_ops.py**

```

PUT:
200 {
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "spam": "ham"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "8",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-66f6bfb0-0602071c2d48dab31a4eeeff"
  },
  "json": null,
  "origin": "69.218.218.146",
  "url": "http://httpbin.org/put"
}

```

```

-----
DELETE:
200 {
  "args": {},
  "data": "",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "0",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.31.0",
    "X-Amzn-Trace-Id": "Root=1-66f6bfb1-4577d2124e6786ce5a950986"
  },
  "json": null,
  "origin": "69.218.218.146",
  "url": "http://httpbin.org/delete"
}

```

```

-----
HEAD:
200

```

```
{'Date': 'Fri, 27 Sep 2024 14:22:42 GMT', 'Content-Type': 'application/json', 'Content-  
Length': '307', 'Connection': 'keep-alive', 'Server': 'unicorn/19.9.0', 'Access-Control-  
Allow-Origin': '*', 'Access-Control-Allow-Credentials': 'true'}
```

-----  
OPTIONS:

200  
-----

# Using Postman

- Graphical HTTP Client
- Specify headers and parameters
- Save searches

**postman** is a GUI-based HTTP client. It lets you specify URLs, plus headers, parameters, data, and any other information that needs to be sent to a web server. You can save searches for replay, and easily make changes to the requests.

This makes it really easy to experiment with REST endpoints without having to constantly re-run your Python scripts.

It also displays the JSON (or other) response in various modes, including pretty-printed with syntax highlighting.



**postman** is free for individuals and small teams, or can be subscribed to for larger teams

Get **postman** at <https://www.getpostman.com/>

## Chapter 14 Exercises

### Exercise 14-1 (noaa\_precip.py)

NOAA provides an API for climate data. The base page is <https://www.ncdc.noaa.gov/cdo-web/webservices/v2#gettingStarted>.

You will need to request a token via email. Specify the token with the `token` header. Use the `headers` parameter to `requests.get()`.

You can get a list of the web services as described on the page.

For hourly precipitation, the dataset ID is "PRECIP\_HLY".

The station ID for Boaz, Alabama is "COOP:010957"

Using your token, the dataset ID, and the station ID, fetch hourly precipitation data for that location from January 1, 1970 through January 31, 1970.

The endpoint is

<https://www.ncdc.noaa.gov/cdo-web/api/v2/data>

and you will need to specify the parameters **datasetid**, **stationid**, **startdate**, and **enddate**.

Remember that only a certain number of values are returned with each call. For purposes of this lab, just get the first page of values.

Try varying the parameters, based on the API docs.

See `ANSWERS/noaa_metadata.py` for a script to retrieve values for some of the endpoints.



# Chapter 15: Database Access

## Objectives

- Understand the Python DB API architecture
- Connect to a database
- Execute simple and parameterized queries
- Fetch single and multiple row results
- Execute non-query statements
- Get metadata about a query
- Start transactions and commit or rollback as needed

## The DB API

- Most popular Python DB interface
- Specification, not abstract class
- Many modules for different DB implementations
- Hides actual DBMS implementation

To make database programming simpler and more consistent, Python provides the DB API. This is an API to standardize working with databases. When a package is written to access a database, it is written to conform to the API, and thus programmers do not have to learn a new set of methods and functions for each different database architecture.

### DB API objects and methods

```
conn = package.connect(server, db)
cursor = conn.cursor()
num_lines = cursor.execute(query)
num_lines = cursor.execute(query-with-placeholders, param-iterable)
num_lines = cursor.executemany(query-with-placeholders, nested-param-iterable)
all_rows = cursor.fetchall()
some_rows = cursor.fetchmany(n)
one_row = cursor.fetchone()
conn.commit()
conn.rollback()
```

Table 32. Available Interfaces (using Python DB API-2.0)

Database	Python package
Firebird (and Interbase)	KInterbasDB
IBM DB2	ibm-db
Informix	informixdb
Ingres	ingmod
Microsoft SQL Server	pymssql
MySQL	pymysql
ODBC	pyodbc
Oracle	cx_oracle
PostgreSQL	psycopg ( <i>previously</i> psycopg2)
SAP DB (also known as "MaxDB")	sapdbapi
SQLite	sqlite3
Sybase	Sybase



This list is not comprehensive, and there are additional interfaces available for some of the listed DBMSs.

## Connecting to a Server

- Import appropriate library
- Use `connect()` to get a connection object
- Specify host, database, username, password, etc.

To connect to a database server, import the package for the specific database. Use the package's `connect()` method to get a connection object. The `connect()` function requires the information needed to access the database, which may include the host, initial database, username, or password.

Argument names for the `connect()` method are not consistent across packages. Most `connect()` methods use individual arguments, such as **host**, **database**, etc., but some use a single string argument.

When finished with the connection, call the `close()` method on the connection object. Many database modules support the context manager (`with` statement), and will automatically close the database when the `with` block is exited. Check the documentation to see how this is implemented for a particular database.

### Example

```
import pymysql

conn = pymysql.connect (host = "dbserver",
                        user = "adeveloper",
                        passwd = "s3cr3t",
                        db = "samples")

# Interact with database here ...


conn.close()
```

```
import sqlite3

with sqlite3.connect('sample.db') as conn:
    # Interact with database here ...
```

Table 33. connect() examples

Database	Python package	Connection
<b>IBM DB2</b>	ibm-db	<pre>import ibm_db_dbi as db2 conn = db2.connect(  "DATABASE=testdb;HOSTNAME=localhost;PORT=50000;PROTOCOL=TCPIP;UI D=db2inst1;PWD=scripts;",     "",     "" ) </pre>
<b>Oracle</b>	cx_oracle	<pre>ip = 'localhost' port = 1521 SID = 'YOURSIDHERE' dsn_tns = cx_Oracle.makedsn(ip, port, SID) db = cx_Oracle.connect('adeveloper', '\$3cr3t', dsn_tns) </pre>
<b>PostgreSQL</b>	psycopg2	<div> <pre>psycopg2.connect (''     host='localhost'     user='adeveloper'     password='\$3cr3t'     dbname='testdb'     '') </pre> </div> <div>  <b>connect()</b> has one <b>str</b> parameter, not multiple parameters </div>
<b>MS-SQL</b>	pymssql	<pre>pymssql.connect (     host="localhost",     user="adeveloper",     passwd="\$3cr3t",     db="testdb", ) pymssql.connect (     dsn="DSN", ) </pre>
<b>MySQL</b>	pymysql	<pre>pymysql.connect (     host="localhost",     user="adeveloper",     passwd="\$3cr3t",     db="testdb", ) </pre>

Database	Python package	Connection
<b>ODBC-compliant DB</b>	pyodbc	<pre> pyodbc.connect('''     DRIVER={SQL Server};     SERVER=localhost;     DATABASE=testdb;     UID=adeveloper;     PWD=\$3cr3t ''')  pyodbc.connect('DSN=testdsn;PWD=\$3cr3t') </pre> <div>  <p><code>connect()</code> has one (string) parameter, not multiple parameters</p> </div>
<b>SQLite3</b>	sqlite3	<pre> sqlite3.connect('testdb')  # on-disk database(single file) sqlite3.connect(':memory:') # in-memory database </pre>

## Creating a Cursor

- Cursor can execute SQL statements
- Create with `cursor()` method
- Multiple cursors available
  - Standard cursor
    - Returns rows as tuples
  - Other cursors
    - Return dictionary
    - Return hybrid dictionary/list
    - Leave data on server

Once you have a connection object, call `cursor()` to create a cursor object. A cursor is an object that can execute SQL code and fetch results. Each connection may have one or more active cursors.

The default cursor for most packages returns each row as a tuple of values. There are optional cursors that can return data in different formats, or that control whether data is stored on the client or the server.



The examples in this chapter are implemented with SQLite, since the `sqlite3` module is part of the standard library. Most of the examples are also implemented for PostgreSQL and MySQL. See `db_mysql_*.py` and `db_postgres_*.py` in EXAMPLES.

### Example

```
import sqlite3
conn = sqlite3.connect("sample.db")
cursor = conn.cursor()
```

## Querying data

- **`cursor.execute(query)`**
  - Gets all data from query
  - Returns # rows in result set
- Use **`fetch...`** methods
  - `.fetchall()`
  - `.fetchone()`
  - `.fetchmany()`
- Return rows as tuples of values

Once you have a cursor, you can use it to execute queries via the `execute()` method. The first argument to `execute()` is a string containing one SQL statement.

For queries, `execute()` returns the number of rows in the result set. For non-query statements, `execute()` returns the number of rows affected by the operation.



For `sqlite3`, `execute()` returns the cursor object, so you can say `execute(QUERY-STATEMENT).fetchall()`.

### Fetch methods

Cursors provide three methods for returning query results.

`fetchone()` returns the next available row from the query results.

`fetchall()` returns a tuple of all rows.

`fetchmany(n)` returns up to `n` rows. This is useful when the query returns a large number of rows.

For all three methods, each row is returned as a tuple of values.



For standard cursors, all data is transferred from the database server to your program's memory when `execute()` is called.



## Example

### db\_sqlite\_basics.py

```
import sqlite3

# conn = sqlite3.Connection(...)
with sqlite3.connect("../DATA/presidents.db") as conn: # connect to the database

    s3_cursor = conn.cursor() # get a cursor object

    # select specified columns from all presidents
    s3_cursor.execute('''
        select termnum, firstname, lastname, party
        from presidents
    ''') # execute a SQL statement

    for term, firstname, lastname, party in s3_cursor.fetchall():
        print(f"{term:2d} {firstname:25} {lastname:20} {party}")
    print()
```

***db\_sqlite\_basics.py***

1	George	Washington	no party
2	John	Adams	Federalist
3	Thomas	Jefferson	Democratic - Republican
4	James	Madison	Democratic - Republican
5	James	Monroe	Democratic - Republican
6	John Quincy	Adams	Democratic - Republican
7	Andrew	Jackson	Democratic
8	Martin	Van Buren	Democratic
9	William Henry	Harrison	Whig
10	John	Tyler	Whig
11	James Knox	Polk	Democratic
12	Zachary	Taylor	Whig

...

36	Lyndon Baines	Johnson	Democratic
37	Richard Milhous	Nixon	Republican
38	Gerald Rudolph	Ford	Republican
39	James Earl 'Jimmy'	Carter	Democratic
40	Ronald Wilson	Reagan	Republican
41	George Herbert Walker	Bush	Republican
42	William Jefferson 'Bill'	Clinton	Democratic
43	George Walker	Bush	Republican
44	Barack Hussein	Obama	Democratic
45	Donald J	Trump	Republican
46	Joseph Robinette	Biden	Democratic

## Non-query statements

- Update database
- Returns count of rows affected
- Changes must be committed

The `execute()` method is also used to execute non-query statements, such as **CREATE**, **ALTER**, **UPDATE**, and **DROP**.

As with queries, the first argument is a string containing one SQL statement.

For most DB packages, `execute()` returns the number of rows affected.

To make changes to the database permanent, changes must be committed with `CONNECTION.commit()`.

## Example

### db\_sqlite\_add\_row.py

```
from datetime import date
import sqlite3

with sqlite3.connect("../DATA/presidents.db") as s3conn: # connect to database

    sql_insert = """
    insert into presidents
    (termnum, lastname, firstname, termstart, termend, birthplace, birthstate, birthdate,
    deathdate, party)
    values (47, 'Ramirez', 'Mary', '2025-01-20', null, 'Topeka',
    'Kansas', '1968-09-22', null, 'Independent')
    """

    cursor = s3conn.cursor()

    try:
        cursor.execute(sql_insert)
    except (sqlite3.OperationalError, sqlite3.DatabaseError, sqlite3.DataError) as err:
        print(err)
        s3conn.rollback()
    else:
        s3conn.commit()

    cursor.close()
```

## Example

### db\_sqlite\_delete\_row.py

```
from datetime import date
import sqlite3

with sqlite3.connect("../DATA/presidents.db") as conn: # connect to DB

    sql_delete = """
    delete from presidents
    where TERMNUM = 47
    """

    cursor = conn.cursor() # get a cursor

    try:
        cursor.execute(sql_delete)
    except (sqlite3.DatabaseError, sqlite3.OperationalError, sqlite3.DataError) as err:
        print(err)
        conn.rollback()
    else:
        conn.commit()

    cursor.close()
```

## SQL Injection

- Hijacks SQL code
- Result of string formatting
- Always use parameterized statements

One kind of vulnerability in SQL code is called *SQL injection*. This happens when using string formatting and raw user input to build SQL statements. An attacker can embed malicious SQL commands in input data.

Since the programmer is generating the SQL code as a string, there is no way to check for malicious SQL code. It is best practice to use parameterized statements, which prevents any user input from being *injected* into the SQL statement.



see <http://www.xkcd.com/327> for a well-known web comic on this subject.

## Example

### db\_sql\_injection.py

```
#
good_input = 'Google'
malicious_input = "'; drop table customers; -- " # input would come from a web form, for
instance

naive_format = "select * from customers where company_name = '{} ' and company_id != 0"

good_query = naive_format.format(good_input) # string formatting naively adds the user
input to a field, expecting only a customer name
malicious_query = naive_format.format(malicious_input) # string formatting naively adds
the user input to a field, expecting only a customer name

print("Good query:")
print(good_query) # non-malicious input works fine
print()

print("Bad query:")
print(malicious_query) # query now drops a table ('--' is SQL comment)
```

### db\_sql\_injection.py

```
Good query:
select * from customers where company_name = 'Google' and company_id != 0

Bad query:
select * from customers where company_name = "'; drop table customers; -- ' and
company_id != 0
```

## Parameterized Statements

- Prevent SQL injection
- More efficient updates
- Use placeholders in query
  - Placeholders vary by DB
- Pass iterable of parameters
- Use `cursor.execute()` or `cursor.executemany()`

For efficiency, you can iterate over of sequence of input datasets when performing a non-query SQL statement. The `execute()` method takes a query, plus an iterable of values to fill in the placeholders. The database manager will only parse the query once, then reuse it for subsequent calls to `execute()`.

All SQL statements may be parameterized, including queries.

Parameterized statements also protect against SQL injection attacks.

Different database modules use different placeholders. To see what kind of placeholder a module uses, check `MODULE.paramstyle`. Types include *pyformat*, meaning `%s`, and *qmark*, meaning `?`.

The `executemany()` method takes a query, plus an iterable of iterables. It will call `execute()` once for each nested iterable.

Table 34. Placeholders for SQL Parameters

Python package	Placeholder
<code>pymysql</code>	<code>%s</code>
<code>cx_oracle</code>	<code>:param_name</code>
<code>pyodbc</code>	<code>?</code>
<code>pymssql</code>	<code>%d</code> for <code>int</code> , <code>%s</code> for <code>str</code> , etc.
<code>Psychopg</code>	<code>%s</code> or <code>%(param_name)s</code>
<code>SQLite</code>	<code>?</code> or <code>:param_name</code>



with the exception of **pymssql** the same placeholder is used for all column types.



## Example

### db\_sqlite\_parameterized.py

```
import sqlite3

TERMS_TO_UPDATE = [1, 5, 19, 22, 36]

PARTY_UPDATE = '''
update presidents
set party = "SURPRISE!"
where termnum = ?
''' # ? is SQLite3 placeholder for SQL statement parameter; different DBMSs use
different placeholders

PARTY_QUERY = """
select termnum, firstname, lastname, party
from presidents
where termnum = ?
"""

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    for termnum in TERMS_TO_UPDATE:
        s3cursor.execute(PARTY_UPDATE, [termnum]) # second argument to execute() is
        iterable of values to fill in placeholders from left to right

    s3conn.commit()

    for termnum in TERMS_TO_UPDATE:
        s3cursor.execute(PARTY_QUERY, [termnum])
        print(s3cursor.fetchone())
```

### db\_sqlite\_parameterized.py

```
(1, 'George', 'Washington', 'SURPRISE!')
(5, 'James', 'Monroe', 'SURPRISE!')
(19, 'Rutherford Birchard', 'Hayes', 'SURPRISE!')
(22, 'Grover', 'Cleveland', 'SURPRISE!')
(36, 'Lyndon Baines', 'Johnson', 'SURPRISE!')
```

## Example

### db\_sqlite\_restore\_parties.py

```
import sqlite3

RESTORE_DATA = [
    (1, 'no party'),
    (5, 'Democratic - Republican'),
    (19, 'Republican'),
    (22, 'Democratic'),
    (36, 'Democratic')
]

PARTY_UPDATE = '''
update presidents
set party = ?
where termnum = ?
''' # ? is SQLite3 placeholder; other DBMSs may use other placeholders

PARTY_QUERY = '''
select termnum, firstname, lastname, party
from presidents
where termnum = ?
'''

with sqlite3.connect("../DATA/presidents.db") as s3conn:
    s3cursor = s3conn.cursor()

    for termnum, party in RESTORE_DATA:
        s3cursor.execute(PARTY_UPDATE, [party, termnum]) # second argument to execute()
        # is iterable of values to fill in placeholders from left to right
    s3conn.commit()

    for termnum, _ in RESTORE_DATA:
        s3cursor.execute(PARTY_QUERY, [termnum])
        print(s3cursor.fetchone())
```

### db\_sqlite\_restore\_parties.py

```
(1, 'George', 'Washington', 'no party')
(5, 'James', 'Monroe', 'Democratic - Republican')
(19, 'Rutherford Birchard', 'Hayes', 'Republican')
(22, 'Grover', 'Cleveland', 'Democratic')
(36, 'Lyndon Baines', 'Johnson', 'Democratic')
```

## Example

### db\_sqlite\_bulk\_insert.py

```
import sqlite3
import os
import csv

DATA_FILE = '../DATA/fruit_data.csv'

DB_NAME = 'fruits.db'
DB_TABLE = 'fruits'

SQL_CREATE_TABLE = f"""
create table {DB_TABLE} (
    id integer primary key,
    name varchar(30),
    unit varchar(30),
    unitprice decimal(6, 2)
)
""" # SQL statement to create table

SQL_INSERT_ROW = f'''
insert into {DB_TABLE} (name, unit, unitprice) values (?, ?, ?)
''' # parameterized SQL statement to insert one record

SQL_SELECT_ALL = f"""
select name, unit, unitprice from {DB_TABLE}
"""

def main():
    """
    Program entry point.

    :return: None
    """
    conn, cursor = get_connection()
    create_database(cursor)
    populate_database(conn, cursor)
    read_database(cursor)

    cursor.close()
    conn.close()

def get_connection():
    """
```

Get a connection to the PRODUCE database

```
:return: SQLite3 connection object.
```

```
"""
```

```
if os.path.exists(DB_NAME):
```

```
    os.remove(DB_NAME) # remove existing database if it exists
```

```
conn = sqlite3.connect(DB_NAME) # connect to (new) database
```

```
cursor = conn.cursor()
```

```
return conn, cursor
```

```
def create_database(cursor):
```

```
    """
```

Create the fruit table

```
:param conn: The database connection
```

```
:return: None
```

```
"""
```

```
cursor.execute(SQL_CREATE_TABLE) # run SQL to create table
```

```
def populate_database(conn, cursor):
```

```
    """
```

Add rows to the fruit table

```
:param conn: The database connection
```

```
:return: None
```

```
"""
```

```
with open(DATA_FILE) as file_in:
```

```
    fruit_data = csv.reader(file_in, quoting=csv.QUOTE_NONNUMERIC)
```

```
    try:
```

```
        cursor.executemany(SQL_INSERT_ROW, fruit_data) # iterate over list of pairs
                                                         # and add each pair to
```

database

```
    except sqlite3.DatabaseError as err:
```

```
        print(err)
```

```
        conn.rollback()
```

```
    else:
```

```
        conn.commit() # commit the inserts; without this, no data would be saved
```

```
def read_database(cursor):
```

```
    cursor.execute(SQL_SELECT_ALL)
```

```
    for name, unit, unitprice in cursor.fetchall():
```

```
        print(f'{name:12s} {unitprice:5.2f}/{unit}')
```

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

***db\_sqlite\_bulk\_insert.py***

pomegranate	0.99/each
cherry	2.25/pound
apricot	3.49/pound
date	1.20/pound
apple	0.55/pound
lemon	0.69/each
kiwi	0.88/each
orange	0.49/each
lime	0.49/each
watermelon	4.50/each
guava	2.88/pound
papaya	1.79/pound
fig	2.29/pound
pear	1.10/pound
banana	0.65/pound

# Metadata

- **`cursor.description`** returns tuple of tuples
- Fields
  - `name`
  - `type_code`
  - `display_size`
  - `internal_size`
  - `precision`
  - `scale`
  - `null_ok`

Once a query has been executed, the cursor's `description` attribute is a tuple with metadata about the columns in the query. It contains one tuple for each column in the query, containing 7 values describing the column.

For instance, to get the names of the columns, you could say

```
names = [d[0] for d in cursor.description]
```

For non-query statements, `CURSOR.description` returns `None`.

The names are based on the query (with possible aliases), and not necessarily on the names in the table.



Not all of the fields will necessarily be populated. For instance, `sqlite3` only provides column names.

## Example

### db\_sqlite\_metadata.py

```
"""
    Provide metadata (tables and column names) for a Sqlite3 database
"""
from pprint import pprint
import sqlite3

DB_NAME = "../DATA/presidents.db"
TABLE_QUERY = '''select * from presidents where 1 == 2'''

def main():
    cursor = connect_to_db(DB_NAME)
    show_metadata(cursor)

def connect_to_db(database_file):
    with sqlite3.connect(database_file) as s3conn:
        return s3conn.cursor()

def show_metadata(cursor):
    cursor.execute(TABLE_QUERY)
    pprint(cursor.description)
    print()
    column_names = [column_data[0] for column_data in cursor.description]
    print(f"{column_names =}")

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

***db\_sqlite\_metadata.py***

```
((('termnum', None, None, None, None, None, None),
  ('lastname', None, None, None, None, None, None),
  ('firstname', None, None, None, None, None, None),
  ('termstart', None, None, None, None, None, None),
  ('termend', None, None, None, None, None, None),
  ('birthplace', None, None, None, None, None, None),
  ('birthstate', None, None, None, None, None, None),
  ('birthdate', None, None, None, None, None, None),
  ('deathdate', None, None, None, None, None, None),
  ('party', None, None, None, None, None, None))

column_names = ['termnum', 'lastname', 'firstname', 'termstart', 'termend', 'birthplace',
                'birthstate', 'birthdate', 'deathdate', 'party']
```



## Dictionary Cursors

- Indexed by column name
- Not standardized in the DB API

Some DB packages provide dictionary cursors, which return a dictionary for each row, instead of a tuple. The keys are the names of the columns, so columns can be accessed by name rather than position.

Each package that provides a dictionary cursor has its own way of creating a dictionary cursor, although they all work the same way.



The `sqlite3` package provides a `Row` cursor, which can be indexed by position or by column name.

Table 35. Builtin Dictionary Cursors

Package	How to get a dictionary cursor
pymssql	<pre>conn = pymssql.connect (... , as_dict=True) dcur = conn.cursor() all cursors will be dict cursors</pre>
psycopg <sup>12</sup>	<pre>import psycopg.extras conn = psycopg.connect(...) dcur = conn.cursor(cursor_factory=psycopg.extras.DictCursor) only this cursor will be a dict cursor</pre>
sqlite3 <sup>1</sup>	<pre>conn = sqlite3.connect (... ) dcur = conn.cursor() dcur.row_factory = sqlite3.Row only this cursor will be a dict cursor  conn = sqlite3.connect (... ) conn.row_factory = sqlite3.Row dcur = conn.cursor() all cursors will be dict cursors</pre>
pymysql <sup>1</sup>	<pre>import pymysql.cursors  conn = pymysql.connect(...) dcur = conn.cursor(pymysql.cursors.DictCursor) only this cursor will be a dict cursor  conn = pymysql.connect(... , cursorclass = pymysql.cursors.DictCursor ) all cursors will be dict cursors</pre>
cx_oracle	Not available — use <code>db_iterrows</code>
pyodbc	Not available — use <code>db_iterrows</code>
pgdb	Not available — use <code>db_iterrows</code>

<sup>1</sup> Cursor supports indexing by either key value (dict style) or integer position (list style), as well as iteration. <sup>2</sup> Also supports `RealDictCursor` which is an actual dictionary, and `NamedTupleCursor`, which is an actual `namedtuple`

## Example

### db\_sqlite\_dict\_cursor.py

```
import sqlite3

s3conn = sqlite3.connect("../DATA/presidents.db")
# uncomment to make _all_ cursors dictionary cursors
# conn.row_factory = sqlite3.Row

NAME_QUERY = '''
    select firstname, lastname
    from presidents
    where termnum < 5
'''

cur = s3conn.cursor()

# select first name, last name from all presidents
cur.execute(NAME_QUERY)

for row in cur.fetchall():
    print(row)
print('-' * 50)

dict_cursor = s3conn.cursor() # get a normal SQLite3 cursor

# make _this_ cursor a dictionary cursor
dict_cursor.row_factory = sqlite3.Row # set the row factory to be a Row object

# Row objects are dict/list hybrids -- row[name] or row[pos]

# select first name, last name from all presidents
dict_cursor.execute(NAME_QUERY)

for row in dict_cursor.fetchall():
    print(row['firstname'], row['lastname']) # index row by column name

print('-' * 50)
```

***db\_sqlite\_dict\_cursor.py***

```
('George', 'Washington')  
('John', 'Adams')  
('Thomas', 'Jefferson')  
('James', 'Madison')
```

```
-----  
George Washington  
John Adams  
Thomas Jefferson  
James Madison  
-----
```

## Generic alternate cursors

- Create generator function
  - Get column names from `cursor.description()`
  - For each row
    - Make object from column names and values
      - Dictionary
      - Named tuple
      - Dataclass

For database modules that don't provide a dictionary cursor, the `iterrows_asdict()` function described below can be used with a cursor from any DB API-compliant package.

The example uses the metadata from the cursor to get the column names, and forms a dictionary by zipping the column names with the column values. `db_iterrows` also provides `iterrows_asnamedtuple()`, which returns each row as a named tuple, and `iterrows_asdataclass()`, which returns each row as an instance of a custom dataclass.

The functions in `db_iterrows` return generator objects. When you loop over the generator object, each element is a dictionary, named tuple, or instance of a dataclass, depending on which function you called.

## Example

### db\_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
from collections import namedtuple
from dataclasses import make_dataclass

def get_column_names(cursor):
    return [desc[0] for desc in cursor.description]

def iterrows_asdict(cursor):
    '''Generate rows as dictionaries'''
    column_names = get_column_names(cursor)
    for row in cursor.fetchall():
        row_dict = dict(zip(column_names, row))
        yield row_dict

def iterrows_asnamedtuple(cursor):
    '''Generate rows as named tuples'''
    column_names = get_column_names(cursor)
    Row = namedtuple('Row', column_names)
    for row in cursor.fetchall():
        yield Row(*row)

def iterrows_asdataclass(cursor):
    '''Generate rows as dataclass instances'''
    column_names = get_column_names(cursor)
    Row = make_dataclass('row_tuple', column_names)

    for row in cursor.fetchall():
        yield Row(*row)
```

## Example

### db\_sqlite\_iterrows.py

```
"""
Generic functions that can be used with any DB API compliant
package.

To use, pass in a cursor after execute()-ing a
SQL query. Then iterate over the generator that is
returned
"""
import sqlite3
from db_iterrows import *

sql_select = """
SELECT firstname, lastname, party
FROM presidents
WHERE termnum > 39
"""

conn = sqlite3.connect("../DATA/presidents.db")

cursor = conn.cursor()

cursor.execute(sql_select)

for row in iterrows_asdict(cursor):
    print(row['firstname'], row['lastname'], row['party'])

print('-' * 60)

cursor.execute(sql_select)

for row in iterrows_asnamedtuple(cursor):
    print(row.firstname, row.lastname, row.party)

print('-' * 60)

cursor.execute(sql_select)

for row in iterrows_asdataclass(cursor):
    print(row.firstname, row.lastname, row.party)
```

***db\_sqlite\_iterrows.py***

```
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
```

```
-----
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
```

```
-----
Ronald Wilson Reagan Republican
George Herbert Walker Bush Republican
William Jefferson 'Bill' Clinton Democratic
George Walker Bush Republican
Barack Hussein Obama Democratic
Donald J Trump Republican
Joseph Robinette Biden Democratic
```



# Transactions

- Transactions allow safer control of updates
- `commit()` to make database changes permanent
- `rollback()` to discard changes

Sometimes a database task involves more than one change to your database (i.e., more than one SQL statement). You don't want the first SQL statement to succeed and the second to fail; this would leave your database in a corrupt state.

To be certain of data integrity, use **transactions**. This lets you make multiple changes to your database and only commit the changes if all the SQL statements were successful.

For all packages using the Python DB API, a transaction is started when you connect. At any point, you can call `CONNECTION.commit()` to save the changes, or `CONNECTION.rollback()` to discard the changes. If you don't call `commit()` after modifying a table, the data will not be saved.



You can also turn on *autocommit*, which calls `commit()` after every statement. See the table below for how autocommit is implemented in various DB packages. This is not considered a best practice.

Table 36. How to turn on autocommit

Package	Method/Attribute
cx_oracle	<code>conn.autocommit = True</code>
ibm_db_api	<code>conn.set_autocommit(True)</code>
pymysql	<code>pymysql.connect(..., autocommit=True)</code> or <code>conn.autocommit(True)</code>
psycopg	<code>conn.autocommit = True</code>
sqlite3	<code>sqlite3.connect(dbname, isolation_level=None)</code>



**pymysql** only supports transaction processing when using the **InnoDB** engine

## Example

```
try:
    for info in list_of_tuples:
        cursor.execute(query, info)
except SQLERROR:
    dbconn.rollback()
else:
    dbconn.commit()
```

# Object-relational Mappers

- No SQL required
- Maps a class to a table
- All DB work is done by manipulating objects
- Most popular Python ORMs
  - SQLAlchemy
  - Django (which is a complete web framework)

An Object-relational mapper is a module or framework that creates a level of abstraction above the actual database tables and SQL queries. As the name implies, a Python class (object) is mapped to the actual table.

The two most popular Python ORMs are SQLAlchemy which is a standalone ORM, and Django ORM. Django is a comprehensive Web development framework, which provides an ORM as a subpackage. SQLAlchemy is the most fully developed package, and is the ORM used by Flask and some other Web development frameworks.

Instead of querying the database, you call a search method on an object representing a table. To add a row to the table, you create a new instance of the table class, populate it, and call a method like `save()`. You can create a large, complex database system, complete with foreign keys, composite indices, and all the other attributes near and dear to a DBA, without writing the first line of SQL.

You can use Python ORMs in two ways.

One way is to design the database with the ORM. To do this, you create a class for each table in the database, specifying the columns with predefined classes from the ORM. Then you run an ORM command which executes the queries needed to build the database. If you need to make changes, you update the class definitions, and run an ORM command to synchronize the actual DBMS to your classes.

The second way is to map tables to an existing database. You create the classes to match the schemas that have already been defined in the database. Both SQLAlchemy and the Django ORM have tools to automate this process.

# NoSQL

- Non-relational database
- Document-oriented
- Can be hierarchical (nested)
- Examples
  - MongoDB
  - Cassandra
  - Redis

A current trend in data storage are called "NoSQL" or non-relational databases. These databases consist of *documents*, which are indexed, and may contain nested data.

NoSQL databases don't contain tables, and do not have relations.

While relational databases are great for tabular data, they are not as good a fit for nested data. Geo-spatial, engineering diagrams, and molecular modeling can have very complex structures. It is possible to shoehorn such data into a relational database, but a NoSQL database might work much better. Another advantage of NoSQL is that it can adapt to changing data structures, without having to rebuild tables if columns are added, deleted, or modified.

Some of the most common NoSQL database systems are MongoDB, Cassandra and Redis.

## Chapter 15 Exercises

### Exercise 15-1 (president\_sqlite.py, president\_main\_sqlite.py)

#### Part A (president\_sqlite.py)

For this exercise, use the SQLite database named `presidents.db` in the DATA folder. It has the following layout

Table 37. Layout of President Table

Field Name	SQLite Data Type	Python Data type	Null	Default
termnum	int(11)	int	YES	NULL
lastname	varchar(32)	str	YES	NULL
firstname	varchar(64)	str	YES	NULL
termstart	date	date	YES	NULL
termend	date	date	YES	NULL
birthplace	varchar(128)	str	YES	NULL
birthstate	varchar(32)	str	YES	NULL
birthdate	date	date	YES	NULL
deathdate	date	date	YES	NULL
party	varchar(32)	str	YES	NULL

Refactor the `president.py` module to get its data from this table, rather than from a file.



If you created a `president.py` module as part of an earlier lab, use that. Otherwise, use the supplied `president.py` module in the top folder of the lab files.

#### Part B (president\_main\_sqlite.py)

Modify `president_main.py` that used `president.py`. It should import the `President` class from this new module that uses the database instead of a text file, but otherwise work the same as before.

## Exercise 15-2 (add\_pres\_sqlite.py)

Add another president to the presidents database. Just make up the data for your new president.

SQL syntax for adding a record is

```
INSERT INTO table ("COL1-NAME",...) VALUES ("VALUE1",...)
```

To do a parameterized insert (the right way!):

```
INSERT INTO table ("COL1-NAME",...) VALUES (?, ?, ...)
```

# Chapter 16: Serializing Data

## Objectives

- Have a good understanding of the XML format
- Know which modules are available to process XML
- Use lxml ElementTree to create a new XML file
- Parse an existing XML file with ElementTree
- Using XPath for searching XML nodes
- Load JSON data from strings or files
- Write JSON data to strings or files
- Read and write CSV data
- Read and write YAML data

## Which XML module to use?

- Bewildering array of XML modules
- Some are SAX, some are DOM
- Use `xml.etree.ElementTree`

When you are ready to process Python with XML, you turn to the standard library, only to find a number of different modules with confusing names.

To cut to the chase, use **lxml.etree**, which is based on **ElementTree** with some nice extra features, such as pretty-printing. While not part of the core Python library, it is provided by the Anaconda bundle.

If **lxml.etree** is not available, you can use **xml.etree.ElementTree** from the core library.



## Getting Started With ElementTree

- Import `xml.etree.ElementTree` (or `lxml.etree`) as `ET` for convenience
- Parse XML or create empty `ElementTree`

`ElementTree` is part of the Python standard library; `lxml` is included with the Anaconda distribution.

Since putting "`xml.etree.ElementTree`" in front of its methods requires a lot of extra typing, it is typical to alias `xml.etree.ElementTree` to just `ET` when importing it: `import xml.etree.ElementTree as ET`

You can check the version of `ElementTree` via the `VERSION` attribute:

```
import xml.etree.ElementTree as ET
print(ET.VERSION)
```

## How ElementTree Works

- ElementTree contains root Element
- Document is tree of Elements

In ElementTree, an XML document consists of a nested tree of Element objects. Each Element corresponds to an XML tag.

An ElementTree object serves as a wrapper for reading or writing the XML text.

If you are parsing existing XML, use `ElementTree.parse()`; this creates the ElementTree wrapper and the tree of Elements. You can then navigate to, or search for, Elements within the tree. You can also insert and delete new elements.

If you are creating a new document from scratch, create a top-level (AKA "root") element, then create child elements as needed.

```
element = root.find('sometag')
for subelement in element:
    print(subelement.tag)
print(element.get('someattribute'))
```

# Elements

- Element has
  - Tag name
  - Attributes (implemented as a dictionary)
  - Text
  - Tail
  - Child elements (implemented as a list) (if any)
- SubElement creates child of Element

When creating a new Element, you can initialize it with the tag name and any attributes. Once created, you can add the text that will be contained within the element's tags, or add other attributes.

When you are ready to save the XML into a file, initialize an ElementTree with the root element.

The **Element** class is a hybrid of list and dictionary. You access child elements by treating it as a list. You access attributes by treating it as a dictionary. (But you can't use subscripts for the attributes – you must use the `get()` method).

The Element object also has several useful properties: **tag** is the element's tag; **text** is the text contained inside the element; **tail** is any text following the element, before the next element.

The **SubElement** class is a convenient way to add children to an existing Element.



Only the tag property of an Element is required; other properties are optional.

Table 38. Element methods and properties

Method/Property	Description
<code>append(element)</code>	Add a subelement element to end of subelements
<code>attrib</code>	Dictionary of element's attributes
<code>clear()</code>	Remove all subelements
<code>find(path)</code>	Find first subelement matching path
<code>findall(path)</code>	Find all subelements matching path
<code>findtext(path)</code>	Shortcut for <code>find(path).text</code>
<code>get(attr)</code>	Get an attribute; Shortcut for <code>attrib.get()</code>
<code>getiterator()</code>	Returns an iterator over all descendants
<code>getiterator(path)</code>	Returns an iterator over all descendants matching path
<code>insert(pos,element)</code>	Insert subelement element at position pos
<code>items()</code>	Get all attribute values; Shortcut for <code>attrib.items()</code>
<code>keys()</code>	Get all attribute names; Shortcut for <code>attrib.keys()</code>
<code>remove(element)</code>	Remove subelement element
<code>set(attrib,value)</code>	Set an attribute value; shortcut for <code>attr[attrib] = value</code>
<code>tag</code>	The element's tag
<code>tail</code>	Text following the element
<code>text</code>	Text contained within the element

Table 39. ElementTree methods and properties

Property	Description
<code>find(path)</code>	Finds the first toplevel element with given tag; shortcut for <code>getroot().find(path)</code> .
<code>findall(path)</code>	Finds all toplevel elements with the given tag; shortcut for <code>getroot().findall(path)</code> .
<code>findtext(path)</code>	Finds element text for first toplevel element with given tag; shortcut for <code>getroot().findtext(path)</code> .
<code>getiterator(path)</code>	Returns an iterator over all descendants of root node matching path. (All nodes if path not specified)
<code>getroot()</code>	Return the root node of the document
<code>parse(filename)</code> <code>parse(fileobj)</code>	Parse an XML source (filename or file-like object)
<code>write(filename,encoding)</code>	Writes XML document to filename, using encoding (Default us-ascii).

## Creating a New XML Document

- Create root element
- Add descendants via SubElement
- Use keyword arguments for attributes
- Add text after element created
- Create ElementTree for import/export

To create a new XML document, first create the root (top-level) element. This will be a container for all other elements in the tree. If your XML document contains books, for instance, the root document might use the "books" tag. It would contain one or more "book" elements, each of which might contain author, title, and ISBN elements.

Once the root element is created, use SubElement to add elements to the root element, and then nested Elements as needed. SubElement returns the new element, so you can assign the contents of the tag to the **text** attribute.

Once all the elements are in place, you can create an ElementTree object to contain the elements and allow you to write out the XML text. From the ElementTree object, call write.

To output an XML string from your elements, call ET.tostring(), passing the root of the element tree as a parameter. It will return a bytes object (pure ASCII), so use .decode() to convert it to a normal Python string.

For an example of creating an XML document from a data file, see **xml\_create\_knights.py** in the EXAMPLES folder

## Example

### xml\_create\_movies.py

```
# from xml.etree import ElementTree as ET
import lxml.etree as ET

movie_data = [
    ('Jaws', 'Spielberg, Stephen'),
    ('Vertigo', 'Alfred Hitchcock'),
    ('Blazing Saddles', 'Brooks, Mel'),
    ('Princess Bride', 'Reiner, Rob'),
    ('Avatar', 'Cameron, James'),
]

movies = ET.Element('movies')

for name, director in movie_data:
    movie = ET.SubElement(movies, 'movie', name=name)
    ET.SubElement(movie, 'director').text = director

print(ET.tostring(movies, pretty_print=True).decode())

doc = ET.ElementTree(movies)

doc.write('movies.xml')
```

***xml\_create\_movies.py***

```
<movies>
  <movie name="Jaws">
    <director>Spielberg, Stephen</director>
  </movie>
  <movie name="Vertigo">
    <director>Alfred Hitchcock</director>
  </movie>
  <movie name="Blazing Saddles">
    <director>Brooks, Mel</director>
  </movie>
  <movie name="Princess Bride">
    <director>Reiner, Rob</director>
  </movie>
  <movie name="Avatar">
    <director>Cameron, James</director>
  </movie>
</movies>
```



## Parsing An XML Document

- Use `ElementTree.parse()`
- returns an `ElementTree` object
- Use `get*` or `find*` methods to select an element

Use the `parse()` method to parse an existing XML document. It returns an `ElementTree` object, from which you can find the root, or any other element within the document.

To get the root element, use the `getroot()` method.

### Example

```
import xml.etree.ElementTree as ET

doc = ET.parse('solar.xml')

root = doc.getroot()
```

## Navigating the XML Document

- Use `find()` or `findall()`
- Element is iterable of its children
- `findtext()` retrieves text from element

To find the first child element with a given tag, use `find(tag)`. This will return the first matching element. The `findtext(tag)` method is the same, but returns the text within the tag.

To get all child elements with a given tag, use the `findall(tag)` method, which returns a list of elements.

to see whether a node was found, say

```
if node is None:
```

but to check for existence of child elements, say

```
if len(node) > 0:
```

A node with no children tests as false because it is an empty list, but it is not None.



The `ElementTree` object also supports the `find()` and `findall()` methods of the `Element` object, searching from the root object.

## Example

### xml\_planets\_nav.py

```
'''Use etree navigation to extract planets from solar.xml'''
import lxml.etree as ET

def main():
    '''Program entry point'''
    doc = ET.parse('../DATA/solar.xml')

    solar_system = doc.getroot()

    print(solar_system)
    print()

    inner = solar_system.find('innerplanets')
    print('Inner:')

    for planet in inner:
        if planet.tag == 'planet':
            print('\t', planet.get("planetname", "NO NAME"))

    outer = solar_system.find('outerplanets')
    print('Outer:')

    for planet in outer:
        print('\t', planet.get("planetname"))

    plutoids = solar_system.find('dwarfplanets')
    print('Dwarf:')

    for planet in plutoids:
        print('\t', planet.get("planetname"))

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

***xml\_planets\_nav.py***

```
<Element solarsystem at 0x10fb05bc0>
```

```
Inner:
```

```
    Mercury
```

```
    Venus
```

```
    Earth
```

```
    Mars
```

```
Outer:
```

```
    Jupiter
```

```
    Saturn
```

```
    Uranus
```

```
    Neptune
```

```
Dwarf:
```

```
    Pluto
```

## Example

### **xml\_read\_movies.py**

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

movies_doc = ET.parse('movies.xml') # read and parse the XML file

movies = movies_doc.getroot() # get the root element (<movies>)

for movie in movies: # loop through children of root element
    movie_name = movie.get('name'), # get 'name' attribute of movie element
    movie_director = movie.findtext('director'), # get 'director' attribute of movie
    element
    print(f'{movie_name} by {movie_director}')
```

### **xml\_read\_movies.py**

```
('Jaws',) by ('Spielberg, Stephen',)
('Vertigo',) by ('Alfred Hitchcock',)
('Blazing Saddles',) by ('Brooks, Mel',)
('Princess Bride',) by ('Reiner, Rob',)
('Avatar',) by ('Cameron, James',)
```

## Using XPath

- Use simple XPath patterns Works with find\* methods

When a simple tag is specified, the find\* methods only search for subelements of the current element. For more flexible searching, the find\* methods work with simplified **XPath** patterns. To find all tags named *spam*, for instance, use `./spam`.

```
./movie  
presidents/president/name/last
```

### Example

#### xml\_planets\_xpath1.py

```
# import xml.etree.ElementTree as ET  
import lxml.etree as ET  
  
doc = ET.parse('../DATA/solar.xml') # parse XML file  
  
inner_nodes = doc.findall('innerplanets/planet') # find all elements (relative to root  
element) with tag "planet" under "innerplanets" element  
  
outer_nodes = doc.findall('outerplanets/planet') # find all elements with tag "planet"  
under "outerplanets" element  
  
print('Inner:')  
for planet in inner_nodes: # loop through search results  
    print('\t', planet.get("planetname")) # print "name" attribute of planet element  
  
print('Outer:')  
for planet in outer_nodes: # loop through search results  
    print('\t', planet.get("planetname")) # print "name" attribute of planet element
```

**xml\_planets\_xpath1.py**

```
Inner:
    Mercury
    Venus
    Earth
    Mars
Outer:
    Jupiter
    Saturn
    Uranus
    Neptune
```

**Example****xml\_planets\_xpath2.py**

```
# import xml.etree.ElementTree as ET
import lxml.etree as ET

doc = ET.parse('../DATA/solar.xml')

jupiter = doc.find('..//planet[@planetname="Jupiter"]')

if jupiter is not None:
    for moon in jupiter:
        print(moon.text) # grab attribute
```

**xml\_planets\_xpath2.py**

```
Metis
Adrastea
Amalthea
Thebe
Io
Europa
Ganymede
Callisto
Themisto
Himalia
Lysithea
Elara
```

Table 40. ElementTree XPath Summary

Syntax	Meaning
<code>tag</code>	Selects all child elements with the given tag. For example, “spam” selects all child elements named “spam”, “spam/egg” selects all grandchildren named “egg” in all child elements named “spam”. You can use universal names (“{url}local”) as tags.
<code>*</code>	Selects all child elements. For example, “*/egg” selects all grandchildren named “egg”.
<code>.</code>	Select the current node. This is mostly useful at the beginning of a path, to indicate that it’s a relative path.
<code>//</code>	Selects all subelements, on all levels beneath the current element (search the entire subtree). For example, “//egg” selects all “egg” elements in the entire tree.
<code>..</code>	Selects the parent element.
<code>[@attrib]</code>	Selects all elements that have the given attribute. For example, “//a[@href]” selects all “a” elements in the tree that has a “href” attribute.
<code>[@attrib=‘value’]</code>	Selects all elements for which the given attribute has the given value. For example, “//div[@class=‘sidebar’]” selects all “div” elements in the tree that has the class “sidebar”. In the current release, the value cannot contain quotes.
<code>parent_tag[child_tag]</code>	Selects all parent elements that has a child element named <i>child_tag</i> . In the current version, only a single tag can be used (i.e. only immediate children are supported). Parent tag can be <code>*</code> .



# About JSON

- Lightweight, human-friendly format for data
- Contains dictionaries and lists
- Stands for JavaScript Object Notation
- Looks like Python
- Basic types: Number, String, Boolean, Array, Object
- White space is ignored
- Stricter rules than Python

JSON is a lightweight and human-friendly format for sharing or storing data. It was developed and popularized by Douglas Crockford starting in 2001.

A JSON file contains objects and arrays, which correspond exactly to Python dictionaries and lists.

White space is ignored, so JSON may be formatted for readability.

Data types are Number, String, and Boolean. Strings are enclosed in double quotes (only); numbers look like integers or floats; Booleans are represented by true or false; null (None in Python) is represented by null.

## Reading JSON

- json module in standard library
- json.load() parse from file-like object
- json.loads() parse from string
- Both methods return Python dict or list

To read a JSON file, import the json module. Use json.loads() to parse a string containing valid JSON. Use json.load() to read JSON from a file-like object.

Both methods return a Python dictionary containing all the data from the JSON file.

## Example

### json\_read.py

```
from pprint import pprint
import json

# json.loads(String)      load from string
# json.load(FILE_OBJECT) load from file-like object

with open('../DATA/solar.json') as solar_in: # open JSON file for reading
    solar = json.load(solar_in) # load from file object and convert to Python data
    structure

# uncomment to see raw Python data
# print('-' * 60)
# pprint(solar)
# print('-' * 60)
# print('\n\n')

print(solar['innerplanets']) # solar is just a Python dictionary
print('*' * 60)
print(solar['innerplanets'][0]['name'])
print('*' * 60)
for planet in solar['innerplanets'] + solar['outerplanets']:
    print(planet['name'])

print('*' * 60)
for group in solar:
    if group.endswith('planets'):
        for planet in solar[group]:
            print(planet['name'])
```

*json\_read.py*

```
[{'name': 'Mercury', 'moons': None}, {'name': 'Venus', 'moons': None}, {'name': 'Earth',  
'moons': ['Moon']}, {'name': 'Mars', 'moons': ['Deimos', 'Phobos']}]
```

```
*****
```

```
Mercury
```

```
*****
```

```
Mercury
```

```
Venus
```

```
Earth
```

```
Mars
```

```
Jupiter
```

```
Saturn
```

```
Uranus
```

```
Neptune
```

```
*****
```

```
Mercury
```

```
Venus
```

```
Earth
```

```
Mars
```

```
Jupiter
```

```
Saturn
```

```
Uranus
```

```
Neptune
```

```
Pluto
```

# Writing JSON

- Use `json.dumps()` or `json.dump()`

To output JSON to a string, use `json.dumps()`. To output JSON to a file, pass a file-like object to `json.dump()`. In both cases, pass a Python data structure as the data to be output.

## Example

### `json_write.py`

```
import json

george = [
    {
        'num': 1,
        'lname': 'Washington',
        'fname': 'George',
        'dstart': [1789, 4, 30],
        'dend': [1797, 3, 4],
        'birthplace': 'Westmoreland County',
        'birthstate': 'Virginia',
        'dbirth': [1732, 2, 22],
        'ddeath': [1799, 12, 14],
        'assassinated': False,
        'party': None,
    },
    {
        'spam': 'ham',
        'eggs': [1.2, 2.3, 3.4],
        'toast': {'a': 5, 'm': 9, 'c': 4},
    }
] # Python data structure

js = json.dumps(george, indent=4) # dump structure to JSON string
print(js)

with open('george.json', 'w') as george_out: # open file for writing
    json.dump(george, george_out, indent=4) # dump structure to JSON file using open
    file object
```

*json\_write.py*

```
[
    {
        "num": 1,
        "lname": "Washington",
        "fname": "George",
        "dstart": [
            1789,
            4,
            30
        ],
        "dend": [
            1797,
            3,
            4
        ],
        "birthplace": "Westmoreland County",
        "birthstate": "Virginia",
        "dbirth": [
            1732,
            2,
            22
        ],
        "ddeath": [
            1799,
            12,
            14
        ],
        "assassinated": false,
        "party": null
    },
    {
        "spam": "ham",
        "eggs": [
            1.2,
            2.3,
            3.4
        ],
        "toast": {
            "a": 5,
            "m": 9,
            "c": 4
        }
    }
]
```

## Customizing JSON

- JSON data types limited
- simple cases — dump dict
- create custom encoders

The JSON spec only supports a limited number of datatypes. If you try to dump a data structure contains dates, user-defined classes, or many other types, the json encoder will not be able to handle it.

You can a custom encoder for various data types. To do this, write a function that expects one Python object, and returns some object that JSON can parse, such as a string or dictionary. The function can be called anything. Specify the function with the **default** parameter to `json.dump()`.

The function should check the type of the object. If it is a type that needs special handling, return a JSON-friendly version, otherwise just return the original object.

Table 41. Python types that JSON can encode

Python	JSON
<code>dict</code>	object
<code>list</code>	array
<code>str</code>	string
<code>int</code>	number (int)
<code>float</code>	number (real)
<code>True</code>	true
<code>False</code>	false
<code>None</code>	null



see the file `json_custom singledispatch.py` in EXAMPLES for how to use the **singledispatch** decorator (in the **functools** module) to handle multiple data types.

## Example

### json\_custom\_encoding.py

```
import json
from datetime import date

class Parrot(): # sample user-defined class (not JSON-serializable)
    def __init__(self, name, color):
        self._name = name
        self._color = color

    @property
    def name(self): # JSON does not understand arbitrary properties
        return self._name

    @property
    def color(self):
        return self._color

parrots = [ # list of Parrot objects
    Parrot('Polly', 'green'), #
    Parrot('Peggy', 'blue'),
    Parrot('Roger', 'red'),
]

def encode(obj): # custom JSON encoder function
    if isinstance(obj, date): # check for date object
        return obj.ctime() # convert date to string
    elif isinstance(obj, Parrot): # check for Parrot object
        return {'name': obj.name, 'color': obj.color} # convert Parrot to dictionary
    return obj # if not processed, return object for JSON to parse with default parser

data = { # dictionary of arbitrary data
    'spam': [1, 2, 3],
    'ham': ('a', 'b', 'c'),
    'toast': date(2014, 8, 1),
    'parrots': parrots,
}

# convert Python data to JSON data;
# 'default' parameter specifies function for custom encoding;
# 'indent' parameter says to indent and add newlines for readability
print(json.dumps(data, default=encode, indent=4))
```



***json\_custom\_encoding.py***

```
{
    "spam": [
        1,
        2,
        3
    ],
    "ham": [
        "a",
        "b",
        "c"
    ],
    "toast": "Fri Aug  1 00:00:00 2014",
    "parrots": [
        {
            "name": "Polly",
            "color": "green"
        },
        {
            "name": "Peggy",
            "color": "blue"
        },
        {
            "name": "Roger",
            "color": "red"
        }
    ]
}
```

## Reading and writing YAML

- yaml module from PYPI
- syntax like **json** module
- `yaml.load()`, `dump()` parse from/to file-like object
- `yaml.loads()`, `dumps()` parse from/to string

YAML is a structured data format which is a superset of JSON. However, YAML allows for a more compact and readable format.

Reading and writing YAML uses the same syntax as JSON, other than using the **yaml** module, which is NOT in the standard library. To install the **yaml** module:

```
pip install pyyaml
```

To read a YAML file (or string) into a Python data structure, use `yaml.load(__file_object__)` or `yaml.loads(__string__)`.

To write a data structure to a YAML file or string, use `yaml.dump(__data__, __file_object__)` or `yaml.dumps(__data__)`.

You can also write custom YAML processors.



YAML parsers will parse JSON data

## Example

### yaml\_read\_solar.py

```
import yaml

PLANET_SECTIONS = "inner outer plutoid".split()

with open('../DATA/solar.yaml') as solar_in:
    solar_data = yaml.load(solar_in, Loader=yaml.FullLoader)

star = solar_data['star']
print(f"Our star is {star}\n")

for section in PLANET_SECTIONS:
    for planet in solar_data[section]:
        print(planet['name'])
        for moon in planet['moons']:
            print(f"    {moon}")
```

***yaml\_read\_solar.py***

Our star is Sun

Mercury

None

Venus

None

Earth

Moon

Mars

Deimos

Phobos

Metis

Jupiter

Adrastea

Amalthea

Thebe

Io

Europa

Ganymede

Callisto

Themisto

Himalia

Lysithea

Elara

Saturn

Rhea

Hyperion

Titan

Iapetus

Mimas

...

## Example

### yaml\_create\_file.py

```
import sys
from datetime import date
import yaml

potus = {
    'presidents': [
        {
            'lastname': 'Washington',
            'firstname': 'George',
            'dob': date(1732, 2, 22),
            'dod': date(1799, 12, 14),
            'birthplace': 'Westmoreland County',
            'birthstate': 'Virginia',
            'term': [ date(1789, 4, 30), date(1797, 3, 4) ],
            'assassinated': False,
            'party': None,
        },
        {
            'lastname': 'Adams',
            'firstname': 'John',
            'dob': date(1735, 10, 30),
            'dod': date(1826, 7, 4),
            'birthplace': 'Braintree, Norfolk',
            'birthstate': 'Massachusetts',
            'term': [date(1797, 3, 4), date(1801, 3, 4)],
            'assassinated': False,
            'party': 'Federalist',
        }
    ]
}

with open('potus.yaml', 'w') as potus_out:
    yaml.dump(potus, potus_out)

yaml.dump(potus, sys.stdout)
```

***yaml\_create\_file.py***

```
presidents:
- assassinated: false
  birthplace: Westmoreland County
  birthstate: Virginia
  dob: 1732-02-22
  dod: 1799-12-14
  firstname: George
  lastname: Washington
  party: null
  term:
    - 1789-04-30
    - 1797-03-04
- assassinated: false
  birthplace: Braintree, Norfolk
  birthstate: Massachusetts
  dob: 1735-10-30
  dod: 1826-07-04
  firstname: John
  lastname: Adams
  party: Federalist
  term:
    - 1797-03-04
    - 1801-03-04
```

## Reading CSV data

- Use `csv` module
- Create a reader with file object or any iterable
- Iterate through reader to get rows as lists of columns

To read CSV data, create an instance of the `reader` class from the `csv` module. Pass in an iterable – typically, but not necessarily, a file object. (A file object is the object returned by `open()`).



You can pass in parameters to customize the input or output data.

### Example

#### `csv_read.py`

```
import csv

with open('../DATA/knights.csv') as knights_in:
    rdr = csv.reader(knights_in) # create CSV reader
    for name, title, color, quest, comment, number, ladies in rdr: # Read and unpack
        records one at a time; each record is a list
        print(f'{title:4s} {name:9s} {quest}')
```

#### `csv_read.py`

King	Arthur	The Grail
Sir	Lancelot	The Grail
Sir	Robin	Not Sure
Sir	Bedeveve	The Grail
Sir	Gawain	The Grail

...

## Customizing CSV readers and writers

- Variations in how CSV data is written
- Most common alternate is for Excel
- Add parameters to reader/writer

You can customize how the CSV parser and generator work by passing extra parameters to `csv.reader()` or `csv.writer()`. You can change the field and row delimiters, the escape character, and for output, what level of quoting. This can be used for any text file, not just CSV formats.

You can also specify a *dialect*, which is a custom set of CSV parameters. TO create a custom dialect, use `csv.register_dialect()`.

### Example

#### csv\_dialects.py

```
import csv

csv.register_dialect('colon-sep', delimiter=":")

with open('../DATA/knights.txt') as knights_in:
    reader = csv.reader(knights_in, dialect="colon-sep")
    for row in reader:
        print(row)
print()

with open('../DATA/primeministers.txt') as pm_in:
    reader = csv.reader(pm_in, dialect="colon-sep")
    for row in reader:
        print(row)
```



**csv\_dialects.py**

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

    '1', 'Sir John A.', 'Macdonald', '1867-7-1', '1873-11-5', 'Glasgow, Scotland', '1867-07-01', '1873-11-05', 'Liberal-Conservative']
    '2', 'Alexander', 'Mackenzie', '1873-11-7', '1878-10-8', 'Logierait, Scotland', '1873-11-07', '1878-10-08', 'Liberal']
    '3', 'Sir John A.', 'Macdonald', '1878-10-17', '1891-6-6', 'Glasgow, Scotland', '1878-10-17', '1891-06-06', 'Liberal-Conservative']
    '4', 'Sir John', 'Albott', '1891-6-16', '1892-11-24', "Saint-Andre-d'Argenteuil, Quebec", '1891-06-16', '1892-11-24', 'Liberal-Conservative']
    '5', 'Thompson', 'Sir John', '1892-12-5', '1894-12-12', 'Halifax, Nova Scotia', '1892-12-05', '1894-12-12', 'Conservative']
    '6', 'Sir Mackenzie', 'Bowell', '1894-12-21', '1896-4-27', 'Rickingham, England', '1894-12-21', '1896-04-27', 'Conservative']
    '7', 'Sir Charles', 'Tupper', '1896-5-1', '1896-7-8', 'Amherst, Nova Scotia', '1896-05-01', '1896-07-08', 'Conservative']
    '8', 'Sir Wilfred', 'Laurier', '1896-7-11', '1911-10-6', 'Saint-Lin-Laurentides, Quebec', '1886-07-11', '1911-10-06', 'Liberal']
    '9', 'Sir Robert', 'Borden', '1911-10-10', '1917-10-12', 'Grand-Pre, Nova Scotia', '1911-10-10', '1917-10-11', 'Conservative']
    '10', 'Sir Robert', 'Borden', '1917-10-12', '1920-7-10', 'Grand-Pre, Nova Scotia', '1917-10-12', '1920-07-10', 'Unionist']
    '11', 'Arthur', 'Meighen', '1920-7-10', '1921-12-29', 'Perth South, Ontario', '1920-07-10', '1921-12-29', 'NLC']
    '12', 'William Lyon Mackenzie', 'King', '1921-12-29', '1926-6-29', 'Kitchener, Ontario', '1921-12-29', '1926-06-28', 'Liberal']
    '13', 'Arthur', 'Meighen', '1926-6-29', '1926-9-25', 'Perth South, Ontario', '1926-06-29', '1926-09-25', 'Conservative']
```

## Example

### csv\_nonstandard.py

```
import csv

with open('../DATA/computer_people.txt') as computer_people_in:
    rdr = csv.reader(computer_people_in, delimiter=';') # specify alternate field
    delimiter

    # iterate over rows of data -- csv reader is an iterator

    for first_name, last_name, known_for, birth_date in rdr:
        print(f'{last_name}: {known_for}')
```

### csv\_nonstandard.py

```
Gates: Gates Foundation
Jobs: Apple
Wall: Perl
Allen: Microsoft
Ellison: Oracle
van Rossum: Python
Kurtz: BASIC
Hopper: COBOL
Gates: Microsoft
Zuckerberg: Facebook
Brin: Google
van Rossum: Python
Lovelace:
Page: Google
Torvalds: Linux
```

Table 42. CSV reader/writer Parameters

Parameter	Meaning
<code>quotechar</code>	One-character string to use as quoting character (default: <code>'</code> )
<code>delimiter</code>	One-character string to use as field separator (default: <code>,</code> )
<code>skipinitialspace</code>	If True, skip white space after field separator (default: <code>False</code> )
<code>lineterminator</code>	The character sequence which terminates rows (default: depends on OS)
<code>quoting</code>	When should quotes be generated when writing CSV <code>csv.QUOTE_MINIMAL</code> – only when needed (default) <code>csv.QUOTE_ALL</code> – quote all fields <code>csv.QUOTE_NONNUMERIC</code> – quote all fields that are not numbers <code>csv.QUOTE_NONE</code> – never put quotes around fields
<code>escapechar</code>	One-character string to escape delimiter when quoting is set to <code>csv.QUOTE_NONE</code>
<code>doublequote</code>	Control quote handling inside fields. When <code>True</code> , two consecutive quotes are read as one, and one quote is written as two. (default: <code>True</code> )
<code>dialect</code>	string representing registered dialect name, such as "excel"

## Using csv.DictReader

- Returns each row as dictionary
- Keys are field names
- Use header or specify

Instead of the normal reader, you can create a dictionary-based reader by using the DictReader class.

If the CSV file has a header, it will parse the header line and use it as the field names. Otherwise, you can specify a list of field names with the **fieldnames** parameter. For each row, you can look up a field by name, rather than position.

### Example

#### csv\_dictreader.py

```
import csv

field_names = ['term', 'firstname', 'lastname', 'birthplace', 'state', 'party'] # field
names, which will become dictionary keys on each row

with open('../DATA/presidents.csv') as presidents_in:
    rdr = csv.DictReader(presidents_in, fieldnames=field_names) # create reader, passing
    in field names (if not specified, uses first row as field names)
    for row in rdr: # iterate over rows in file
        print(f"{row['firstname']:25} {row['lastname']:12} {row['party']}")
```

**csv\_dictreader.py**

George	Washington	no party
John	Adams	Federalist
Thomas	Jefferson	Democratic - Republican
James	Madison	Democratic - Republican
James	Monroe	Democratic - Republican
John Quincy	Adams	Democratic - Republican
Andrew	Jackson	Democratic
Martin	Van Buren	Democratic
William Henry	Harrison	Whig
John	Tyler	Whig
James Knox	Polk	Democratic
Zachary	Taylor	Whig
Millard	Fillmore	Whig
Franklin	Pierce	Democratic
James	Buchanan	Democratic
Abraham	Lincoln	Republican
Andrew	Johnson	Republican
Ulysses Simpson	Grant	Republican
Rutherford Birchard	Hayes	Republican
James Abram	Garfield	Republican

...

## Writing CSV Data

- Use `csv.writer()`
- Parameter is file-like object (must implement `write()` method)
- Can specify parameters to writer constructor
- Use `writerow()` or `writerows()` to output CSV data

To output data in CSV format, first create a writer using `csv.writer()`. Pass in a file-like object.

For each row to write, call the `writerow()` method of the writer, passing in an iterable with the values for that row.

To modify how data is written out, pass parameters to the writer.



On Windows, to prevent double-spaced output, add `lineterminator='\n'` when creating a CSV writer.

## Example

### csv\_write.py

```
import sys
import csv

chicago_data = [
    ['Name', 'Position Title', 'Department', 'Employee Annual Salary'],
    ['BONADUCE, MICHAEL J', 'POLICE OFFICER', 'POLICE', '$80724.00'],
    ['MELLON, MATTHEW J "Matt"', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FIERI, JOHN J', 'FIREFIGHTER-EMT', 'FIRE', '$75342.00'],
    ['GALAHAD, MERLE S', 'CLERK III', 'BUSINESS AFFAIRS', '$45828.00'],
    ['ORCATTI, JENNIFER L', 'FIRE COMMUNICATIONS OPERATOR I', 'OEMC', '$63121.68'],
    ['ASHE, JOHN W', 'FOREMAN OF MACHINISTS', 'AVIATION', '$96553.60'],
    ['SADINSKY BLAKE, MICHAEL G', 'POLICE OFFICER', 'POLICE', '$78012.00'],
    ['GRANT, CRAIG A', 'SANITATION LABORER', 'STREETS & SAN', '$69576.00'],
    ['MILLER, JONATHAN D', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['FRANK, ARTHUR R', 'POLICE OFFICER/EXPLSV DETECT, K9 HNDLR', 'POLICE', '$87918.00'],
    ['POVOTTI, JAMES S "Jimmy P"', 'TRAFFIC CONTROL AIDE-HOURLY', 'OEMC', '$19167.20'],
    ['TRAWLER, DANIEL J', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SCUBA, ANDREW G', 'POLICE OFFICER', 'POLICE', '$75372.00'],
    ['SWINE, MATTHEW W', 'SERGEANT', 'POLICE', '$99756.00'],
    ['"RYDER, MYRTA T "Lil Myrt"', 'POLICE OFFICER', 'POLICE', '$83706.00'],
    ['KORSHAK, ROMAN', 'PARAMEDIC', 'FIRE', '$75372.00']
]

with open('../TEMP/chi_data.csv', 'w') as chi_out:
    # On Windows, output line terminator must be set to '\n'.
    # While it's not needed on Linux/Mac, it doesn't cause any problems,
    # so this keeps the code portable.
    wtr = csv.writer(chi_out, lineterminator='\n') # create CSV writer from file object
    that is opened
    for data_row in chicago_data: # iterate over records from file
        data_row[0] = data_row[0].title() # make first field title case rather than all
        uppercase
        data_row[-1] = data_row[-1].rstrip('$') # strip leading $ from last field
        wtr.writerow(data_row) # write one row (of iterables) to output file
```

# Pickle

- Use the pickle module
- Create a binary stream that can be saved to file
- Can also be transmitted over the network

Python uses the pickle module for data serialization. This format is unique to Python, and a binary format.

The extension for pickle files is ".pkl", or sometimes ".pickle".

To create pickled data, use either `pickle.dump()` or `pickle.dumps()`. Both functions take a data structure as the first argument. `dumps()` returns the pickled data as a string. `dump()` writes the data to a file-like object which has been specified as the second argument. The file-like object must be opened for writing.

To read pickled data, use `pickle.load()`, which takes a file-like object that has been open for writing, or `pickle.loads()` which reads from a string. Both functions return the original data structure that had been pickled.



The syntax of the **json** module is based on the **pickle** module.



## Example

### pickling.py

```
import pickle
from pprint import pprint

# some data structures
airports = {
    'RDU': 'Raleigh-Durham', 'IAD': 'Dulles', 'MGW': 'Morgantown',
    'EWR': 'Newark', 'LAX': 'Los Angeles', 'ORD': 'Chicago'
}

colors = [
    'red', 'blue', 'green', 'yellow', 'black',
    'white', 'orange', 'brown', 'purple'
]

values = [
    3/7, 1/9, 14.5
]

data = [ # list of data structures
    colors,
    airports,
    values,
]

print("BEFORE:")
pprint(data)
print('-' * 60)

with open('../TEMP/pickled_data.pkl', 'wb') as pkl_out: # open pickle file for writing
    in binary mode
    pickle.dump(data, pkl_out) # serialize data structures to pickle file

with open('../TEMP/pickled_data.pkl', 'rb') as pkl_in: # open pickle file for reading in
    in binary mode
    pickled_data = pickle.load(pkl_in) # de-serialize pickle file back into data
    structures

print("AFTER:")
pprint(pickled_data) # view data structures
```

***pickling.py***

```
BEFORE:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
{'EWR': 'Newark',
 'IAD': 'Dulles',
 'LAX': 'Los Angeles',
 'MGW': 'Morgantown',
 'ORD': 'Chicago',
 'RDU': 'Raleigh-Durham'}],
[0.42857142857142855, 0.1111111111111111, 14.5]]
```

```
-----
AFTER:
[['red',
  'blue',
  'green',
  'yellow',
  'black',
  'white',
  'orange',
  'brown',
  'purple'],
{'EWR': 'Newark',
 'IAD': 'Dulles',
 'LAX': 'Los Angeles',
 'MGW': 'Morgantown',
 'ORD': 'Chicago',
 'RDU': 'Raleigh-Durham'}],
[0.42857142857142855, 0.1111111111111111, 14.5]]
```

## Chapter 16 Exercises

### Exercise 16-1 (xwords.py)

Using ElementTree, create a new XML file containing all the words that start with x from words.txt. The root tag should be named *words*, and each word should be contained in a *word* tag. The finished file should look like this:

```
<words>
  <word>xanthan</word>
  <word>xanthans</word>
  and so forth
</words>
```

### Exercise 16-2 (xpresidents.py)

Use ElementTree to parse presidents.xml. Loop through and print out each president's first and last names and their state of birth.

### Exercise 16-3 (jpresidents.py)

Rewrite xpresidents.py to parse presidents.json using the json module.

### Exercise 16-4 (cpresidents.py)

Rewrite xpresidents.py to parse presidents.csv using the csv module.

### Exercise 16-5 (pickle\_potus.py)

Write a script which reads the data from presidents.csv into a dictionary where the key is the term number, and the value is another dictionary of data for one president.

Using the `pickle` module, Write the entire dictionary out to a file named `presidents.pkl`.

### Exercise 16-6 (unpickle\_potus.py)

Write a script to open presidents.pkl, and restore the data back into a dictionary.

Then loop through the array and print out each president's first name, last name, and party.

## Extra topics

The following topics will be covered at the discretion of the instructor if there is enough time.

# Chapter 17: Virtual Environments

## Objectives

- Learn what virtual environments are
- Understand why you should use them
- Implement virtual environments

## Why do we need virtual environments?

Consider the following scenario:

Mary creates an app using Python modules **spamlib** and **hamlib**. She builds a distribution package and gives it to Paul. He installs **spamlib** and **hamlib** on his computer, which has a compatible Python interpreter, and then installs Mary's app. It starts to run, but then crashes with errors. What happened?

In the meantime, since Mary created her app, the author of **spamlib** removed a function "that almost noone used". When Paul installed **spamlib**, he installed the latest version, which does not have the function. One possible fix is for Paul to revert to an older version of **spamlib**, but suppose he has another app that uses the newer version? This can get very messy very quickly.

The solution to this problem is a **virtual environment**.

## What are virtual environments?

A virtual environment starts with a snapshot (copy) of a plain Python installation, before any other libraries are added. It is used to isolate a particular set of modules that will successfully run a given application.

Each application can have its own virtual environment, which ensures that it has the required versions of dependencies. Virtual environments do not have to be in the same folder or folder tree as projects that use them, and multiple projects can use the same virtual environment. However, it's best practice for each project to have its own.

There are many tools for creating and using virtual environments, but the primary ones are **pip** and the **venv** module.

## Preparing the virtual environment

Before creating a virtual environment, it is best to start with a "plain vanilla" Python installation of the desired release. You can use the base Python bundle from <https://www.python.org/downloads>, or the Anaconda bundle from <https://www.anaconda.com/distribution>.

You won't use this installation directly — it will be more of a reference installation.

## Creating the environment

The **venv** module provides the tools to create a new virtual environment. Basic usage is

```
python -m venv environment_name
```

This creates a virtual environment named *environment\_name* in the current directory. It is a copy of the required parts of the original installation.

The virtual environment does not need to be in the same location as your application. It is common to create a folder named *.envs* under your home folder to contain all your virtual environments.



## Activating the environment

To use a virtual environment, it must be **activated**. This means that it takes precedence over any other installed Python version. This is implemented by changing the PATH variable in your operating system's environment to point to the virtual copy.

**venv** will put the name of the environment in the terminal prompt.

### Activating on Windows

To **activate** the environment on a Windows system, run the **activate.bat** script in the **Scripts** folder of the environment.

```
environment_name\Scripts\activate
```

### Activating on non-Windows

To activate the environment on a Linux, Mac, or other Unix-like system, source the **activate** script in the **bin** folder of the environment. This must be *sourced* — run the script with the **source** builtin command, or the **.** shortcut

```
source environment_name/bin/activate  
or  
. environment_name/bin/activate
```

## Deactivating the environment

When you are finished with an environment, you can deactivate it with the **deactivate** command. It does not need the leading path.

**venv** will remove the name of the environment from the terminal prompt.

### Deactivating on Windows

Run the deactivate.bat script:

```
deactivate
```

### Deactivating on non-Windows

Run the deactivate shell script:

```
deactivate
```



To run an app with a particular environment, create a batch file or shell script that activates the environment, then runs the app with the environment's interpreter. When the script is finished, it should deactivate the environment.

## Freezing the environment

When ready to share your app, specify the dependencies by running the **pip freeze** command. This will create a list of all the modules you have added to the virtual environments, and with their current versions. Since the output normally goes to *stdout*, it is conventional to redirect the output to a file named **requirements.txt**.

```
pip freeze > requirements.txt
```

Now you can provide your requirements.txt file to anyone who plans to run your app, and they can create a Python environment with the same versions of all required modules.

## Duplicating an environment

When someone sends you an app with a requirements.txt file, it is easy to reproduce their environments. Install Python, then use pip to add the modules from requirements.txt. pip has a `-r` option, which says to read a file for a list of modules to be installed.

```
pip install -r requirements.txt
```

That will add the module dependencies with the correct versions.

# The pipenv/conda/virtualenv/PyCharm swamp

There are more tools to help with virtual environment management, and having them all available can be confusing. You can always just use pip and venv, as described above.

Here are a few of these tools, and what they do

## conda

A tool provided by Anaconda that replaces both pip and venv. It assumes you have the Anaconda bundle installed.

## pipenv

Another tool that replaces both pip and venv. It is very convenient, but has some annoying issues.

## virtualenv

The original name of the **venv** module.

## PyCharm

A Python IDE that will create virtual environments for you. It does not come with Python itself.

## virtualenvwrapper

A workflow manager that makes it more convenient to switch from one environment to another.

## Chapter 17 Exercises

### Exercise 17-1

Create a virtual environment named **spam** and activate it.

Install the **roman** module.

Create a **requirements.txt** file.

Deactivate **spam**.

Create another virtual environment named **ham** and activate it.

Using **requirements.txt**, make the **ham** environment the same as **spam**.

# Chapter 18: Effective Scripts

## Objectives

- Launch external programs
- Check permissions on files
- Get system configuration information
- Store data offline
- Create Unix-style filters
- Parse command line options
- Configure application logging

## Using **glob**

- Expands wildcards
- Windows and non-windows
- Useful with **subprocess** module

When executing external programs, sometimes you want to specify a list of files using a wildcard. The **glob** function in the **glob** module will do this. Pass one string containing a wildcard (such as `*.txt`) to `glob()`, and it returns a sorted list of the matching files. If no files match, it returns an empty list.

### Example

#### **glob\_example.py**

```
from glob import glob

files = glob('../DATA/*.txt') # expand file name wildcard into sorted list of matching
names
print(files, '\n')

no_files = glob('../JUNK/*.avi')
print(no_files, '\n')
```



**glob\_example.py**

```
['../DATA/columns_of_numbers.txt', '../DATA/test_scores.txt', '../DATA/poe_sonnet.txt',  
 '../DATA/computer_people.txt', '../DATA/owl.txt', '../DATA/eggs.txt',  
 '../DATA/world_airport_codes.txt', '../DATA/stateinfo.txt', '../DATA/fruit2.txt',  
 '../DATA/us_airport_codes.txt', '../DATA/parrot.txt', '../DATA/http_status_codes.txt',  
 '../DATA/fruit1.txt', '../DATA/simple_entries.txt', '../DATA/upper_parrot.txt',  
 '../DATA/alice.txt', '../DATA/littlewomen.txt', '../DATA/spam.txt',  
 '../DATA/world_median_ages.txt', '../DATA/phone_numbers.txt',  
 '../DATA/sales_by_month.txt', '../DATA/engineers.txt', '../DATA/underrated.txt',  
 '../DATA/tolkien.txt', '../DATA/tyger.txt', '../DATA/example_data.txt',  
 '../DATA/states.txt', '../DATA/kjv.txt', '../DATA/fruit.txt', '../DATA/areacodes.txt',  
 '../DATA/float_values.txt', '../DATA/unabom.txt', '../DATA/chaos.txt',  
 '../DATA/noisewords.txt', '../DATA/presidents.txt', '../DATA/bible.txt',  
 '../DATA/breakfast.txt', '../DATA/Pride_and_Prejudice.txt', '../DATA/customerdata1.txt',  
 '../DATA/nsfw_words.txt', '../DATA/mary.txt',  
 '../DATA/2017FullMembersMontanaLegislators.txt', '../DATA/customerdata2.txt',  
 '../DATA/README.txt', '../DATA/words.txt', '../DATA/edgar_tickers.txt',  
 '../DATA/ncvoter32.txt', '../DATA/primeministers.txt',  
 '../DATA/nc_counties_avg_wage.txt', '../DATA/grail.txt', '../DATA/alt.txt',  
 '../DATA/knights.txt', '../DATA/world_airports_codes_raw.txt', '../DATA/overlord.txt',  
 '../DATA/correspondence.txt', '../DATA/ctemps.txt']
```

```
[]
```

## Using shlex.split()

- Splits string
- Preserves white space

If you have an external command you want to execute, you should split it into individual words. If your command has quoted whitespace, the normal **split()** method of a string won't work.

For this you can use **shlex.split()**, which preserves quoted whitespace within a string.

### Example

#### shlex\_split.py

```
#
import shlex

cmd = 'herp derp "fuzzy bear" "wanga tanga" pop' # Command line with quoted whitespace

print(cmd.split()) # Normal split does the wrong thing
print()

print(shlex.split(cmd)) # shlex.split() does the right thing
```

#### shlex\_split.py

```
['herp', 'derp', '"fuzzy', 'bear"', '"wanga', 'tanga"', 'pop']

['herp', 'derp', 'fuzzy bear', 'wanga tanga', 'pop']
```

# The subprocess module

- Spawns new processes
- works on Windows and non-Windows systems
- Convenience methods
  - `run()`
  - `call()`, `check_call()`

The **subprocess** module spawns and manages new processes. You can use this to run local non-Python programs, to log into remote systems, and generally to execute command lines.

subprocess implements a low-level class named `Popen`; However, the convenience methods `run()`, `check_call()`, and `check_output()`, **which are built on top of `Popen()`, are commonly used, as they have a simpler interface. You can capture `*stdout` and `stderr`, separately.** If you don't capture them, they will go to the console.

In all cases, you pass in an iterable containing the command split into individual words, including any file names. This is why this chapter starts with `glob.glob()` and `shlex.split()`.

Table 43. *CalledProcessError* attributes

Attribute	Description
<code>args</code>	The arguments used to launch the process. This may be a list or a string.
<code>returncode</code>	Exit status of the child process. Typically, an exit status of 0 indicates that it ran successfully. A negative value -N indicates that the child was terminated by signal N (POSIX only).
<code>stdout</code>	Captured stdout from the child process. A bytes sequence, or a string if <code>run()</code> was called with an encoding or errors. None if stdout was not captured. If you ran the process with <code>stderr=subprocess.STDOUT</code> , stdout and stderr will be combined in this attribute, and stderr will be None. stderr

## subprocess convenience functions

- `run()`, `check_call()`, `check_output()`
- Simpler to use than `Popen`

**subprocess** defines convenience functions, **`call()`**, **`check_call()`**, and **`check_output()`**.

```
proc subprocess.run(cmd, ...)
```

Run command with arguments. Wait for command to complete, then return a **CompletedProcess** instance.

```
subprocess.check_call(cmd, ...)
```

Run command with arguments. Wait for command to complete. If the exit code was zero then return, otherwise raise `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute.

```
check_output(cmd, ...)
```

Run command with arguments and return its output as a byte string. If the exit code was non-zero it raises a `CalledProcessError`. The `CalledProcessError` object will have the return code in the `returncode` attribute and output in the `output` attribute.



`run()` is only implemented in Python 3.5 and later.

## Example

### subprocess\_conv.py

```
import sys
from subprocess import check_call, check_output, CalledProcessError
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

try:
    check_call(full_cmd)
except CalledProcessError as err:
    print("Command failed with return code", err.returncode)

print('-' * 60)

try:
    output = check_output(full_cmd)
    print("Output:", output.decode(), sep='\n')
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

**subprocess\_conv.py**

```
-rw-r--r-- 1 jstrick staff 3178541 Nov 2 2020 ../DATA/tate_data.zip
-rw-r--r-- 1 jstrick staff 869349 Apr 9 2022 ../DATA/taxis.csv
drwxr-xr-x 10 jstrick staff 320 Oct 9 2022 ../DATA/templates
-rwxr-xr-x@ 1 jstrick staff 279 Sep 3 2022 ../DATA/test_scores.txt
-rwxr-xr-x 1 jstrick staff 2198 Feb 14 2016 ../DATA/textfiles.zip
-rw-r--r-- 1 jstrick staff 9729 Apr 7 2022 ../DATA/tips.csv
-rw-r--r-- 1 jstrick staff 57018 Apr 8 2022 ../DATA/titanic.csv
-rw-r--r-- 1 jstrick staff 106960 Jul 26 2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick staff 284160 Jul 26 2017 ../DATA/titanic3.xls
-rwxr-xr-x 1 jstrick staff 73808 Feb 14 2016 ../DATA/tolkien.txt
-rw-r--r--@ 1 jstrick staff 4461 Oct 19 2023 ../DATA/train.csv
-rw-r--r--@ 1 jstrick staff 32128 May 2 2022 ../DATA/tree_census.npy
-rwxr-xr-x 1 jstrick staff 834 Feb 14 2016 ../DATA/tyger.txt
```

**Output:**

```
-rw-r--r-- 1 jstrick staff 3178541 Nov 2 2020 ../DATA/tate_data.zip
-rw-r--r-- 1 jstrick staff 869349 Apr 9 2022 ../DATA/taxis.csv
drwxr-xr-x 10 jstrick staff 320 Oct 9 2022 ../DATA/templates
-rwxr-xr-x@ 1 jstrick staff 279 Sep 3 2022 ../DATA/test_scores.txt
-rwxr-xr-x 1 jstrick staff 2198 Feb 14 2016 ../DATA/textfiles.zip
-rw-r--r-- 1 jstrick staff 9729 Apr 7 2022 ../DATA/tips.csv
-rw-r--r-- 1 jstrick staff 57018 Apr 8 2022 ../DATA/titanic.csv
-rw-r--r-- 1 jstrick staff 106960 Jul 26 2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick staff 284160 Jul 26 2017 ../DATA/titanic3.xls
-rwxr-xr-x 1 jstrick staff 73808 Feb 14 2016 ../DATA/tolkien.txt
-rw-r--r--@ 1 jstrick staff 4461 Oct 19 2023 ../DATA/train.csv
-rw-r--r--@ 1 jstrick staff 32128 May 2 2022 ../DATA/tree_census.npy
-rwxr-xr-x 1 jstrick staff 834 Feb 14 2016 ../DATA/tyger.txt
```



showing Unix/Linux/Mac output – Windows will be similar



(Windows only) The following commands are *internal* to CMD.EXE, and must be preceded by **cmd /c** or they will not work: ASSOC, BREAK, CALL, CD/CHDIR, CLS, COLOR, COPY, DATE, DEL, DIR, DPATH, ECHO, ENDLOCAL, ERASE, EXIT, FOR, FTYPE, GOTO, IF, KEYS, MD/MKDIR, MKLINK (vista and above), MOVE, PATH, PAUSE, POPD, PROMPT, PUSHD, REM, REN/RENAME, RD/RMDIR, SET, SETLOCAL, SHIFT, START, TIME, TITLE, TYPE, VER, VERIFY, VOL

# Capturing stdout and stderr

- Add stdout, stderr args
- Assign subprocess.PIPE

To capture stdout and stderr with the subprocess module, import **PIPE** from subprocess and assign it to the stdout and stderr parameters to run(), check\_call(), or check\_output(), as needed.

For check\_output(), the return value is the standard output; for run(), you can access the **stdout** and **stderr** attributes of the CompletedProcess instance returned by run().



output is returned as a bytes object; call decode() to turn it into a normal Python string.

## Example

### subprocess\_capture.py

```
import sys
from subprocess import check_output, Popen, CalledProcessError, STDOUT, PIPE # need to
import PIPE and STDOUT
from glob import glob
import shlex

if sys.platform == 'win32':
    CMD = 'cmd /c dir'
    FILES = r'..\DATA\t*'
else:
    CMD = 'ls -ld'
    FILES = '../DATA/t*'

cmd_words = shlex.split(CMD)
cmd_files = glob(FILES)

full_cmd = cmd_words + cmd_files

# capture only stdout
try:
    output = check_output(full_cmd) # check_output() returns stdout
    print("Output:", output.decode(), sep='\n') # stdout is returned as bytes (decode to
    str)
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```

```
# capture stdout and stderr together
try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=STDOUT) # assign PIPE to stdout, so it is
    captured; assign STDOUT to stderr, so both are captured together
    stdout, stderr = proc.communicate() # call communicate to get the input streams of
    the process; it returns two bytes objects representing stdout and stderr
    print("Output:", stdout.decode()) # decode the stdout object to a string
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)

try:
    cmd = cmd_words + cmd_files + ['spam.txt']
    proc = Popen(cmd, stdout=PIPE, stderr=PIPE) # assign PIPE to stdout and PIPE to
    stderr, so both are captured individually
    stdout, stderr = proc.communicate() # now stdout and stderr each have data
    print("Output:", stdout.decode()) # decode from bytes and output
    print("Error:", stderr.decode()) # decode from bytes and output
except CalledProcessError as e:
    print("Process failed with return code", e.returncode)

print('-' * 50)
```



**subprocess\_capture.py**

Output:

```
-rw-r--r-- 1 jstrick staff 3178541 Nov 2 2020 ../DATA/tate_data.zip
-rw-r--r-- 1 jstrick staff 869349 Apr 9 2022 ../DATA/taxis.csv
drwxr-xr-x 10 jstrick staff 320 Oct 9 2022 ../DATA/templates
-rwxr-xr-x@ 1 jstrick staff 279 Sep 3 2022 ../DATA/test_scores.txt
-rwxr-xr-x 1 jstrick staff 2198 Feb 14 2016 ../DATA/textfiles.zip
-rw-r--r-- 1 jstrick staff 9729 Apr 7 2022 ../DATA/tips.csv
-rw-r--r-- 1 jstrick staff 57018 Apr 8 2022 ../DATA/titanic.csv
-rw-r--r-- 1 jstrick staff 106960 Jul 26 2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick staff 284160 Jul 26 2017 ../DATA/titanic3.xls
-rwxr-xr-x 1 jstrick staff 73808 Feb 14 2016 ../DATA/tolkien.txt
-rw-r--r--@ 1 jstrick staff 4461 Oct 19 2023 ../DATA/train.csv
-rw-r--r--@ 1 jstrick staff 32128 May 2 2022 ../DATA/tree_census.npy
-rwxr-xr-x 1 jstrick staff 834 Feb 14 2016 ../DATA/tyger.txt
```

```
-----
Output: -rw-r--r-- 1 jstrick staff 3178541 Nov 2 2020 ../DATA/tate_data.zip
-rw-r--r-- 1 jstrick staff 869349 Apr 9 2022 ../DATA/taxis.csv
drwxr-xr-x 10 jstrick staff 320 Oct 9 2022 ../DATA/templates
-rwxr-xr-x@ 1 jstrick staff 279 Sep 3 2022 ../DATA/test_scores.txt
-rwxr-xr-x 1 jstrick staff 2198 Feb 14 2016 ../DATA/textfiles.zip
-rw-r--r-- 1 jstrick staff 9729 Apr 7 2022 ../DATA/tips.csv
-rw-r--r-- 1 jstrick staff 57018 Apr 8 2022 ../DATA/titanic.csv
-rw-r--r-- 1 jstrick staff 106960 Jul 26 2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick staff 284160 Jul 26 2017 ../DATA/titanic3.xls
-rwxr-xr-x 1 jstrick staff 73808 Feb 14 2016 ../DATA/tolkien.txt
-rw-r--r--@ 1 jstrick staff 4461 Oct 19 2023 ../DATA/train.csv
-rw-r--r--@ 1 jstrick staff 32128 May 2 2022 ../DATA/tree_census.npy
-rwxr-xr-x 1 jstrick staff 834 Feb 14 2016 ../DATA/tyger.txt
-rw-r--r--@ 1 jstrick staff 22 Sep 9 11:33 spam.txt
```

```
-----
Output: -rw-r--r-- 1 jstrick staff 3178541 Nov 2 2020 ../DATA/tate_data.zip
-rw-r--r-- 1 jstrick staff 869349 Apr 9 2022 ../DATA/taxis.csv
drwxr-xr-x 10 jstrick staff 320 Oct 9 2022 ../DATA/templates
-rwxr-xr-x@ 1 jstrick staff 279 Sep 3 2022 ../DATA/test_scores.txt
-rwxr-xr-x 1 jstrick staff 2198 Feb 14 2016 ../DATA/textfiles.zip
-rw-r--r-- 1 jstrick staff 9729 Apr 7 2022 ../DATA/tips.csv
-rw-r--r-- 1 jstrick staff 57018 Apr 8 2022 ../DATA/titanic.csv
-rw-r--r-- 1 jstrick staff 106960 Jul 26 2017 ../DATA/titanic3.csv
-rw-r--r--@ 1 jstrick staff 284160 Jul 26 2017 ../DATA/titanic3.xls
-rwxr-xr-x 1 jstrick staff 73808 Feb 14 2016 ../DATA/tolkien.txt
-rw-r--r--@ 1 jstrick staff 4461 Oct 19 2023 ../DATA/train.csv
-rw-r--r--@ 1 jstrick staff 32128 May 2 2022 ../DATA/tree_census.npy
-rwxr-xr-x 1 jstrick staff 834 Feb 14 2016 ../DATA/tyger.txt
```

```
-rw-r--r--@ 1 jstrick staff    22 Sep  9 11:33 spam.txt
```

Error:

-----

## Permissions

- Simplest is `os.access()`
- Get mode from `os.lstat()`
- Use binary AND with permission constants

Each entry in a Unix filesystem has a inode. The inode contains low-level information for the file, directory, or other filesystem entity. Permissions are stored in the *mode*, which is a 16-bit unsigned integer. The first 4 bits indicate what kind of entry it is, and the last 12 bits are the permissions.

To see if a file or directory is readable, writable, or executable use `os.access()`. To test for specific permissions, use the `os.lstat()` method to return a tuple of inode data, and use the `S_IMODE()` method to get the mode information as a number. Then use predefined constants such as `stat.S_IRUSR`, `stat.S_IWGRP`, etc. to test for permissions.

## Example

### file\_access.py

```
import sys
import os

if len(sys.argv) < 2:
    start_dir = "."
else:
    start_dir = sys.argv[1]

for base_name in os.listdir(start_dir): # os.listdir() lists the contents of a directory
    file_name = os.path.join(start_dir, base_name)
    if os.access(file_name, os.W_OK): # os.access() returns True if file has specified
        permissions (can be os.W_OK, os.R_OK, or os.X_OK, combined with | (OR))
        print(file_name, "is writable")
```

### *file\_access.py*

```
./boto3_build_function.py is writable
./db_mysql_basics.py is writable
./bs4_parse_longest_rivers.py is writable
./edgar_cc_class.py is writable
./zipfile_write.py is writable
./db_postgres_get_version.py is writable
./deco_bark.py is writable
./bs4_parse_html.py is writable
./xml_from_presidents.py is writable
./fmt_types.py is writable
```

...

## Using shutil

- Portable ways to copy, move, and delete files
- Create archives
- Misc utilities

The **shutil** module provides portable functions for copying, moving, renaming, and deleting files. There are several variations of each command, depending on whether you need to copy all the attributes of a file, for instance.

The module also provides an easy way to create a zip file or compressed **tar** archive of a folder.

In addition, there are some miscellaneous convenience routines.

## Example

### shutil\_ex.py

```
import shutil
import os

shutil.copy('../DATA/alice.txt', 'betsy.txt') # copy file

print("betsy.txt exists:", os.path.exists('betsy.txt'))

shutil.move('betsy.txt', 'fred.txt') # rename file
print("betsy.txt exists:", os.path.exists('betsy.txt'))
print("fred.txt exists:", os.path.exists('fred.txt'))

new_folder = 'remove_me'

os.mkdir(new_folder) # create new folder
shutil.move('fred.txt', new_folder)

shutil.rmtree(new_folder) # recursively remove folder

print(f"{new_folder} exists:", os.path.exists(new_folder))
```

### shutil\_ex.py

```
betsy.txt exists: True
betsy.txt exists: False
fred.txt exists: True
remove_me exists: False
```

## Creating a useful command line script

- More than just some lines of code
- Input + Business Logic + Output
- Process files for input, or STDIN
- Allow options for customizing execution
- Log results

A good system administration script is more than just some lines of code hacked together. It needs to gather data, apply the appropriate business logic, and, if necessary, output the results of the business logic to the desired destination.

Python has two tools in the standard library to help create professional command line scripts. One of these is the **argparse** module, for parsing options and parameters on the script's command line. The other is `fileinput`, which simplifies processing a list of files specified on the command line.

We will also look at the logging module, which can be used in any application to output to a variety of log destinations, including a plain file, syslog on Unix-like systems or the NTLog service on Windows, or even email.

## Creating filters

- Filter reads files or STDIN and writes to STDOUT

Common on Unix systems Well-known filters: awk, sed, grep, head, tail, cat Reads command line arguments as files, otherwise STDIN use `fileinput.input()`

A common kind of script iterates over all lines in all files specified on the command line. The algorithm is

```
for filename in sys.argv[1:]:
    with open(filename) as F:
        for line in F:
            # process line
```

Many Unix utilities are written to work this way – sed, grep, awk, head, tail, sort, and many more. They are called filters, because they filter their input in some way and output the modified text. Such filters read STDIN if no files are specified, so that they can be piped into.

The `fileinput.input()` class provides a shortcut for this kind of file processing. It implicitly loops through `sys.argv[1:]`, opening and closing each file as needed, and then loops through the lines of each file. If `sys.argv[1:]` is empty, it reads `sys.stdin`. If a filename in the list is `-`, it also reads `sys.stdin`.

`fileinput` works on Windows as well as Unix and Unix-like platforms.

To loop through a different list of files, pass an iterable object as the argument to `fileinput.input()`.

There are several methods that you can call from `fileinput` to get the name of the current file, e.g.



Table 44. *fileinput* Methods

Method	Description
<code>filename()</code>	Name of current file being read
<code>lineno()</code>	Cumulative line number from all files read so far
<code>filelineno()</code>	Line number of current file
<code>isfirstline()</code>	<code>True</code> if current line is first line of a file
<code>isstdin()</code>	<code>True</code> if current file is <code>sys.stdin</code>
<code>close()</code>	Close <code>fileinput</code>

## Example

### file\_input.py

```
import fileinput

# fileinput.input() is an iterator of all lines
# in all files in sys.argv[1:]
for line in fileinput.input():
    if 'bird' in line:
        # fileinput.filename() has the name of the current file
        print(f'{fileinput.filename()}: {line}', end='')
```

### *file\_input.py*

```
../DATA/parrot.txt: For the first few seconds there is a terrible din. The bird kicks
../DATA/parrot.txt: bird may be hurt. After a couple of minutes of silence, he's so
../DATA/parrot.txt: The bird calmly climbs onto the man's out-stretched arm and says,
../DATA/alice.txt: with the birds and animals that had fallen into it: there were a
../DATA/alice.txt: bank--the birds with draggled feathers, the animals with their
../DATA/alice.txt: some of the other birds tittered audibly.
../DATA/alice.txt: and confusion, as the large birds complained that they could not
../DATA/alice.txt: after the birds! Why, she'll eat a little bird as soon as look
```

# Parsing the command line

- Parse and analyze **sys.argv**
- Use **argparse**
  - Parses entire command line
  - Flexible
  - Validates options and arguments

Many command line scripts need to accept options and arguments. In general, options control the behavior of the script, while arguments provide input. Arguments are frequently file names, but can be anything. All arguments are available in Python via `sys.argv`

There are at least three modules in the standard library to parse command line options. The oldest module is **getopt** (earlier than v1.3), then **optparse** (introduced 2.3, now deprecated), and now, **argparse** is the latest and greatest. (Note: **argparse** is only available in 2.7 and 3.0+).

To get started with **argparse**, create an `ArgumentParser` object. Then, for each option or argument, call the parser's `add_argument()` method.

The `add_argument()` method accepts the name of the option (e.g. *-count*) or the argument (e.g. *filename*), plus named parameters to configure the option.

Once all arguments have been described, call the parser's `parse_args()` method. (By default, it will process `sys.argv`, but you can pass in any list or tuple instead.) `parse_args()` returns an object containing the arguments. You can access the arguments using either the name of the argument or the name specified with `dest`.

One useful feature of **argparse** is that it will convert command line arguments for you to the type specified by the `type` parameter. You can write your own function to do the conversion, as well.

Another feature is that **argparse** will automatically create a help option, *-h*, for your application, using the help strings provided with each option or parameter.

**argparse** parses the entire command line, not just arguments

Table 45. `add_argument()` named parameters

parameter	description
<code>action</code>	How to handle argument <code>append</code> : store multiple instances of argument as a list <code>append_const</code> : append value of <code>const</code> parameter to a list <code>count</code> : store number of instances of argument <code>help</code> : print help and exit (expects <code>help=</code> parameter to <code>add_argument()</code> ) <code>store</code> : store argument <i>default</i> <code>store_const</code> : store value specified by <code>const</code> parameter <code>store_false</code> : store <code>False</code> as value <code>store_true</code> : store <code>True</code> as value <code>version</code> : print version and exit (expects <code>version=</code> parameter to <code>add_argument()</code> )
<code>choices</code>	A list of valid choices for the option
<code>const</code>	Value for options that do not take a user-specified value
<code>default</code>	Value if option not specified
<code>dest</code>	Name of attribute (defaults to argument name)
<code>help</code>	Help text for option or argument
<code>metavar</code>	A name to use in the help string (default: same as <code>dest</code> )
<code>nargs</code>	Number of arguments Default: one argument, returns string <code>*</code> : 0 or more arguments, returns list <code>+</code> : 1 or more arguments, returns list <code>?</code> : 0 or 1 arguments, returns list <code>N</code> : exactly N arguments, returns list
<code>required</code>	If set to <code>True</code> , this option must be specified
<code>type</code>	type which the command-line arguments should be converted ; one of <code>string</code> , <code>int</code> , <code>float</code> , <code>complex</code> or a function that accepts a single string argument and returns the desired object. (Default: <code>string</code> )

## Example

### parsing\_args.py

```
import re
import fileinput
import argparse

arg_parser = argparse.ArgumentParser(description="Emulate grep") # argument parser

arg_parser.add_argument(
    '-i',
    dest='ignore_case', action='store_true',
    help='ignore case'
) # add option to the parser; dest is name of option attribute

arg_parser.add_argument(
    '-n',
    dest='show_names', action='store_true',
    help='display file names'
) # add option to the parser; dest is name of option attribute

arg_parser.add_argument(
    'pattern', help='Pattern to find (required)'
) # add required argument to the parser

arg_parser.add_argument(
    'filenames', nargs='*',
    help='filename(s) (if no files specified, read STDIN)'
) # add optional arguments to the parser

args = arg_parser.parse_args() # actually parse the arguments

print('ARGS:', args, "\n")

regex = re.compile(args.pattern, re.I if args.ignore_case else 0) # compile regex

for line in fileinput.input(args.filenames): # loop over list of file names and read
    them one line at a time
    if regex.search(line):
        if args.show_names:
            print(fileinput.filename(), end=": ")
        print(line.rstrip())
```

***parsing\_args.py***

```
usage: parsing_args.py [-h] [-i] [-n] pattern [filenames ...]
parsing_args.py: error: the following arguments are required: pattern, filenames
```

***parsing\_args.py -i '\bbil' ../DATA/alice.txt ../DATA/presidents.txt***

```
ARGS: Namespace(ignore_case=True, show_names=False, pattern='\bbil',
filenames=['../DATA/alice.txt', '../DATA/presidents.txt'])
```

The Rabbit Sends in a Little Bill

```
Bill's got the other--Bill! fetch it here, lad!--Here, put 'em up
Here, Bill! catch hold of this rope--Will the roof bear?--Mind
crash)--`Now, who did that?--It was Bill, I fancy--Who's to go
then!--Bill's to go down--Here, Bill! the master says you're to
`Oh! So Bill's got to come down the chimney, has he?' said
Alice to herself. `Shy, they seem to put everything upon Bill!
I wouldn't be in Bill's place for a good deal: this fireplace is
above her: then, saying to herself `This is Bill,' she gave one
Bill!' then the Rabbit's voice along--`Catch him, you by the
Last came a little feeble, squeaking voice, (`That's Bill,'
The poor little Lizard, Bill, was in the middle, being held up by
end of the bill, "French, music, AND WASHING--extra."'
Bill, the Lizard) could not make out at all what had become of
Lizard as she spoke. (The unfortunate little Bill had left off
42:Clinton:William Jefferson 'Bill':1946-08-19:NONE:Hope:Arkansas:1993-01-20:2001-01-
20:Democratic
```

***parsing\_args.py -h***

```
usage: parsing_args.py [-h] [-i] [-n] pattern [filenames ...]
```

Emulate grep

positional arguments:

pattern      Pattern to find (required)

filenames    filename(s) (if no files specified, read STDIN)

options:

-h, --help    show this help message and exit

-i            ignore case

-n            display file names

## Simple Logging

- Specify file name
- Configure the minimum logging level
- Messages added at different levels
- Call methods on logging

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

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

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

Table 46. Logging Levels

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



## Example

### logging\_simple.py

```
import logging

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

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

### ../LOGS/simple.log

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

...

## Formatting log entries

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

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

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

### Example

#### `logging_formatted.py`

```
import logging

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

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

**../LOGS/formatted.log**

```
INFO root 09/27/24-10:22:46 logging_formatted.py 11 this is information
WARNING root 09/27/24-10:22:46 logging_formatted.py 12 this is a warning
ERROR root 09/27/24-10:22:46 logging_formatted.py 13 this is an ERROR
ERROR root 09/27/24-10:22:46 logging_formatted.py 15 Invalid value 38.7
INFO root 09/27/24-10:22:46 logging_formatted.py 16 this is information
CRITICAL root 09/27/24-10:22:46 logging_formatted.py 17 this is critical
```

Table 47. Log entry formatting directives

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

## Logging exception information

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

The `logging.exception()` function will add exception information to the log message. It should only be called in an **except** block.

### Example

#### `logging_exception.py`

```
import logging

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

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

**../LOGS/exception.log**

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

## Logging to other destinations

- Use specialized handlers to write to other destinations
- Multiple handlers can be added to one logger
  - NTEventLogHandler for Windows event log
  - SysLogHandler for syslog
  - SMTPHandler for logging via email

The logging module provides some preconfigured log handlers to send log messages to destinations other than a file.

Each handler has custom configuration appropriate to the destination. Multiple handlers can be added to the same logger, so a log message will go to a file and to email, for instance, and each handler can have its own minimum level. Thus, all messages could go to the message file, but only CRITICAL messages would go to email.

Be sure to read the documentation for the particular log handler you want to use



On Windows, you must run the example script (`logging.altdest.py`) as administrator. You can find **Command Prompt (admin)** on the main Windows 8/10 menu. You can also right-click on **Command Prompt** from the Windows 7 menu and choose "Run as administrator".

## Example

### logging\_altdest.py

```
import sys
from getpass import getpass
import logging
import logging.handlers

logger = logging.getLogger() # get logger
logger.setLevel(logging.DEBUG) # minimum log level

if sys.platform == 'win32':
    eventlog_handler = logging.handlers.NTEventLogHandler("Python Log Test") # create NT
    event log handler
    logger.addHandler(eventlog_handler) # install NT event handler
else:
    syslog_handler = logging.handlers.SysLogHandler() # create syslog handler
    logger.addHandler(syslog_handler) # install syslog handler

smtp_password = getpass("SMTP Password: ")

# note -- use your own SMTP server...
email_handler = logging.handlers.SMTPHandler(
    ("smtp2go.com", 2525),
    'LOGGER@pythonclass.com',
    ['jstrickler@gmail.com'],
    'Alternate Destination Log Demo',
    ('pythonclass', smtp_password),
) # create email handler

logger.addHandler(email_handler) # install email handler

logger.debug('this is debug') # goes to all handlers
logger.critical('this is critical') # goes to all handlers
logger.warning('this is a warning') # goes to all handlers
```



## Chapter 18 Exercises

### Exercise 18-1 (copy\_files.py)

Write a script to find all text files (only the files that end in ".txt") in the DATA folder of the student files and copy them to C:\TEMP (Windows) or /tmp (non-windows). On Windows, create the C:\TEMP folder if it does not already exist.

Add logging to the script, and log each filename at level INFO.



use `shutil.copy()` to copy the files.

# Chapter 19: Regular Expressions

## Objectives

- Using RE patterns
- Creating regular expression objects
- Matching, searching, replacing, and splitting text
- Adding option flags to a pattern
- Specifying capture groups
- Replacing text with backrefs and callbacks

# Regular expressions

- Specialized language for pattern matching
- Begun in UNIX; expanded by **Perl**
- Python adds some conveniences

Regular expressions (or REs) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as *Does this string match the pattern?*, or *Is there a match for the pattern anywhere in this string?*. You can also use REs to modify a string or to split it apart in various ways.

— Python Regular Expression HOWTO

Regular expressions were first popularized thirty years ago as part of Unix text processing programs such as **vi**, **sed**, and **awk**. While they were improved incrementally over the years, it was not until the advent of **Perl** that they substantially changed from the originals. **Perl** added extensions of several different kinds – shortcuts for common sequences, look-ahead and look-behind assertions, non-greedy repeat counts, and a general syntax for embedding special constructs within the regular expression itself.

Python uses **Perl**-style regular expressions (AKA PCREs) and adds a few extensions of its own.

Two good web sites for creating and deciphering regular expressions:

Regex 101: <https://regex101.com/#python>

Pythex: <http://www.pythex.org/>

Two sites for having fun (yes, really!) with regular expressions:

Regex Golf: <https://alf.nu/RegexGolf>

Regex Crosswords: <https://regexcrossword.com/>

## RE syntax overview

- Regular expressions contain branches
- Branches contain atoms
- Atoms may be quantified
- Branches and atoms may be anchored

A regular expression consists of one or more *branches* separated by the pipe symbol. The regular expression matches any text that is matched by any of the branches.

A branch is a left-to-right sequence of *atoms*. Each atom consists of either a one-character match or a parenthesized group. Each atom can have a *quantifier* (repeat count). The default repeat count is one.

A branch can be anchored to the beginning or end of the text. Any part of a branch can be anchored to the beginning or end of a word.



There is frequently only one branch.

Branch<sub>1</sub> | Branch<sub>2</sub>

Atom<sub>1</sub>Atom<sub>2</sub>Atom<sub>3</sub>(Atom<sub>4</sub>Atom<sub>5</sub>Atom<sub>6</sub>)Atom<sub>7</sub>

A a 1 ;

. \d \w \s  
[abc]  
[^abc]

Atom<sub>repeat</sub>

Table 48. Regular Expression Metacharacters

Pattern	Description
.	any character
[abc]	any character in set
[^abc]	any character not in set
\w, \W	any word, non-word char
\d, \D	any digit, non-digit
\s, \S	any space, non-space char
^, \$	beginning, end of string
\b	beginning or end of word
\	escape a special character
*, +, ?	0 or more, 1 or more, 0 or 1
{m}	exactly m occurrences
{m, }	at least m occurrences
{m, n}	m through n occurrences
a b	match a or b
(?aiLmsux)	Set the A, I, L, M, S, U, or X flag for the RE (see below).
(?:...)	Non-capturing version of regular parentheses.
(?P<name>...)	The substring matched by the group is accessible by name.
(?P=name)	Matches the text matched earlier by the group named name.
(?#...)	A comment; ignored.
(?=...)	Matches if ... matches next, but doesn't consume the string.
(?!...)	Matches if ... doesn't match next.
(?<=...)	Matches if preceded by ... (must be fixed length).
(?<!=...)	Matches if not preceded by ... (must be fixed length).

## Finding matches

- Module defines static functions
- Arguments: pattern, string

There are three primary methods for finding matches.

### Find first match

```
re.search(pattern, string)
```

Searches `s` and returns the first match. Returns a match object (**SRE\_Match**) on success or **None** on failure. A match object is always evaluated as **True**, and so can be used in **if** statements and **while** loops. Call the **group()** method on a match object to get the matched text.

### Find all matches

```
re.finditer(pattern, string)
```

Provides a match object for each match found. Normally used with a **for** loop.

### Retrieve text of all matches

```
re.findall(pattern, string)
```

Finds all matches and returns a list of matched strings. Since regular expressions generally contain many backslashes, it is usual to specify the pattern with a raw string.

### Other search methods

`re.match()` is like `re.search()`, but searches for the pattern at beginning of `s`. There is an implied `^` at the beginning of the pattern.

Likewise `re.fullmatch()` only succeeds if the pattern matches the entire string. `^` and `$` around the pattern are implied.

## Example

### regex\_finding\_matches.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}' # store pattern in raw string

if re.search(pattern, s): # search returns True on match
    print("Found pattern.")
print()

m = re.search(pattern, s) # search actually returns match object
print(m)
if m:
    print("Found:", m.group(0)) # group(0) returns text that was matched by entire
    expression (or just m.group())
print()

for m in re.finditer(pattern, s): # iterate over all matches in string:
    print(m.group())
print()

matches = re.findall(pattern, s) # return list of all matches
print("matches:", matches)
```

***regex\_finding\_matches.py***

Found pattern.

<re.Match object; span=(12, 17), match='M-302'>

Found: M-302

M-302

H-476

Q-51

A-110

H-332

Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']



## RE objects

- `re` object contains a compiled regular expression
- Call methods on the object, with strings as parameters.

An `re` object is created by calling `re.compile()` with a pattern string. Once created, the object can be used for searching (matching), replacing, and splitting any string. `re.compile()` has an optional argument for flags which enable special features or fine-tune the match.



It is generally a good practice to create your `re` objects in a location near the top of your script, and then use them as necessary

## Example

### regex\_objects.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'[A-Z]-\d{2,3}') # Create an re (regular expression) object

if rx_code.search(s): # Call search() method from the object
    print("Found pattern.")
print()

m = rx_code.search(s)
if m:
    print("Found:", m.group())
print()

for m in rx_code.finditer(s):
    print(m.group())
print()

matches = rx_code.findall(s)
print("matches:", matches)
```

***regex\_objects.py***

Found pattern.

Found: M-302

M-302

H-476

Q-51

A-110

H-332

Y-45

matches: ['M-302', 'H-476', 'Q-51', 'A-110', 'H-332', 'Y-45']

## Compilation flags

- Fine-tune match
- Add readability

When using functions from `re`, or when compiling a pattern, you can specify various flags to control how the match occurs. The flags are aliases for numeric values, and can be combined with `|`, the bitwise **OR** operator. Each flag has a short for and a long form.

### Ignoring case

```
re.I, re.IGNORECASE
```

Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match "Spam", "spam", or "spAM". This lower-casing doesn't take the current locale into account; it will if you also set the `LOCALE` flag.

### Using the locale

```
re.L, re.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.

## Ignoring whitespace

```
re.X, re.VERBOSE
```

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. It also enables you to put comments within a RE that will be ignored by the engine; comments are marked by a `#` that's neither in a character class or preceded by an unescaped backslash. Use a triple-quoted string for your pattern to make best advantage of this flag.

```
RE_ENTRY = r"""
    (?P<month>[A-Z][a-z]{2})\s+(?P<day>\d+)\s+      # date
    (?P<hour>\d{2}):(?P<minute>\d{2}):(?P<second>\d{2})\s+ # timestamp
    (?P<hostname>\S+)\s+                          # hostname
    (?P<process_name>.*?)                          # process name
    \[(?P<pid>\d+)\]\s+                             # PID
    (?P<message>.*?)                               # message
    """
```

## Example

### regex\_flags.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'[A-Z]-\d{2,3}'

if re.search(pattern, s, re.IGNORECASE): # make search case-insensitive
    print("Found pattern.")
print()

m = re.search(pattern, s, re.I | re.M) # short version of flag
if m:
    print("Found:", m.group())
print()

for m in re.finditer(pattern, s, re.I):
    print(m.group())
print()

matches = re.findall(pattern, s, re.I)
print("matches:", matches)
```

***regex\_flags.py***

Found pattern.

Found: M-302

M-302

r-99

H-476

Q-51

z-883

A-110

H-332

Y-45

matches: ['M-302', 'r-99', 'H-476', 'Q-51', 'z-883', 'A-110', 'H-332', 'Y-45']

## Working with embedded newlines

- `re.S`, `re.DOTALL` lets `.` match newline
- `re.M`, `re.MULTILINE` lets `^` and `$` match lines

Some text contains newlines (`\n`), representing multiple lines within the string. There are two regular expression flags you can use with `re.search()` and other functions to control how they are searched.



These flags are not useful if the string has no embedded newlines.

### Treating text like a single string

By default, `.` does not match newline. Thus, `spam.*ham` will not match the text if `spam` is on one line and `ham` is on a subsequent line. The `re.DOTALL` flag allows `.` to match newline, enabling searches that span lines.

`re.S` is an abbreviation for `re.DOTALL`.

### Treating text like multiple lines

Normally, `^` only matches the beginning of a string and `$` only matches the end. If you use the `re.MULTILINE` flag, these anchors will also match the beginning and end of embedded lines.

`re.M` is an abbreviation for `re.MULTILINE`.



## Example

### regex\_newlines.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

line_start_word = r'^\w+' # match word at beginning of string/line

matches = re.findall(line_start_word, s) # only matches at beginning of string
print("matches:", matches)
print()

matches = re.findall(line_start_word, s, re.M) # matches at beginning of lines
print("matches:", matches)
print()

phrase = r"aliquip.*commodo"

match = re.search(phrase, s)
if match:
    print(match.group(), match.start())
else:
    print(f"{phrase} not found")
print()

match = re.search(phrase, s, re.S)
if match:
    print(repr(match.group()), match.start())
else:
    print(f"{phrase} not found")
print()
```

***regex\_newlines.py***

```
matches: ['lorem']
```

```
matches: ['lorem', 'ad', 'ea', 'voluptate', 'Excepteur', 'officia']
```

```
aliquip.*commodo not found
```

```
'aliquip ex \nea commodo' 223
```

# Groups

- Marked with parentheses
- Capture whatever matched pattern within
- Access with `match.group()`

Sometimes you need to grab just *part* of the text matched by an RE. Your pattern can contain one or more subgroups which match the parts you're interested in. For example, you could pull the hour, minute, and second separately out of the text "11:13:42". You can write a pattern that matches the entire time string and has a group for each part:

```
r"(\d{2}):(\d{2}):(\d{2})"
```

Groups are marked with parentheses, and "capture" whatever matched the pattern inside the parentheses.

`re.findall()` returns a list of tuples, where each tuple contains the matches for all each group.

To access groups in more detail, use `re.finditer()` and call the `.group()` method on each match object. The default group is 0, which is always the entire match. It can be retrieved with either `_match_.group(0)`, or just `_match_.group()`. Then, `_match_.group(1)` returns text matched by the first set of parentheses, `_match_.group(2)` returns the text from the second set, etc.

In the same vein, `_match_.start()` or `_match_.start(0)` return the beginning 0-based offset of the entire match; `_match_.start(1)` returns the beginning offset of group 1, and so forth. The same is true for `_match_.end()` and `_match_.end(n)`.

`_match_.span()` returns the the start and end offsets for the entire match. `_match_.span(1)` returns start and end offsets for group 1, and so forth.

## Example

### regex\_group.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'([A-Z])-(\d{2,3})' # parens delimit groups

print("Group 0          Group 1          Group 2")
header2 = "text  start  end  text  start  end  text  start  end"
print(header2)
print("-" * len(header2))

for m in re.finditer(pattern, s):
    print(
        f"{m.group(0):5s} {m.start(0):3d} {m.end(0):3d}"
        f" {m.group(1):5s} {m.start(1):3d} {m.end(1):3d}"
        f" {m.group(2):5s} {m.start(2):3d} {m.end(2):3d}"
    )
print()

matches = re.findall(pattern, s) # findall() returns list of tuples containing groups
print("matches:", matches)
```

***regex\_group.py***

Group 0			Group 1			Group 2		
text	start	end	text	start	end	text	start	end
M-302	12	17	M	12	13	302	14	17
H-476	102	107	H	102	103	476	104	107
Q-51	134	138	Q	134	135	51	136	138
A-110	398	403	A	398	399	110	400	403
H-332	436	441	H	436	437	332	438	441
Y-45	470	474	Y	470	471	45	472	474

```
matches: [('M', '302'), ('H', '476'), ('Q', '51'), ('A', '110'), ('H', '332'), ('Y', '45')]
```

## Special groups

- Non-capture groups are used just for grouping
- Named groups allow retrieval of sub-expressions by name rather than number
- Look-ahead and look-behind match, but do not capture

There are two variations on RE groups that are useful. If the first character inside the group is a question mark, then the parentheses contain some sort of extended pattern, designated by the next character after the question mark.

### Non-capture groups

The most basic special is `(?:pattern)`, which groups but does not capture.

### Named groups

A welcome addition in Python is the concept of named groups. Instead of remembering that the month is the 3rd group and the year is the 4th group, you can use the syntax `(?'P<name>pattern')`. You can then call `__match__.group("__name__")` to fetch the text match by that sub-expression; alternatively, you can call `__match__.groupdict()`, which returns a dictionary where the keys are the pattern names, and the values are the text matched by each pattern.

### Lookaround assertions

Another advanced concept is an assertion, either lookahead or lookbehind. A lookahead assertion uses the syntax `(?=pattern)`. The string being matched must match the lookahead, but does not become part of the overall match.

For instance, `\d(?:st|nd|rd|th)(?=street)` matches "1st", "2nd", etc., but only where they are followed by "street".

## Example

### regex\_special.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

pattern = r'(?P<letter>[A-Z])-(?P<number>\d{2,3})' # Use (?P<NAME>...) to name groups

for m in re.finditer(pattern, s):
    print(m.group('letter'), m.group('number')) # Use m.group(NAME) to retrieve text
```

### regex\_special.py

```
M 302
H 476
Q 51
A 110
H 332
Y 45
```

## Replacing text

- Use `re.sub(pattern, replacement, string[, count])`
- `re.subn(...)` returns string and count

To find and replace text using a regular expression, use the `re.sub()` method. It takes the pattern, the replacement text and the string to search as arguments, and returns the modified string.

The third (optional) argument is one or more compilation flags. The fourth argument is the maximum number of replacements to make. Both are optional, but if the fourth argument is specified and no flags are needed, use `0` as a placeholder for the third argument.



Be sure to put the arguments in the proper order!

### Example

#### regex\_sub.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub("[REDACTED]", s) # replace pattern with string
print(s2)
print()

s3, count = rx_code.subn("___", s) # subn returns tuple with result string and
replacement count
print(f"Made {count} replacements")
print()
print(s3)
```



**regex\_sub.py**

```
lorem ipsum [REDACTED] dolor sit amet, consectetur [REDACTED] adipiscing elit, sed do
  eiusmod tempor incididunt [REDACTED] ut labore et dolore magna [REDACTED] aliqua. Ut
  enim
  ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
  ea commodo [REDACTED] consequat. Duis aute irure dolor in reprehenderit in
  voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
  Excepteur sint occaecat [REDACTED] cupidatat non proident, sunt in [REDACTED] culpa qui
  officia deserunt [REDACTED] mollit anim id est laborum
```

Made 8 replacements

```
lorem ipsum ___ dolor sit amet, consectetur ___ adipiscing elit, sed do
  eiusmod tempor incididunt ___ ut labore et dolore magna ___ aliqua. Ut enim
  ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
  ea commodo ___ consequat. Duis aute irure dolor in reprehenderit in
  voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
  Excepteur sint occaecat ___ cupidatat non proident, sunt in ___ culpa qui
  officia deserunt ___ mollit anim id est laborum
```

## Replacing with backrefs

- Use match or groups in replacement text
  - `\g<__num__>` *group number*
  - `\g<__name__>` *group name*
  - `{backslash}__num__` *shortcut for group number*

It is common to need all or part of the match in the replacement text. To allow this, the `re` module provides *backrefs*, which are special variables that *refer back* to the groups in the match, including group 0 (the entire match).

To refer to a particular group, use the syntax `\g<__num__>`.

To refer to a particular named group, use `\g<__name__>`.

As a shortcut, you can use `{backslash}__num__`. This does not work for group 0 — use `\g<0>` instead.



`\10` is group 10, not group 1 followed by 0

## Example

### regex\_sub\_backrefs.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est laborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

s2 = rx_code.sub(r"(\g<1>)[\g<2>]", s)
print(f"s2: {s2}")
print('-' * 60)

s3 = rx_code.sub(r"\g<number>-\g<letter>", s)
print(f"s3: {s3}")
print('-' * 60)

s4 = rx_code.sub(r"[\1:\2]", s)
print(f"s4: {s4}")
print('-' * 60)
```

**regex\_sub\_backrefs.py**

```
s2: lorem ipsum (M)[302] dolor sit amet, consectetur (r)[99] adipiscing elit, sed do
    eiusmod tempor incididunt (H)[476] ut labore et dolore magna (Q)[51] aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo (z)[883] consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat (A)[110] cupidatat non proident, sunt in (H)[332] culpa qui
    officia deserunt (Y)[45] mollit anim id est laborum
```

```
-----
s3: lorem ipsum 302-M dolor sit amet, consectetur 99-r adipiscing elit, sed do
    eiusmod tempor incididunt 476-H ut labore et dolore magna 51-Q aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo 883-z consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat 110-A cupidatat non proident, sunt in 332-H culpa qui
    officia deserunt 45-Y mollit anim id est laborum
```

```
-----
s4: lorem ipsum [M:302] dolor sit amet, consectetur [r:99] adipiscing elit, sed do
    eiusmod tempor incididunt [H:476] ut labore et dolore magna [Q:51] aliqua. Ut enim
    ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
    ea commodo [z:883] consequat. Duis aute irure dolor in reprehenderit in
    voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
    Excepteur sint occaecat [A:110] cupidatat non proident, sunt in [H:332] culpa qui
    officia deserunt [Y:45] mollit anim id est laborum
    -----
```

## Replacing with a callback

- Replacement can be a callback function
- Function expects match object, returns replacement text
- Use either normally defined function or a lambda

In addition to using a string, possibly containing backrefs, as the replacement, you can specify a function. This function will be called once for each match, with the match object as its only parameter.

Using a callback is necessary if you need to modify any of the original text.

Whatever string the function returns will be used as the replacement text. This lets you have complete control over the replacement.

Using a callback makes it simple to:

- add text around the replacement (can also be done with backrefs)
- search ignoring case and preserve case in a replacement
- look up the text in a dictionary or database to find replacement text

## Example

### regex\_sub\_callback.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est dlaborum"""

rx_code = re.compile(r'(?P<letter>[A-Z])-(?P<number>\d{2,3})', re.I)

def update_code(m): # callback function is passed each match object
    letter = m.group('letter').upper()
    number = int(m.group('number'))
    return f'{letter}:{number:04d}' # function returns replacement text

s2, count = rx_code.subn(update_code, s) # sub takes callback function instead of
replacement text
print(s2)
print(count, "replacements made")
```

### regex\_sub\_callback.py

```
lorem ipsum M:0302 dolor sit amet, consectetur R:0099 adipiscing elit, sed do
eiusmod tempor incididunt H:0476 ut labore et dolore magna Q:0051 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo Z:0883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A:0110 cupidatat non proident, sunt in H:0332 culpa qui
officia deserunt Y:0045 mollit anim id est dlaborum
8 replacements made
```

## Splitting a string

- Syntax: `re.split(pattern, string[,max])`

The `re.split()` method splits a string into pieces, using the regex to match the delimiters, and returning the pieces as a list. The optional `max` argument limits the numbers of pieces.

### Example

#### regex\_split.py

```
import re

s = """lorem ipsum M-302 dolor sit amet, consectetur r-99 adipiscing elit, sed do
eiusmod tempor incididunt H-476 ut labore et dolore magna Q-51 aliqua. Ut enim
ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex
ea commodo z-883 consequat. Duis aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore U901 eu fugiat nulla pariatur.
Excepteur sint occaecat A-110 cupidatat non proident, sunt in H-332 culpa qui
officia deserunt Y-45 mollit anim id est dlaborum"""

# pattern is one or more non-letters
rx_wordsep = re.compile(r"^[a-z0-9-]+", re.I) # When splitting, pattern matches what you
don't want

words = rx_wordsep.split(s) # Retrieve text _separated_ by your pattern
unique_words = set(words)

print(sorted(unique_words))
```

***regex\_split.py***

```
['A-110', 'Duis', 'Excepteur', 'H-332', 'H-476', 'M-302', 'Q-51', 'U901', 'Ut', 'Y-45',  
'ad', 'adipiscing', 'aliqua', 'aliquip', 'amet', 'anim', 'aute', 'cillum', 'commodo',  
'consectetur', 'consequat', 'culpa', 'cupidatat', 'deserunt', 'dlaborum', 'do', 'dolor',  
'dolore', 'ea', 'eiusmod', 'elit', 'enim', 'esse', 'est', 'et', 'eu', 'ex',  
'exercitation', 'fugiat', 'id', 'in', 'incidunt', 'ipsum', 'irure', 'labore',  
'laboris', 'lorem', 'magna', 'minim', 'mollit', 'nisi', 'non', 'nostrud', 'nulla',  
'occaecat', 'officia', 'pariatur', 'proident', 'qui', 'quis', 'r-99', 'reprehenderit',  
'sed', 'sint', 'sit', 'sunt', 'tempor', 'ullamco', 'ut', 'velit', 'veniam', 'voluptate',  
'z-883']
```



## Chapter 19 Exercises

### Exercise 19-1 (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>

Example

```
python pyfind.py freezer DATA/words.txt DATA/parrot.txt
```

### Exercise 19-2 (mark\_big\_words\_callback.py, mark\_big\_words\_backrefs.py)

Copy `parrot.txt` to `bigwords.txt` adding asterisks around all words that are 8 or more characters long.

HINT: Use the `\b` anchor to indicate beginning or end of a word.



There are two solutions to this exercise in the ANSWERS folder: one using a callback function, and one using backrefs.

### Exercise 19-3 (print\_numbers.py)

Write a script to print out all lines in `custinfo.dat` which contain phone numbers. A phone number consists of three digits, a dash, and four more digits.

### Exercise 19-4 (word\_freq.py)

Write a script that will read a text file and print out a list of all the words in the file, normalized to lower case, and with the number of times that word occurred in the file. Use the regular expression `[^\w']+` for splitting each line into words.

Test with any of the text files in the DATA folder.

HINT: Use a dictionary for counting.



The specified pattern matches one or more characters that are neither letters, digits, underscores, nor apostrophes.

[1] Any similarity to the Unix `grep` command is purely intentional.

# Appendix A: Where do I go from here?

## Resources for learning Python

These are from Jessica Garson, who, among other things, teaches Python classes at NYU. (Used with permission).

Run the script **where\_do\_i\_go.py** to display a web page with live links.

### [Resources for Learning Python](#)

#### Just getting started

Here are some resources that can help you get started learning how to code.

- [Code Newbie Podcast](#)
- [Dive into Python3](#)
- [Learn Python the Hard Way](#)
- [Learn Python the Hard Way](#)
- [Automate the Boring Stuff with Python](#)
- [Automate the Boring Stuff with Python](#)

#### So you want to be a data scientist?

- [Data Wrangling with Python](#)
- [Data Analysis in Python](#)
- [Titanic: Machine Learning from Disaster](#)
- [Deep Learning with Python](#)
- [How to do X with Python](#)
- [Machine Learning: A Probabilistic Prospective](#)

#### So you want to write code for the web?

- [Learn flask, some great resources are listed here](#)
- [Django Polls Tutorial](#)
- [Hello Web App](#)
- [Hello Web App Intermediate](#)
- [Test-Driven-Development for Web Programming](#)
- [2 Scoops of Django](#)

- [HTML and CSS: Design and Build Websites](#)
- [JavaScript and JQuery](#)

Not sure yet, that's okay!

Here are some resources for self guided learning. I recommend trying to be very good at Python and the rest should figure itself out in time.

- [Python 3 Crash Course](#)
- [Base CS Podcast](#)
- [Writing Idiomatic Python](#)
- [Fluent Python](#)
- [Pro Python](#)
- [Refactoring](#)
- [Clean Code](#)
- [Write music with Python, since that's my favorite way to learn a new language](#)

## Appendix B: Field Guide to Python Expressions

Table 49. Python Expressions

Expression	Meaning
<code>a, b, c</code> <code>(a, b, c)</code>	tuple
<code>a,</code> <code>(a,)</code>	tuple with one element
<code>()</code>	empty tuple
<code>[a, b, c]</code>	list
<code>[]</code>	empty list
<code>{a, b, c}</code>	set
<code>{a:r, b:s, c:t}</code>	dictionary
<code>{}</code>	empty dictionary
<code>[expr for var in iterable if condition]</code>	list comprehension
<code>(expr for var in iterable if condition)</code>	generator expression
<code>{expr for var in iterable if condition}</code>	set comprehension
<code>{key:value for var in iterable if condition}</code>	dictionary comprehension
<code>a, b, c = iterable</code>	iterable unpacking
<code>a, *b, c = iterable</code>	extended iterable unpacking
<code>a if b else c</code>	conditional expression
<code>lambda VAR ...: VALUE</code>	lambda function

# Appendix C: String Formatting

## Overview

- Strings have a `format()` method
- Allows values to be inserted in strings
- Values can be formatted
- Add a field as placeholders for variable
- Field syntax: `{SELECTOR:FORMATTING}`
- Selector can be empty, index, or keyword
- Formatting controls alignment, width, padding, etc.

Python provides a powerful and flexible way to format data. The string method `format()` takes one or more parameters, which are inserted into the string via placeholders.

The placeholders, called fields, consist of a pair of braces enclosing parameter selectors and formatting directives.

The selector can be followed by a set of formatting directives, which always start with a colon. The simplest directives specify the type of variable to be formatted. For instance, `{1:d}` says to format the second parameter as an integer; `{0:.2f}` says to format the first parameter as a float, rounded to two decimal points.

The formatting part can consist of the following components, which will be explained in detail in the following pages:

```
: [[fill]]align[[sign]][#][0][width][,][.precision][type]
```

## Parameter Selectors

- Null for auto-numbering
- Can be numbers or keywords
- Start at 0 for numbers

Selectors refer to which parameter will be used in a placeholder.

Null (empty) selectors — the most common — will be treated as though they were filled in with numbers from left to right, beginning with 0. Null selectors cannot be mixed with numbered or named selectors — either all of the selectors or none of the selectors must be null.

Non-null selectors can be either numeric indices or keywords (strings). Thus, {0} will be replaced with the first parameter, {4} will be replaced with the fifth parameter, and so on. If using keywords, then `__name__` will be replaced by the value of keyword *name*, and {age} will be replaced by keyword *age*.

Parameters do not have to be in the same order in which they occur in the string, although they typically are. The same parameter can be used in multiple fields.

If positional and keyword parameters are both used, the keyword parameters must come after all positional parameters.

## Example

### *fmt\_params.py*

```
person = 'Bob'
age = 22

print("{} is {} years old.".format(person, age)) # Placeholders can be numbered
print("{} , {} , {} your boat".format('row')) # Placeholders can be reused
print("The {}-year-old is {}".format(person, age)) # They do not have to be in order
(but usually are)
print("{name} is {age} years old.".format(name=person, age=age)) # Selectors can be
named
print()
print("{} is {} years old.".format(person, age)) # Empty selectors are autonumbered (but
all selectors must either be empty or explicitly numbered)
print("{name} is {} and his favorite color is {}".format(22, 'blue', name='Bob')) #
Named and numbered selectors can be mixed
```

### *fmt\_params.py*

```
Bob is 22 years old.
row, row, row your boat
The 22-year-old is Bob
Bob is 22 years old.

Bob is 22 years old.
Bob is 22 and his favorite color is blue
```

## f-strings

- **f** in front of literal strings
- More readable
- Same rules as `__string__.format()`

Starting with version 3.6, Python also supports *f-strings*.

The big difference from the `format()` method is that the parameters are inside the `{}` placeholders. Place formatting details after a `:` as usual.

Since the parameters are part of the placeholders, parameter numbers are not used.

All of the following formatting tools work with both `__string__.format()` and f-strings.

### Example

#### **`fmt_fstrings.py`**

```
person = 'Bob'
age = 22
result = 39.128935

print(f"{person} is {age} years old.")
print(f"The {age}-year-old is {person}.")
print(f"Result is {result:.2f}")
print()
```

#### **`fmt_fstrings.py`**

```
Bob is 22 years old.
The 22-year-old is Bob.
Result is 39.13
```



## Data types

- Fields can specify data type
- Controls formatting
- Raises error for invalid types

The type part of the format directive tells the formatter how to convert the value. Builtin types have default formats – *s* for strings, *d* for integers, *f* for float.

Some data types can be specified as either upper or lower case. This controls the output of letters. E.g, `{:x}` would format the number 48879 as *beef*, but `{:X}` would format it as *BEEF*.

The type must generally match the type of the parameter. An integer cannot be formatted with type *s*. Integers can be formatted as floats, but not the other way around. Only integers may be formatted as binary, octal, or hexadecimal.

## Example

### *fmt\_types.py*

```
person = 'Bob'
value = 488
bigvalue = 3735928559
result = 234.5617282027

print('{:s}'.format(person))    # String
print('{name:s}'.format(name=person))  # String
print('{:d}'.format(value))      # Integer (displayed as decimal)
print('{:b}'.format(value))      # Integer (displayed as binary)
print('{:o}'.format(value))      # Integer (displayed as octal)
print('{:x}'.format(value))      # Integer (displayed as hex)
print('{:X}'.format(bigvalue))   # Integer (displayed as hex with uppercase digits)
print('{:f}'.format(result))     # Float (defaults to 6 places after the decimal point)
print('{:.2f}'.format(result))   # Float rounded to 2 decimal places
```

***fmt\_types.py***

```

Bob
Bob
488
111101000
750
1e8
DEADBEEF
234.561728
234.56

```

Table 50. Formatting Types

Type	Description
<b>b</b>	Binary – converts number to base 2
<b>c</b>	Character – converts to corresponding character, like chr()
<b>d</b>	Decimal – outputs number in base 10
<b>e, E</b>	Exponent notation. <i>e</i> prints the number in scientific notation using the letter <i>e</i> to indicate the exponent. <i>E</i> is the same, except it uses the letter <i>E</i>
<b>f, F</b>	Floating point. <i>F</i> and <i>f</i> are the same.
<b>g</b>	General format. For a given precision <i>p</i> $\geq 1$ , rounds the number to <i>p</i> significant digits and then formats the result in fixed-point or scientific notation, depending on magnitude. This is the default for numbers
<b>G</b>	Same as <i>g</i> , but upper-cases <i>e</i> , <i>nan</i> , and 'inf'
<b>n</b>	Same as <i>d</i> , but uses locale setting for number separators
<b>o</b>	Octal – converts number to base 8
<b>s</b>	String format. This is the default type for strings
<b>x, X</b>	Hexadecimal – convert number to base 16; A-F match case of <i>x</i> or <i>X</i>
<b>%</b>	Percentage. Multiplies the number by 100 and displays in fixed ( <i>f</i> ) format, followed by a percent sign.

## Field Widths

- Specified as {0:width.precision}
- Width is really minimum width
- Precision is either maximum width or # decimal points

Fields can specify a minimum width by putting a number before the type. If the parameter is shorted than the field, it will be padded with spaces, on the left for numbers, and on the right for strings.

The precision is specified by a period followed by an integer. For strings, precision means the maximum width. Strings longer than the maximum will be truncated. For floating point numbers, precision means the number of decimal places displayed, which will be padded with zeros as needed.

Width and precision are both optional. The default width for all fields is 0; the default precision for floating point numbers is 6.

It is invalid to specify precision for an integer.

## Example

### *fmt\_width.py*

```
name = 'Ann Elk'
value = 10000
airspeed = 22.347
# note: [] are used to show blank space, and are not part of the formatting
print('{:s}'.format(name))      # Default format -- no padding
print('{:10s}'.format(name))    # Left justify, 10 characters wide
print('{:3s}'.format(name))     # Left justify, 3 characters wide, displays entire
string
print('{:3.3s}'.format(name))   # Left justify, 3 characters wide, truncates string to
max width
print()
print('{:8d}'.format(value))     # Right justify, decimal, 8 characters wide (all
numbers are right-justified by default)
print('{:8f}'.format(value))     # Right justify int as float, 8 characters wide
print('{:8f}'.format(airspeed))  # Right justify float as float, 8 characters wide
print('{:.2f}'.format(airspeed)) # Right justify, float, 3 decimal places, no maximum
width
print('{:8.3f}'.format(airspeed)) # Right justify, float, 3 decimal places, maximum
width 8
```

***fmt\_width.py***

```
[Ann Elk]  
[Ann Elk  ]  
[Ann Elk]  
[Ann]  
  
[  10000]  
[10000.000000]  
[22.347000]  
[22.35]  
[ 22.347]
```

# Alignment

- Alignment within field can be left, right, or centered
  - < left align
  - > right align
  - ^ center
  - = right align but put padding after sign

You can align the data to be formatted. It can be left-aligned (the default), right-aligned, or centered. If formatting signed numbers, the minus sign can be placed on the left side.

## Example

### *fmt\_align.py*

```
name = 'Ann'
value = 12345
nvalue = -12345

# note: all of the following print in a field 10 characters wide
print('{0:10s}'.format(name))    # Default (left) alignment
print('{0:<10s}'.format(name))   # Explicit left alignment
print('{0:>10s}'.format(name))   # Right alignment
print('{0:^10s}'.format(name))   # Centered
print()
print('{0:10d} {1:10d}'.format(value, nvalue))    # Default (right) alignment
print('{0:>10d} {1:>10d}'.format(value, nvalue))   # Explicit right alignment
print('{0:<10d} {1:<10d}'.format(value, nvalue))   # Left alignment
print('{0:^10d} {1:^10d}'.format(value, nvalue))   # Centered
print('{0:=10d} {1:=10d}'.format(value, nvalue))   # Right alignment, but pad _after_
sign
```



***fmt\_align.py***

```
[Ann      ]  
[Ann      ]  
[      Ann]  
[  Ann    ]  
  
[  12345] [ -12345]  
[  12345] [ -12345]  
[12345   ] [-12345   ]  
[ 12345  ] [ -12345  ]  
[   12345] [-   12345]
```

## Fill characters

- Padding character must precede alignment character
- Default is one space
- Can be any character except }

By default, if a field width is specified and the data does not fill the field, it is padded with spaces. A character preceding the alignment character will be used as the fill character.

### Example

#### *fmt\_fill.py*

```
name = 'Ann'
value = 123

print('{:>10s}'.format(name))    # Right justify string, pad with space (default)
print('{:.>10s}'.format(name))    # Right justify string, pad with '.'
print('{:~>10s}'.format(name))    # Right justify string, pad with '-'
print('{:~.10s}'.format(name))    # Left justify string, pad with '.'
print()
print('{:10d}'.format(value))      # Right justify number, pad with space (default)
print('{:010d}'.format(value))     # Right justify number, pad with zeroes
print('{:_>10d}'.format(value))    # Right justify, pad with '_' ('>' required)
print('{:+>10d}'.format(value))    # Right justify, pad with '+' ('>' required)
```

***fmt\_fill.py***

```
[      Ann]
[.....Ann]
[-----Ann]
[Ann]

[      123]
[000000123]
[_____123]
[+++++++123]
```

## Signed numbers

- Can pad with any character except `{}`
- Sign can be `+`, `-`, or space
- Only appropriate for numeric types

The sign character follows the alignment character, and can be plus, minus, or space.

A plus sign means always display `+` or `-` preceding non-zero numbers.

A minus sign means only display a sign for negative numbers.

A space means display a `-` for negative numbers and a space for positive numbers.

## Example

### *fmt\_signed.py*

```
values = 123, -321, 14, -2, 0

for value in values:
    print("default: |{:d}|".format(value)) # default (pipe symbols just to show white
    space)
    print()

for value in values:
    print("plus: |{:+d}|".format(value)) # plus sign puts '+' on positive numbers
    (and zero) and '-' on negative
    print()

for value in values:
    print("minus: |{: -d}|".format(value)) # minus sign only puts '-' on negative
    numbers
    print()

for value in values:
    print("space: |{: d}|".format(value)) # space puts '-' on negative numbers and
    space on others
    print()
```

***fmt\_signed.py***

```
default: |123|  
default: |-321|  
default: |14|  
default: |-2|  
default: |0|
```

```
    plus: |+123|  
    plus: |-321|  
    plus: |+14|  
    plus: |-2|  
    plus: |+0|
```

```
minus: |123|  
minus: |-321|  
minus: |14|  
minus: |-2|  
minus: |0|
```

```
space: | 123|  
space: |-321|  
space: | 14|  
space: |-2|  
space: | 0|
```

## Parameter Attributes

- Specify elements or properties in template
- No need to repeat parameters
- Works with sequences, mappings, and objects

When specifying container variables as parameters, you can select elements in the format rather than in the parameter list. For sequences or dictionaries, index on the selector with []. For object attributes, access the attribute from the selector with . (period).

### Example

*fmt\_attr.py*

```
from datetime import date

fruits = 'apple', 'banana', 'mango'
values = [5, 18, 27, 6]
dday = date(1944, 6, 6)
pythons = {'Idle': 'Eric', 'Cleese': 'John', 'Gilliam': 'Terry',
           'Chapman': 'Graham', 'Palin': 'Michael', 'Jones': 'Terry'}

print('{0[0]} {0[2]}'.format(fruits)) # select from tuple
print('{f[0]} {f[2]}'.format(f=fruits)) # named parameter + select from tuple
print()
print('{0[0]} {0[2]}'.format(values)) # Select from list
print()
print('{0[Palin]} {0[Cleese]}'.format(pythons)) # select from dict
print('{names[Palin]} {names[Cleese]}'.format(names=pythons)) # named parameter + select
from dict
print()
print('{0.month}-{0.day}-{0.year}'.format(dday)) # select attributes from date
```

***fmt\_attrib.py***

```
apple mango  
apple mango
```

```
5 27
```

```
Michael John  
Michael John
```

```
6-6-1944
```



# Formatting Dates

- Special formats for dates
- Pull appropriate values from date/time objects

To format dates, use special date formats. These are placed, like all formatting codes, after a colon. For instance, `{0:%B %d, %Y}` will format a parameter (which must be a `datetime.datetime` or `datetime.date`) as "Month DD, YYYY".

## Example

***fmt\_dates.py***

```
from datetime import datetime

event = datetime(2016, 1, 2, 3, 4, 5)

print(event) # Default string version of date
print()

print("Date is {0:%m}/{0:%d}/{0:%y}".format(event)) # Use three placeholders for month,
day, year
print("Date is {:%m/%d/%y}".format(event)) # Format month, day, year with a single
placeholder
print("Date is {:%A, %B %d, %Y}".format(event)) # Another single placeholder format
```

***fmt\_dates.py***

```
2016-01-02 03:04:05
```

```
Date is 01/02/16
```

```
Date is 01/02/16
```

```
Date is Saturday, January 02, 2016
```

Table 51. Date Formats

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

1. When used with the `strptime()` method, the `%f` directive accepts from one to six digits and zero pads on the right. `%f` is an extension to the set of format characters in the C standard (but implemented separately in datetime objects, and therefore always available).
2. When used with the `strptime()` method, the `%p` directive only affects the output hour field if the `%I` directive is used to parse the hour.
3. The range really is 0 to 61; according to the Posix standard this accounts for leap seconds and the (very rare) double leap seconds. The time module may produce and does accept leap seconds since it is based on the Posix standard, but the datetime module does not accept leap seconds in `strptime()` input nor will it produce them in `strftime()` output.
4. When used with the `strptime()` method, `%U` and `%W` are only used in calculations when the day of the week and the year are specified.
5. For example, if `utcoffset()` returns `timedelta(hours=-3, minutes=-30)`, `%z` is replaced with the string `-0330`.

## Run-time formatting

- Use parameters to specify alignment, precision, width, and type
- Use {} placeholders for runtime values for the above

To specify formatting values at runtime, use a {} placeholder for the value, and insert the desired value in the parameter list. These placeholders are numbered along with the normal placeholders.

## Example

### *fmt\_runtime.py*

```
FIRST_NAME = 'Fred'
LAST_NAME = 'Flintstone'
AGE = 35

print("{0} {1}".format(FIRST_NAME, LAST_NAME))

WIDTH = 12
print("{0:{width}s} {1:{width}s}".format( # value of WIDTH used in format spec
    FIRST_NAME,
    LAST_NAME,
    width=WIDTH,
))

PAD = '-'
WIDTH = 20
ALIGNMENTS = ('<', '>', '^')

for alignment in ALIGNMENTS:
    print("{0:{pad}{align}{width}s} {1:{pad}{align}{width}s}".format( # values of PAD,
        WIDTH, ALIGNMENTS used in format spec
        FIRST_NAME,
        LAST_NAME,
        width=WIDTH,
        pad=PAD,
        align=alignment,
    ))
```

### *fmt\_runtime.py*

```
Fred Flintstone
Fred      Flintstone
Fred----- Flintstone-----
-----Fred -----Flintstone
-----Fred----- -----Flintstone-----
```

## Miscellaneous tips and tricks

- Adding commas to large numbers {n:,}
- Auto-converting parameters to strings (!s)
- Non-decimal prefixes

- Adding commas to large numbers {n:,}

You can add a comma to the format to add commas to numbers greater than 999.

Using a format type of !s will call str() on the parameter and force it to be a string.

Using a # (pound sign) will cause binary, octal, or hex output to be preceded by 0b, 0o, or 0x. This is only valid with type codes b, o, and x.

### Example

*fmt\_misc.py*

```
'''Demonstrate misc formatting'''

big_number = 2303902390239

print("Big number: {:,d}".format(big_number)) # Add commas for readability
print()

value = 27

print("Binary: {:#010b}".format(value)) # Binary format with leading 0b
print("Octal:   {:#010o}".format(value)) # Octal format with leading 0o
print("Hex:     {:#010x}".format(value)) # Hexadecimal format with leading 0x
print()
```

***fmt\_misc.py***

Big number: 2,303,902,390,239

Binary: 0b00011011

Octal: 0o00000033

Hex: 0x0000001b



# Appendix D: Python Bibliography

## Data Science

- ***Building machine learning systems with Python***. William Richert, Luis Pedro Coelho. Packt Publishing
- ***High Performance Python***. Mischa Gorlelick and Ian Ozsvald. O'Reilly Media
- ***Introduction to Machine Learning with Python***. Sarah Guido. O'Reilly & Assoc.
- ***iPython Interactive Computing and Visualization Cookbook***. Cyril Rossant. Packt Publishing
- ***Learning iPython for Interactive Computing and Visualization***. Cyril Rossant. Packt Publishing
- ***Learning Pandas***. Michael Heydt. Packt Publishing
- ***Learning scikit-learn: Machine Learning in Python***. Raúl Garreta, Guillermo Moncecchi. Packt Publishing
- ***Mastering Machine Learning with Scikit-learn***. Gavin Hackeling. Packt Publishing
- ***Matplotlib for Python Developers***. Sandro Tosi. Packt Publishing
- ***Numpy Beginner's Guide***. Ivan Idris. Packt Publishing
- ***Numpy Cookbook***. Ivan Idris. Packt Publishing
- ***Practical Data Science Cookbook***. Tony Ojeda, Sean Patrick Murphy, Benjamin Bengfort, Abhijit Dasgupta. Packt Publishing
- ***Python Text Processing with NLTK 2.0 Cookbook***. Jacob Perkins. Packt Publishing
- ***Scikit-learn cookbook***. Trent Hauck. Packt Publishing
- ***Python Data Visualization Cookbook***. Igor Milovanovic. Packt Publishing
- ***Python for Data Analysis***. Wes McKinney. O'Reilly & Assoc

## Design Patterns

- ***Design Patterns: Elements of Reusable Object-Oriented Software***. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley Professional
- ***Head First Design Patterns***. Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. O'Reilly Media
- ***Learning Python Design Patterns***. Gennadiy Zlobin. Packt Publishing
- ***Mastering Python Design Patterns***. Sakis Kasampalis. Packt Publishing

## General Python development

- ***Expert Python Programming***. Tarek Ziadé. Packt Publishing
- ***Fluent Python***. Luciano Ramalho. O'Reilly & Assoc.

- ***Learning Python, 2nd Ed..Mark Lutz, David Asher.*** O'Reilly & Assoc.
- ***Mastering Object-oriented Python.Stephen F. Lott.*** Packt Publishing
- ***Programming Python, 2nd Ed. .Mark Lutz.*** O'Reilly & Assoc.
- ***Python 3 Object Oriented Programming.Dusty Phillips.*** Packt Publishing
- ***Python Cookbook, 3rd. Ed.. David Beazley, Brian K. Jones.*** O'Reilly & Assoc.
- ***Python Essential Reference, 4th. Ed..David M. Beazley.*** Addison-Wesley Professional
- ***Python in a Nutshell.Alex Martelli.*** O'Reilly & Assoc.
- ***Python Programming on Win32.Mark Hammond, Andy Robinson.*** O'Reilly & Assoc.
- ***The Python Standard Library By Example.Doug Hellmann.*** Addison-Wesley Professional

## Misc

- ***Python Geospatial Development.Erik Westra.*** Packt Publishing
- ***Python High Performance Programming.Gabriele Lanaro.*** Packt Publishing

## Networking

- ***Python Network Programming Cookbook.Dr. M. O. Faruque Sarker.*** Packt Publishing
- ***Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers.TJ O'Connor.*** Syngress
- ***Web Scraping with Python.Ryan Mitchell.*** O'Reilly & Assoc.

## Testing

- ***Python Testing Cookbook.Greg L. Turnquist.*** Packt Publishing
- ***Learning Python Testing.Daniel Arbuckle.*** Packt Publishing
- ***Learning Selenium Testing Tools, 3rd Ed. .Raghavendra Prasad MG.*** Packt Publishing

## Web Development

- ***Building Web Applications with Flask.Italo Maia.*** Packt Publishing
- ***Django 1.0 Website Development.Ayman Hourieh.*** Packt Publishing
- ***Django 1.1 Testing and Development.Karen M. Tracey.*** Packt Publishing
- ***Django By Example.Antonio Melé.*** Packt Publishing
- ***Django Design Patterns and Best Practices.Arun Ravindran.*** Packt Publishing
- ***Django Essentials.Samuel Dauzon.*** Packt Publishing

- ***Django Project Blueprints*.Asad Jibran Ahmed**.Packt Publishing
- ***Flask Blueprints*.Joel Perras**.Packt Publishing
- ***Flask by Example*.Gareth Dwyer**.Packt Publishing
- ***Flask Framework Cookbook*.Shalabh Aggarwal**.Packt Publishing
- ***Flask Web Development*.Miguel Grinberg**. O'Reilly & Assoc.
- ***Full Stack Python (e-book only)*.Matt Makai**.Gumroad (or free download)
- ***Full Stack Python Guide to Deployments (e-book only)*.Matt Makai**.Gumroad (or free download)
- ***High Performance Django*.Peter Baumgartner, Yann Malet**.Lincoln Loop
- ***Instant Flask Web Development*.Ron DuPlain**.Packt Publishing
- ***Learning Flask Framework*.Matt Copperwaite, Charles O Leifer**.Packt Publishing
- ***Mastering Flask*.Jack Stouffer**.Packt Publishing
- ***Two Scoops of Django 3.X: Best Practices for the Django Web Framework*. Daniel Roy Greenfeld, Audrey Roy Greenfeld**.Two Scoops Press
- ***Web Development with Django Cookbook*.Aidas Bendoraitis**.Packt Publishing

# Index

## @

- [@name.setter](#), 225
- [@property](#), 225
- [@staticmethod](#), 230
- [\\_\\_call\\_\\_](#)(, 276
- [\\_\\_init\\_\\_](#)(, 276
- [\\_\\_init\\_\\_.py](#), 185
- [\\_\\_new\\_\\_](#)(, 276
- [\\_\\_prepare\\_\\_](#)(, 276
- [\\_\\_pycache\\_\\_](#), 172
- `[]`, 348, 352, 355, 360, 360, 366, 415

## A

- [activate](#), 433
- Anaconda**, 8
- Anaconda, 385
- Anaconda Prompt, 8
- Anaconda prompt, 8
- API, 346
- [argparse](#), 459
- Array types, 83
- Atom**, 15
- attributes, 247
- [autocommit](#), 377

## B

- [backrefs](#), 498
- benchmarking, 315
- [bool](#), 38
- Boolean operators, 73
- break statement, 78
- builtin classes, 20
- builtin functions, 20
  - table, 21

## C

- [callable](#), 254
- CamelCase, 214
- CapWords, 214
- Cassandra, 380
- class
  - constructor, 223

- defining at runtime, 268
- definition, 214
- instance methods, 224
- private method, 231
- static method, 230
- class data, 228
- class data*, 213
- class method, 228
- class object, 228
- class** statement, 214
- classes, 213
- command line arguments, 61
- command line scripts, 455
- command prompt, 8
- comments, 303, 303
- commit, 377
- [complex](#), 38
- conda, 437
- conditional expression*, 70
- connection object, 351
- constructor, 223
- constructor*, 213
- context manager, 348
- continue statement, 78
- contravariant**, 293
- counting, 136
- covariance**, 293
- creating Unix-style filters, 456
- CSV, 415
  - custom, 417
  - dialects, 417
  - parameters, 419
  - reader, 417
  - writer, 417
- csv
  - DictReader, 420
- csv.writer(), 422
- cursor, 351
- cursor object, 351
- cx\_oracle, 347

**D**

- database programming, [346](#)
- database server, [348](#)
- DB API, [346](#)
- deactivate**, [434](#)
- debugger
  - setting breakpoints, [312](#)
  - starting, [310](#)
  - stepping through a program, [312](#)
- decorator arguments, [266](#)
- decorator class, [263](#)
- decorator function, [260](#)
- decorators, [254](#)
- decorators in the standard library, [255](#)
- delattr(), [247](#)
- delete, [339](#)
- derived class, [232](#)
- dict()**, [137](#)
- dictionary, [136](#)
  - adding elements, [137](#)
  - counting with, [147](#)
  - getting values, [140](#)
  - iterating over, [143](#)
  - reading file into, [145](#)
- dictionary cursor
  - emulating, [374](#)
- dictionary cursors, [369](#)
- dictionary functions, [139](#)
- Django, [379](#)
- Django framework, [276](#)
- Django ORM, [379](#)
- Douglas Crockford, [401](#)

**E**

- Eclipse**, [15](#)
- editors and IDEs, [15](#)
- Element**, [387](#)
- Element, [386](#)
- ElementTree, [385](#)
  - find(), [394](#)
  - findall(), [394](#)
- enumerate(), [107](#)
- Escape characters, [23](#)
- escape codes, [24](#)

- escape sequences
  - table, [31](#)
- exceptions*, [191](#)
- exceptions, [191](#)
  - else, [196](#)
  - finally, [198](#)
  - generic, [194](#)
  - ignoring, [195](#)
  - list, [201](#)
  - multiple, [193](#)
- executing SQL statements, [352](#)
- exponentiation, [40](#)

**F**

- file modes
  - table, [124](#)
- file(), [131](#)
- files
  - opening, [129](#)
- Firebird (and Interbase, [347](#)
- floats, [38](#)
- floored division, [40](#)
- flow control, [66](#)
- for loop, [93](#)
- format, [49](#)
- format strings, [50](#)
- formatting, [49](#)
- function parameters, [160](#)
  - default, [164](#)
  - named, [160](#)
  - optional, [160](#)
  - positional, [160](#)
  - required, [160](#)
- functions, [158](#)
- functools**, [407](#)
- functools.wraps, [260](#)

**G**

- generator expressions, [115](#)
- getattr(), [247](#)
- getroot(), [393](#)
- glob**, [440](#)
- global, [170](#)
- globals(), [240](#)

**H**

hasattr(), 247  
head, 339  
help(), 13  
http options, 339

**I**

IBM DB2, 347  
ibm-db, 347  
if statement, 67  
if-else, 70  
if/elif/else, 67  
import \*, 178  
import {star}, 177  
in, 103  
indentation, 68  
indexing, 90  
Informix, 347  
informixdb, 347  
ingmod, 347  
Ingres, 347  
inheritance, 232  
initializer, 213  
inspect module, 243  
instance data, 213  
instance methods, 224  
integers, 38  
interactive prompt, 10  
interpreter, 8  
**invariant**, 294  
**IPython**, 11  
iterable, 98  
iterating through a sequence, 93

**J**

JSON, 401  
    custom encoding, 407  
    types, 401  
json module, 402  
JSON response, 324  
json.dumps(), 405  
json.loads(), 402

**K**

key/value pairs, 137  
keywords, 20  
KInterbasDB, 347

**L**

legacy string formatting, 56  
len(), 105  
**Liskov Substitution Principle**, 293  
list comprehension, 112  
list methods  
    table, 86  
lists, 85  
literal dictionary constructor, 137  
literal numbers, 38  
literal Unicode characters, 29  
locals(), 240  
logging  
    alternate destinations, 471  
    exceptions, 208, 469  
    formatted, 204, 466  
    simple, 202, 464  
lxml  
    Element, 387  
    SubElement, 387  
**lxml.etree**, 384

**M**

mapping, 136  
math operators  
    table, 42  
math operators and expressions, 40  
max(), 105  
metaclass, 275, 276  
metaclasses, 274  
metadata, 366  
metaprogramming, 239  
Microsoft SQL Server, 347  
min(), 105  
modules, 171, 171  
    documenting, 187  
    importing, 172  
    search path, 179  
MongoDB, 380

monkey patches, [271](#)

**mypy**, [287](#)

MySQL, [347](#)

## N

namedtuple cursor, [374](#)

nested sequences, [101](#)

non-printable characters, [23](#)

non-query statement, [360](#)

non-relational, [380](#)

None, [19](#)

nonlocal, [170](#)

NoSQL, [380](#)

**Notepad++**, [15](#)

numeric literals, [38](#)

## O

object-oriented language, [213](#)

object-oriented programming, [213](#)

Object-relational mapper, [379](#)

*objects*, [213](#)

ODBC, [347](#)

opening files, [123](#), [129](#)

optional type, [295](#)

Oracle, [347](#)

order of operations, [40](#)

ORM, [379](#)

## P

packages, [182](#)

    configuring, [185](#)

parameterized SQL statements, [360](#)

parsing the command line, [460](#)

patch, [339](#)

**PATH**, [9](#)

**PATH** variable, [8](#)

**PATH** variable, [9](#)

PEP 8, [188](#)

**Perl**, [475](#)

permissions, [451](#)

    checking, [451](#)

**pip freeze**, [435](#)

pipenv, [437](#)

placeholder, [360](#)

polymorphic, [213](#)

Popen, [443](#)

**POST**, [336](#)

PostgreSQL, [347](#)

**postman**, [343](#)

preconfigured log handlers, [471](#)

print() function, [47](#)

private data, [223](#)

private methods, [231](#)

profiler, [313](#)

properties, [225](#)

psycopg, [347](#)

put, [339](#)

pycallgraph, [314](#)

PyCharm, [437](#)

**PyCharm Community Edition**, [15](#)

pychecker, [304](#)

**pydoc**, [13](#)

pyflakes, [304](#)

pylint, [304](#)

    customizing, [305](#)

pymssql, [347](#)

pymysql, [347](#)

pyodbc, [347](#)

pyreverse, [306](#)

**pyreverse**, [306](#)

python debugger, [309](#)

Python Interpreter, [10](#)

Python script, [12](#)

python style, [188](#)

Python style guide, [19](#)

Python Wiki, [15](#)

**python3**, [8](#)

PYTHONPATH, [179](#)

## R

raw strings, [27](#)

re.compile(), [481](#)

re.findall(), [478](#)

re.finditer(), [478](#)

re.search(), [478](#)

read(), [131](#)

Reading from the keyboard, [63](#)

reading text files, [125](#)

readline(), [131](#)

- `readlines()`, 131
- Redis, 380
- Regular Expression Metacharacters
  - table, 477
- regular expressions, 475, 475
  - about, 475
  - atoms, 476
  - branches, 476
  - compilation flags, 485
  - finding matches, 478
  - grouping, 491
  - re objects, 481
  - replacing text, 496
  - replacing text with callback, 501
  - special groups, 494
  - splitting text, 503
  - syntax overview, 476
- Relational Operators, 71
- requests
  - methods
    - keyword parameters, 327
  - Response
    - attributes, 328
- requirements.txt**, 435
- `reversed()`, 105
- rollback, 377
- Running Python scripts, 12

## S

- SAP DB, 347
- sapdbapi, 347
- scope
  - builtin, 168
  - global, 168
  - nonlocal, 168
- `self`, 224
- sequence functions, 105
- Sequences, 83
- set, 149
  - creating, 150
  - functions and methods (table), 154
  - operations, 151
- `setattr()`, 247
- sets, 149

- shlex.split()**, 442
- shutil**, 453
- singledispatch**, 407
- slicing, 90
- `sorted()`, 105
- Spyder**, 15
- SQL code, 351
- SQL data integrity, 377
- SQL injection*, 358
- SQL queries, 352
- SQLAlchemy, 379
- SQLite, 347
- sqlite3, 347
- standard exception hierarchy, 201
- starting python, 8
- static method, 230
- string formatting, 49
  - alignment, 519
  - data types, 513
  - dates, 529
  - field widths, 516
  - fill characters, 522
  - misc, 535
  - parameter attributes, 527
  - run-time, 533
  - selectors, 510
  - signed numbers, 524
- string methods, 32
- string operators, 32
- Strings, 23
- strings
  - literal, 24
  - single-delimited, 24
- StudyCaps, 214
- SubElement**, 387
- Sublime**, 15
- subprocess**, 443
- subprocess, 444
  - capturing stdout/stderr, 447
  - `check_call()`, 444
  - `check_output()`, 444
  - `run()`, 444
- `sum()`, 105
- Sybase, 347, 347



`sys.path`, [179](#)

## T

tagged union, [295](#)

terminal window, [8](#)

ternary operator, [70](#)

`timeit`, [315](#)

**transactions**, [377](#)

triple-delimited strings, [25](#)

triple-quoted strings, [25](#)

tuple unpacking, [100](#)

tuples, [96](#)

type, [275](#)

type conversions, [43](#)

type hinting, [286](#)

`type()`, [268](#)

## U

Unicode, [23](#)

Unicode characters, [29](#)

UpperCamelCase, [214](#)

using try/except, [192](#)

## V

variable names, [19](#)

variable scope, [168](#)

variable typing, [22](#)

Variables, [19](#)

variance, [292](#)

**venv**, [432](#)

virtual environment, [431](#)

`virtualenv`, [437](#)

`virtualenvwrapper`, [437](#)

**Visual Studio Code**, [15](#)

## W

while loop, [77](#)

whitespace, [68](#)

`write()`, [131](#)

`writelines()`, [131](#)

## X

XML, [384](#)

    root element, [390](#)

**xml.etree.ElementTree**, [384](#)

`xml.etree.ElementTree`, [385](#)

**XPath**, [398](#)

## Z

`zip()`, [105](#)