# Getting Started with Python Programming Basics (for Non-Developers)

## Student Guide

**TTPS4803-APL3**

## Trivera Technologies LLC

Collaborative IT Training, Mentoring & Courseware Solutions

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

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

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

**Trivera Technologies Services Include**

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

### Trivera Technologies

**Educate. Collaborate. Accelerate!**

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

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

TriveraTech TECHNOLOGY TRAINING

***100% Woman-Owned Small Business***

### A Sample of our Course Offerings includes....

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

**> Agile & TDD:** Agile Development • TDD / BDD • Unit Testing • Frameworks • Scrum • SAFE • Kanban

**> Web Development & Design:** HTML5 / CSS3 • JavaScript • JQuery • Angular • React / Redux • MEANStack / MERNStack • Node.JS • UX / UI • BootStrap • XML / XSLT • Responsive Design • Python • Perl • PHP • XML

**> Python:** Python for Web • Python Networking / Sys Admin • Python Data Science • Python for Security Pros

**> Security**: CyberSecurity • Secure Software Design • Secure Coding • Secure Web Development • Database Security • OWASP • STIG • CCSK

**> Mobile**: Android • IOS • PhoneGap • Kotlin • Swift • Mobile Application Testing • Secure Mobile Development

**> Big Data / Data Science, AI , Machine Learning, & Deep Learning:** Machine Learning • Hadoop Admin • Hadoop Development • Pig / Hive / MapReduce • Scala • Spark • R for Data Science • Data Science • Python for Data Science • AI – Artifical Intelligence

**> SOA:** SOA • Architecture • Analysis • Design • Governance • Implementation • SOA Security

**> Databases**: DB Design • DB Security • DB2 • SQL • PLSQL • Oracle • MySQL • SQLServer • MongoDB • NoSQL • MariaDB • Cassandra • Oracle • JDBC

**> MainFrame**: C • COBOL • DB2 • Assembler • Transition to Web • Java for COBOL

**> O/S & SysAdmin:** Windows • Linux • UNIX • Mainframe • Z/OS • Administration • IOS

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

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

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

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

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

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

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

This page has been left intentionally blank.

# Table of Contents

# About this course

# Welcome!

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

**Instructor name:**

**Instructor e-mail:**

## Have Fun!

# Classroom etiquette for 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.

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

# Classroom etiquette for remote learning

- Please turn your mic off when you’re not speaking. If multiple mics are on, it makes it difficult for all to hear
- The instructor doesn’t know you need help unless you tell them. It’s ok to ask for help often.
- Ask questions. Ask questions. Ask questions.
- ***INTERACT*** with the instructor and other students.
- Log off the remote software (**Zoom**, **Webex**, etc.) at the end of the day.
- Ask questions.

# Course Outline

## Day 1

Chapter 1 An overview of Python
Chapter 2 The Python environment
Chapter 3 Getting started
Chapter 4 Flow control

## Day 2

Chapter 5 Array types
Chapter 6 Working with files
Chapter 7 Dictionaries and sets
Chapter 8 Functions

## Day 3

Chapter 9 Sorting
Chapter 10 Errors and exception handling
Chapter 11 Creating and using modules
Chapter 12 Introduction to Python classes

**NOTE** 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 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 **py3intro3day**.

What’s in the files?

**py3intro3day** contains data and other files needed for the exercises
**py3intro3day/EXAMPLES** contains the examples from the course manuals.
**py3intro3day/ANSWERS** contains sample answers to the labs.

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

# Extracting the student files

## Windows

Open the file **py3intro3day.zip**. Extract all files to your desktop. This will create the folder **py3intro3day**.

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

Copy or download **py3intro3day.tgz** to your home directory. In your home directory, type

```
tar xzvf py3intro3day.tgz
```

This will create the **py3intro3day** directory under your home directory.

# Examples

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

It will look like this:

## Example*

**cmd_line_args.py**

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

import sys   ①

print(sys.argv) ②

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

① Import the **sys** module

② Print all parameters, including script itself

③ Get the first actual parameter

***cmd_line_args.py Fred***

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

# Lab Exercises

- Relax – the labs are not quizzes
- Feel free to modify labs
- Ask the instructor for help
- Work on your own scripts or data
- Answers are in py3intro3day/ANSWERS

# Appendices

- Appendix A: Where do I go from here?
- Appendix B: Python Bibliography
- Appendix C: String formatting

# Chapter 1: An Overview of Python

## Objectives

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

# What is Python?

- All-purpose interpreted language
- Created by Guido van Rossum
- First released (ver. 0.9) February 20, 1991

**Python** is an open-source, all-purpose programming language.

Python was created by Guido van Rossum beginning in 1989. He was involved with the development of Amoeba, a distributed operating system, and had previously worked on ABC, a scripting language designed to be easier to learn for non-programmers.

Van Rossum took ABC and improved it, adding new features, some of which came from other languages such as Perl and Lisp. His design goal was

> to serve as a second language for people who were C or C++ programmers, but who had work where writing a C program was just not effective.

The first public release was version 0.9 (beta) in 1991.

*Guido van Rossum*

# The Birth of Python

About the origin of Python, Van Rossum wrote in 1996:

> Over six years ago, in December 1989, I was looking for a "hobby" programming project that would keep me occupied during the week around Christmas. My office … would be closed, but I had a home computer, and not much else on my hands. I decided to write an interpreter for the new scripting language I had been thinking about lately: a descendant of ABC that would appeal to Unix/C hackers. I chose Python as a working title for the project, being in a slightly irreverent mood (and a big fan of Monty Python's Flying Circus). (Introduction to Programming Python, by Mark Lutz, published by O'Reilly)
>
> He says, "The first sound bite I had for Python was, 'Bridge the gap between the shell and C.'".
>
> — Guido van Rossum

*Monty Python in 1970*

*Table 1. Python Timeline*

| Year | Event | Notes |
|---|---|---|
| 1969 | "Monty Python's Flying Circus" premieres on the BBC | |
| 1991 | version 0.9.0 (first release) | classes, try/except, lists, dictionaries, modules |
| 1992 | version 0.9.6 ported to MS-DOS | |
| 1993 | comp.lang.python created | |
| 1994 (Jan) | version 1.0 | lambda, reduce(), filter() and map() |
| 1995 | version 1.4 | keyword arguments, complex numbers |
| 1997 (Dec 31) | version 1.5 | |
| 2000 (Oct 16) | version 2.0 final release | list comprehensions |
| 2002 (Dec 21) | version 2.2 | unification of types and classes |
| 2004 (Nov 30) | Version 2.4 | |
| 2006 (Sept 19) | version 2.5 | |
| 2008 (Oct 1) | version 2.6 | backward-compatible with 2.5 |
| 2008 (Dec 3) | version 3.0 | removing old features and adding new features (not backward-compatible with Python 2) |
| 2009 (June 27) | Version 3.1 | Backward-compatible with 3.0; new features/modules; deprecated modules |
| 2018 (July 3) | Version 2.7 | Final 2.x version |
| 2012 (Sept 29) | Version 3.3 | |
| 2014 (Mar 16) | Version 3.4 | pip is included, pathlib |
| 2015 (Sept 13) | Version 3.5 | subprocess.run |
| 2016 (Dec 23) | Version 3.6 | f-strings, variable annotations |
| 2018 (June 27) | Version 3.7 | data classes |
| 2019 (Oct 14) | Version 3.8 | assignment expressions, pos-only parameters |
| 2020 (Oct 5) | Version 3.9 | |
| 2021 (Oct 4) | Version 3.10 | structural pattern matching |

# About Interpreted Languages

- Python is an interpreted language
- The Python interpreter reads a script and interprets it on-the-fly.
- Since there is no compile phase, the development cycle can be very rapid

Like Perl, Ruby, and Bash, Python is an *interpreted* language. The program consists of a text file containing Python commands. To run the program, you run the interpreter (normally called "python.exe", "python", *etc.*) and tell it which file contains the commands.

# Advantages of Python

- Clear, readable syntax
- Multi-paradigm
  - object-oriented programming
  - procedural
  - functional
- Dynamic data structures (e.g., lists and dictionaries)
- Exception-based error handling
- Code can be organized into modules and packages
- Extensive standard library and third party modules
- Fun!!

# Disadvantages of Python

# How to get Python

- Download from **www.python.org**
- Versions available for most operating systems
- **Anaconda** is superset of standard Python

The latest version of Python is always available via the Python home page. http://www.python.org/download will direct you to the latest binaries.

The above URL has Windows MSI installation files.

Linux and OS X come with Python.

For scientific and engineering tasks, the **Anaconda** bundle is a great choice. It contains the Python interpreter plus hundreds of libraries in addition to the standard modules. Among others, it contains **NumPy**, **SciPy**, **pandas**, **iPython**, and **Matplotlib**. Even if you're not doing scientific programming, it includes **Requests**, **PyQt**, **OpenPyxl**, and many other useful modules.

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

# Which version of Python?

- python -V displays current version

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

Check https://www.python.org for the latest release.

```
$ python -V
Python 3.10.0
```

# The end of Python 2

Python 2.7 is intended to be the last minor release in the 2.x series. The Python maintainers are planning to focus their future efforts on Python 3.

This means that 2.7 will continue to run production systems that have not been ported to Python 3. Two consequences of the long-term significance of 2.7 are:

1. It's very likely the 2.7 release will have a longer period of maintenance compared to earlier 2.x versions. Python 2.7 will continue to be maintained while the transition to 3.x continues, and the developers are planning to support Python 2.7 with bug-fix releases beyond the typical two years.
2. A policy decision was made to silence warnings only of interest to developers. DeprecationWarning and its descendants are now ignored unless otherwise requested, preventing users from seeing warnings triggered by an application. This change was also made in the branch that will become Python 3.2. (Discussed on stdlib-sig and carried out in issue 7319.)

```
-- from the Python 2.7 documentation_
```

At PyCon 2014 in Montreal, Guido van Rossum extended the end-of-life date for 2.7 to 2020. More recently, the end-of-life date for Python 2.7 has been established on January 1, 2020. This means that there will be no more support from the core Python developers, including bug fixes.

# Getting Help

- Books
- Web sites
- pydoc

There are many ways of getting help with Python. The bibliography at the end of this course lists some of the best Python books.

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

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

# Chapter 2: The Python Environment

## Objectives

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

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

# Starting Python

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

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

- Python is not installed on your computer
- The interpreter (`python`) is not in your PATH variable

# If the interpreter is not in your PATH

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

## Type the full path

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

## Add the directory to PATH temporarily

### Windows (at a command prompt)

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

### Linux/Mac

```
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 interpreter

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

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

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

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

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

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

*Table 2. emacs-style command line editing*

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

# Trying out a few commands

Try out the following commands in the interpreter:

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

You don’t really need **print()**

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

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

| | |
|---|---|
| **NOTE** | If you have **ipython** installed, use **ipython** for a better interactive interpreter (**ipython** is included with Anaconda). |

# Running Python scripts

- Use Python interpreter
- Same for any OS

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

```
python myscript.py
```

This will work on any operating system.

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

# Using pydoc

## From the Python interpreter

Type

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

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

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

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

## From a command line

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.
```

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

**TIP** Run pydoc -k <keyword> to search packages by keyword

## From iPython

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

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

# Python Editors and IDEs

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

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

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

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

# Chapter 2 Exercises

## Exercise 2-1 (hello.py)

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

Run the script from the command line.

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

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

# Chapter 3: Getting Started

## Objectives

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

# Using variables

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

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

Variable names are composed of letters, digits, and underscores, and may not start with a digit. Any Unicode character that corresponds to a letter or digit may also be used.

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

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

```
all_lower_case_with_underscores
```

## Example

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

# Keywords and Builtins

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

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

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

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

## Python 3 Keywords

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

*Table 3. Builtin functions*

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

*These functions are class constructors

# Variable typing

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

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

Certain operations are only valid with the appropriate types.

| | |
|---|---|
| **WARNING** | Python does not automatically convert strings to numbers or numbers to strings. |

# Strings

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

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

# Single-delimited string literals

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

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

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

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

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

**TIP** Adjacent string literals are concatenated.

## Example

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

# Triple-delimited string literals

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

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

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

## Example

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

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

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

# Raw string literals

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

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

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

## Example

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

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

# Unicode characters

- Use \uXXXX to specify non-ASCII Unicode characters
- XXXX is Unicode value in hex

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

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

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

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

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

## Example

### unicode.py

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

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

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

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

① Use \uXXXX where XXXX is the Unicode value in hex

② The Unicode entity name can be used, enclosed in \N{}

③ \N{} is not expanded in raw strings

④ More examples.

⑤ Python answers the age-old question.

*unicode.py*

```
26°
26°
26\u00B0\n

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

unsorted: ['🥚', '🐔']
sorted: ['🐔', '🥚']
```

*Table 4. Escape Sequences*

| Sequence | Description |
| --- | --- |
| *newline* | Embedded newline |
| \\\\ | Backslash |
| \\' | Single quote |
| \\" | Double quote |
| \\a | BEL |
| \\b | BACKSPACE |
| \\f | FORMFEED |
| \\n | LINEFEED |
| \\N{name} | Unicode named code point *name* |
| \\r | Carriage Return |
| \\t | TAB |
| \\uxxxx | 16-bit Unicode code point |
| \\Uxxxxxxxx | 32-bit Unicode code point (for values above 0xFFFF) |
| \\ooo | Char with octal ASCII value ooo |
| \\xhh | Character with hex ASCII value hh |

# String operators and methods

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

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

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

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

Use + (plus) to concatenate two strings.

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

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

String methods may be called on literal strings as well

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

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

## Example

### strings.py

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

a = "My hovercraft is full of EELS"

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

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

① Swap upper and lower case

② All words are capitalized

③ Methods can be chained. The next method is called on the object returned by the previous method.

④ Returns -1 if substring not found

⑤ Capitalizes first character of string, only if it is a letter

***strings.py***

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

# String Methods

*Table 5. string methods*

| Method | Description |
| --- | --- |
| S.capitalize() | Return a capitalized version of S, i.e. make the first character have upper case and the rest lower case. |
| S.casefold() | Return a version of S suitable for caseless comparisons. |
| S.center(width[, fillchar]) | Return S centered in a string of length width. Padding is done using the specified fill character (default is a space) |
| S.count(sub, [, start[, end]]) | Return the number of non-overlapping occurrences of substring sub. Optional arguments start and end specify a substring to search. |
| S.encode(encoding='utf-8', errors='strict') | 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 UnicodeEncodeError. Other possible values are 'ignore', 'replace' and 'xmlcharrefreplace' as well as any other name registered with codecs.register_error that can handle UnicodeEncodeErrors. |
| S.endswith(suffix[, start[, end]]) | 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. |
| S.expandtabs(tabsize=8) | 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. |
| S.find(sub[, start[, end]]) | 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. |
| S.format(*args, **kwargs) | Return a formatted version of S, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}'). |
| S.format_map(mapping) | Return a formatted version of S, using substitutions from mapping. The substitutions are identified by braces ('{' and '}'). |
| S.index(sub[, start[, end]]) | Like find() but raise ValueError when the substring is not found. |
| S.isalnum() | Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise. |
| S.isalpha() | Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise. |
| S.isdecimal() | Return True if there are only decimal characters in S, False otherwise. |

| Method | Description |
| --- | --- |
| S.isdigit() | Return True if all characters in S are digits and there is at least one character in S, False otherwise. |
| S.isidentifier() | Return True if S is a valid identifier according to the language definition. |
| S.islower() | Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise. |
| S.isnumeric() | Return True if there are only numeric characters in S, False otherwise. |
| S.isprintable() | Return True if all characters in S are considered printable in repr() or S is empty, False otherwise. |
| S.isspace() | Return True if all characters in S are whitespace and there is at least one character in S, False otherwise. |
| S.istitle() | Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones. Return False otherwise. |
| S.isupper() | Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise. |
| S.join(iterable) | 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 |
| S.ljust(width[, fillchar]) | Return S left-justified in a Unicode string of length width. Padding is done using the specified fill character (default is a space). |
| S.lower() | Return a copy of the string S converted to lowercase. |
| S.lstrip([chars]) | Return a copy of the string S with leading whitespace removed. If chars is given and not None, remove characters in chars instead. |
| S.partition(sep) | 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. |
| S.replace(old, new[, count]) | 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. |
| S.rfind(sub[, start[, end]]) | 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. |
| S.rindex(sub[, start[, end]]) | Like rfind() but raise ValueError when the substring is not found. |
| S.rjust(width[, fillchar]) | Return S right-justified in a string of length width. Padding is done using the specified fill character (default is a space). |

| Method | Description |
|---|---|
| S.rpartition(sep) | 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 |
| S.rsplit(sep=None, maxsplit=-1) | 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. |
| S.rstrip([chars]) | Return a copy of the string S with trailing whitespace removed. If chars is given and not None, remove characters in chars instead. |
| S.split(sep=None, maxsplit=-1) | 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. |
| S.splitlines([keepends]) | 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. |
| S.startswith(prefix[, start[, end]]) | 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. |
| S.strip([chars]) | Return a copy of the string S with leading and trailing whitespace removed. If chars is given and not None, remove characters in chars instead. |
| S.swapcase() | Return a copy of S with uppercase characters converted to lowercase and vice versa. |
| S.title() | Return a titlecased version of S, i.e. words start with title case characters, all remaining cased characters have lower case. |
| S.translate(table) | Return a copy of the string S, where all characters have been mapped through the given translation table, which must be a mapping of Unicode ordinals to Unicode ordinals, strings, or None. Unmapped characters are left untouched. Characters mapped to None are deleted. |
| S.upper() | Return a copy of S converted to uppercase. |
| S.zfill(width) | Pad a numeric string S with zeros on the left, to fill a field of the specified width. The string S is never truncated. |

# Numeric literals

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

## Boolean

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

## Integers

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

## Floats

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

## Complex Numbers

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

## Example

### numeric.py

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

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

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

① Octal

② Hex

③ Binary

***numeric.py***

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

# Math operators and expressions

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

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

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

## Division

Division (/) always returns a float result.

## Assignment-with-operation

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

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

is equivalent to

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

## Exponentiation

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

## Floored Division

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

## Order of operations

*P*lease *E*xcuse *M*y *D*ear *A*unt *S*ally!

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

## Example

### math_operators.py

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

x = 22
x += 10  ①

y = 5
y *= 3  ①

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

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

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

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

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

### *math_operators.py*

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

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

*Table 6. Python Math Operators and Functions*

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

# Converting among types

- No automatic conversion between numbers and strings
- Builtin functions
  - int() convert string or number to integer
  - float() convert string or number to float
  - str() convert anything to string
  - bool() convert anything to bool
  - list() convert any iterable to a list
  - tuple() convert any iterable to a tuple
  - set() convert any iterable to a set
  - dict() convert any iterable of pairs to a dict

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

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

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

# Writing to the screen

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

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

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

## Example

**print_examples.py**

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

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

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

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

x = "Hello"
y = "world"

print(x, y)  ③
print("#------------------------")

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

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

① Print space instead of newline at the end

② Print bang instead of newline at end

③ Item separator is space instead of comma

④ Item separator is comma + space

⑤ Item separator is empty string

***print_examples.py***

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

# String Formats

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

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

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

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

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

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

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

See [string_formatting] for details on formatting.

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

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

## Example

### string_formatting.py

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

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

print("Name is [{:<10s}]".format(name))  ①
print("Name is [{:>10s}]".format(name))  ②
print("count is {:03d} avg is {:.2f}".format(count, avg))  ③

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

print("${:,d}".format(38293892))  ⑥

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

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

② > means right justify

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

④ d is decimal, o is octal, x is hex

⑤ # means add 0x, 0o, etc.

⑥ , means add commas to numeric value

⑦ parameters can be selected by name instead of position :b ***string_formatting.py***

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

# 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**

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

person = 'Bob'
age = 22

print(f"{person} is {age} years old.")
print(f"The {age}-year-old is {person}.")
print()
```

*fmt_fstrings.py*

```
Bob is 22 years old.
The 22-year-old is Bob.
```

# Legacy String Formatting

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

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

```
%flagW.Ptype
```

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

The placeholders are similar to standard formatting, but are positional-only, and are specified with a percent symbol **(%)**, rather than braces.

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

*Table 7. Legacy formatting types*

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

*Table 8. Legacy formatting flags*

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

## Example

### string_formatting_legacy.py

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

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

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

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

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

① Dash means left justify string

② Right justify (default)

③ Argument to % is either a single variable or a tuple

④ Arguments must be repeated to be used more than once

⑤ Obscure way of doing the same thing Note: (x,) is singleton tuple

⑥ # means add 0x, 0o, etc.

***string_formatting_legacy.py***

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

# Command line arguments

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

To get the command line 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 argument to your script.

## Example

**sys_argv.py**

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

import sys

print(sys.argv)
print()

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

① First command line parameter

***sys_argv.py Gawain***

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

name is Gawain
```

**TIP** If you use an index for a non-existent 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, and it returns the text that was entered. You can use int() or float() to convert the input to an integer or a floating-point number.

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

## Example

### keyboard_input.py

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

user_name = input("What is your name: ")
quest = input("What is your quest? ")
print("{} seeks {}".format(user_name, quest))

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

print("2 times {} is {}".format(num, 2 * num))
```

① input is always a string

② convert to numbers as needed

***keyboard_input.py***

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

To run the script at a command prompt:

```
python c2f.py
```

*(or run from PyCharm/VS Code/Spyder etc)*

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

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

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

## Exercise 3-2 (c2f_batch.py)

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

To run the script at a command prompt:

```
python c2f_batch.py 100
```

*(or run from PyCharm/VS Code/Spyder etc)*

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

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

## Exercise 3-3 (string_fun.py)

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

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

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

# Chapter 4: Flow Control

## Objectives

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

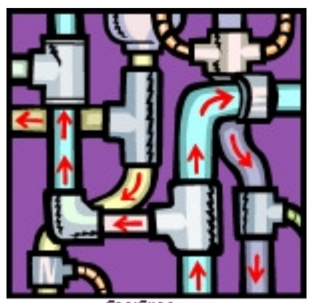
# About flow control

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

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

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

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



CoolClips.com

# What's with the white space?

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

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

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

## Example

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

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

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

# if and elif

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

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

### Example

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

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

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

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

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

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

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

**WARNING**

Don't say

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

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

```
if x:
```

# Conditional Expressions

- Used for simple if-then-else conditions

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

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

This is a shortcut for

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

## 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**.

**NOTE** Strings and numbers cannot be compared using any of the greater-than or less-than operators. Also, no string is equal to any number.

## Example

### if_else.py

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

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

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

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

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

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

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

*if_else.py*

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

# Boolean operators

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

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

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

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

*Table 9. Boolean Operators*

| Expression | Value |
|---|---|
| **AND** | |
| 12 and 5 | 5 |
| 5 and 12 | 12 |
| 0 and 12 | 0 |
| 12 and 0 | 0 |
| "" and 12 | "" |
| 12 and "" | "" |
| **OR** | |
| 12 or 5 | 12 |
| 5 or 12 | 5 |
| 0 or 12 | 12 |
| 12 or 0 | 12 |
| "" or 12 | 12 |
| 12 or "" | 12 |

# while loops

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

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

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

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

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

# Alternate ways to exit a loop

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

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

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

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

## Example

**while_loop_examples.py**

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

print("Welcome to ticket sales\n")

while True:  ①
    raw_quantity = input("Enter quantity to purchase (or q to quit): ")
    if raw_quantity == '':
        continue  ②
    if raw_quantity.lower() == 'q':
        print("goodbye!")
        break  ③

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

① Loop "forever"

② Skip rest of loop; start back at top

③ Exit loop

***while_loop_examples.py***

```
Welcome to ticket sales

Enter quantity to purchase (or q to quit):  4
sending 4 ticket(s)
Enter quantity to purchase (or q to quit):
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".

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

## Exercise 4-2 (guess.py)

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

**TIP**

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

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

## Exercise 4-3 (guessx.py)

Get the maximum number from the command line *or* prompt the user to input the maximum, or both (if no value on command line, 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. Some of these are array types. These hold values in a sequence, such that they can be retrieved by a numerical index.

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

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

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

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

## Example

### typical_arrays.py

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

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

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

### *typical_arrays.py*

```
kiwi
i
Arthur
```

# 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 with the `list()` class or a pair of square brackets. A list can be Initialized with a comma-separated list of values.

*Table 10. List Methods (note L represents a list)*

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

## Example

### creating_lists.py

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

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

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

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

print("list4[-1]:", list4[-1])  ⑦
```

① Create new empty list

② Initialize list

③ Create new empty list

④ Create list of strings with less typing

⑤ First element of **list2**

⑥ Third element of **list4**

⑦ *Last* element of **list4**

***creating_lists.py***

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

# Indexing and slicing

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

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

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

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

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

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

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

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

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

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

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

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

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

The general syntax for a slice is

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

which means all elements s[N], where

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

and start is incremented by step

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

## Example

### indexing_and_slicing.py

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

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

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

phrase = "She turned me into a newt"

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

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

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

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

① First element

② Sixth element

③ First 3 elements

④ Third element through the end

⑤ First 2 elements

⑥ Second through next-to-last element

⑦ Every other element, starting with first

⑧ Every other element, starting with second

⑨ Last element

***indexing_and_slicing.py***

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

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

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

# Iterating through a sequence

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

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

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

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

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

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

## Example

### iteration_over_arrays.py

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

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

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

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

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

① Iterate over elements of list

② Iterate over elements of tuple

③ Iterate over characters of string

***iterating_over_arrays.py***

```
Idle
Cleese
Chapman
Gilliam
Palin
Jones

Roger
Old Woman
Prince Herbert
Brother Maynard

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

# Tuples

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

Python has a second 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.

**NOTE**

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

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

**TIP**

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

## Example

### creating_tuples.py

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

birth_date = 1901, 5, 5

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

latlon = 35.99, -72.390

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

***creating_tuples.py***

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

**TIP**

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

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

# Iterable Unpacking

- Copy elements to variables
- Works with any 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**

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

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

x, y, z = values  ①

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  ② ③
    print(first_name, last_name)
print()

for first_name, last_name, _ in people:  ④
    print(first_name, last_name)
print()

# extended unpacking
values = ['a', 'b', 'c', 'd', 'e', 'f']
x, y, *z = values
print(x, y, z)

x, *y, z = values
print(x, y, z)

*x, y, z = values
print(x, y, z)
```

① unpack values (which is an iterable) into individual variables

② unpack **row** into variables

③ _ is used as a "junk" variable that won't be used

④ a **for** loop unpacks if there is more than one variable

*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

a b ['c', 'd', 'e', 'f']
a ['b', 'c', 'd', 'e'] f
['a', 'b', 'c', 'd'] e f
```

# Nested sequences

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

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

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

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

## Example

### nested_sequences.py

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

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

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

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

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

① person is a tuple

② unpack person into variables

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

***nested_sequences.py***

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

# Operators and keywords for sequences

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

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

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

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

+ adds one sequence to another

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

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

## Example

### sequence_operators.py

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

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

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

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

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

del colors[4]  # remove brown   ③

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

colors.remove('green')  ④

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

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

print("sum of lists:", sum_of_lists)

product = [True] * 5  ⑥

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

① Test for membership in list

② Concatenate iterable using ", " as delimiter

③ Permanently remove element with index 4

④ Remove element by value

⑤ Add 3 lists together; combines all elements

⑥ Multiply a list; replicates elements

***sequence_operators.py***

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

# Functions for all sequences

- Many builtin functions expect a sequence
- Syntax

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

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

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

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

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

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

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

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

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

## Example

### sequence_functions.py

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

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

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

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

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

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

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

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

① sorted() returns a sorted list

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

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

***sequence_functions.py***

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

sorted: black blue brown green red yellow
man bites dog

full_names: <zip object at 0x7fb2b8195370>

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

# Using enumerate()

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

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

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

for i,item in enumerate(s):
    print(i,item)

for i,item in enumerate(s,1)
    print(i,item)
```

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

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

you get this (virtual) list of tuples:

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

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

## Example

### enumerate.py

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

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

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

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

print()


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

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

② Second parameter to enumerate is added to index

### *enumerate.py*

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

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

# The range() function

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

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

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

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

## Example

### using_ranges.py

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

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

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

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

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

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

① Start=1, Stop=6 (1 through 5)

② Start=0, Stop=6 (0 through 5)

③ Start=3, Stop=12 (3 through 11)

④ Start=5, Stop=30, Step=5 (5 through 25 by 5)

⑤ Start=10, Stop=1, Step=-1 (10 through 1 by 1)

***using_ranges.py***

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

# List comprehensions

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

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

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

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

can be rewritten as

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

A conditional if may be added:

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

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

## Example

### list_comprehensions.py

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

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

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

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

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

doubles = [v * 2 for v in values]   ③

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

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

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

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

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

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

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

① Simple transformation of all elements

② Transformation of selected elements only

③ Any kind of data is OK

④ Select only integers from list

⑤ More than one **for** is OK

***list_comprehensions.py***

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

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

[2, 42, 18, 92]

>gronk< >pulaba< >floog<

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

*etc etc*

# Generator Expressions

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

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

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

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

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

## Example

### generator_expressions.py

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

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

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

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

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

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

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

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

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

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

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

***generator_expressions.py***

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

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

2 42 18 92

>gronk< >pulaba< >floog<

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

# Chapter 5 Exercises

## Exercise 5-1 (pow2.py)

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

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

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

## Exercise 5-2 (sequences.py)

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

## Exercise 5-3 (sequences.py)

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

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

## Exercise 5-4 (sieve.py)

*FOR ADVANCED STUDENTS*

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

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

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

Use two *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.

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

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

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

## Pseudocode for sieve.py

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

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

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

# Chapter 6: Working with Files

## Objectives

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

# Text file I/O

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

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

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

# Opening a text file

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

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

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

## Example

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

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

# The *with* block

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

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

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

# Reading a text file

- Iterate through file with for/in

```
    for line in file_in
```

- Use methods of the file object

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

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

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

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

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

## Example

### read_tyger.py

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

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

① **tyger_in** is return value of **open(...)**

② **tyger_in** is a *generator*, returning one line at a time

③ the line already has a newline, so **print()** does not need one

### *read_tyger.py*

```
           The Tyger

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

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

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

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

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

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

                    by William Blake
```

## Example

### reading_files.py

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

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

mary_in = open(FILE_NAME)  ①
# read file...
mary_in.close()  ②

with open(FILE_NAME) as mary_in:  ③
    for raw_line in mary_in:  ④
        line = raw_line.rstrip()  ⑤
        print(line)
print('-' * 60)

with open(FILE_NAME) as mary_in:
    contents = mary_in.read()  ⑥
    print("NORMAL:")
    print(contents)
    print("=" * 20)
    print("RAW:")
    print(repr(contents))  ⑦
print('-' * 60)

with open(FILE_NAME) as mary_in:
    lines_with_nl = mary_in.readlines()  ⑧
    print(lines_with_nl)
print('-' * 60)

with open(FILE_NAME) as mary_in:
    lines_without_nl = mary_in.read().splitlines()  ⑨
    print(lines_without_nl)
```

① open file for reading

② close file (easy to forget to do this!)

③ open file for reading

④ iterate over lines in file (line retains \n)

⑤ rstrip('\n\r') removes whitespace (including \r or \n) from end of string

⑥ read entire file into one string

⑦ print string in "raw" mode

⑧ readlines() reads all lines into an array

⑨ splitlines() splits string on '\n' into lines

***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']
```

# Writing to a text file

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

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

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

## Example

**write_file.py**

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

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

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

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

② write() does not add \n automatically

***write_file.py***

***cat states.txt (Windows: type states.txt)***

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

> "writelines" should have been called "writestrings"

*Table 11. File Methods*

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

# Chapter 6 Exercises

## Exercise 6-1 (line_no.py)

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

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

Test with the following commands:

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

Test with other files, as desired

## Exercise 6-2 (alt_lines.py)

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

## Exercise 6-3 (count_alice.py, count_words.py)

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

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

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

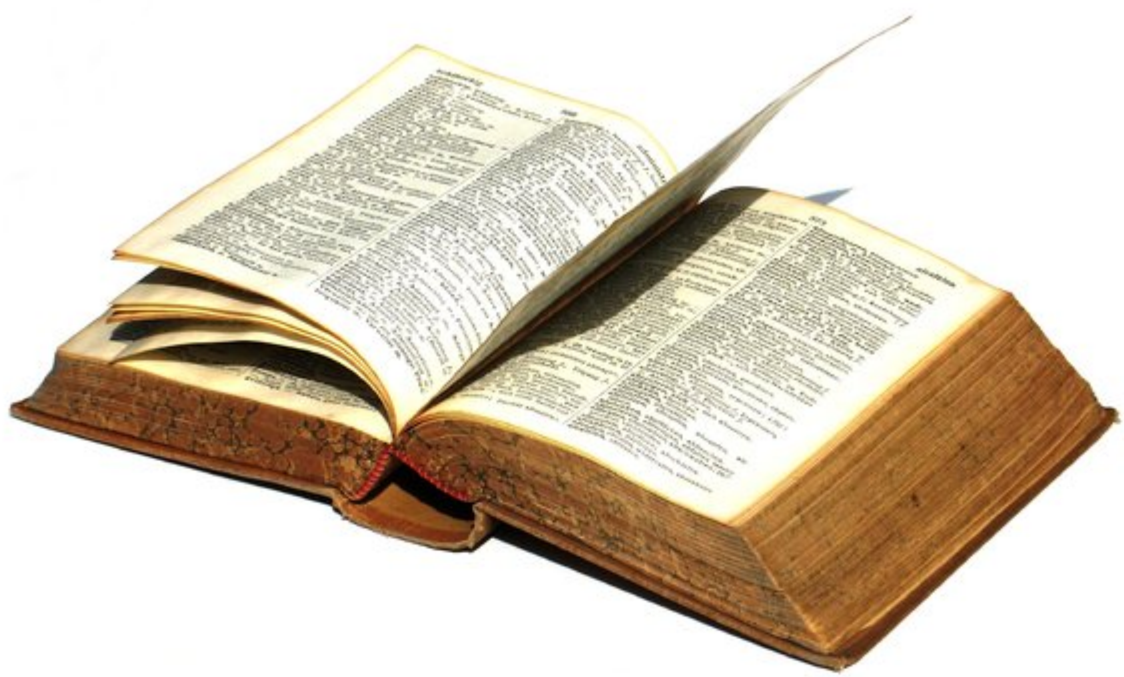
> FOR ADVANCED STUDENTS (icount_words.py) Modify count_words.py to make the search case-insensitive.

# Chapter 7: Dictionaries 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
- Associates keys with values
- called "hashes", "hash tables" or "associative arrays" in other languages
- Rich set of functions available

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

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

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

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

For instance, a dictionary might

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

# When to use dictionaries?

- Mapping
- Counting

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

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

# Creating dictionaries

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

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

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

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

## Example

### creating_dicts.py

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

d1 = dict()  ①

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

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

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

state_caps = dict(pairs)  ④

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

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

① create new empty dict

② initialize dict with literal key/value pairs (keys can be any string, number or tuple)

③ initialize dict with named parameters; keys must be valid identifier names

④ initialize dict with an iterable of pairs

⑤ print value for given key

⑥ assign to new key

⑦ overwrite existing key

***creating_dicts.py***

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

*Table 12. Frequently used dictionary functions and operators*

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

*Table 13. Less frequently used dictionary functions*

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

# Getting dictionary values

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

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

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

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

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

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

## Example

### getting_dict_values.py

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

d1 = dict()

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

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

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

state_caps = dict(pairs)

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

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

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

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

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

① print value where key is 'SLC'

② print key if key is in dictionary

③ get value if key in dict, otherwise get None

④ get value if key in dict, otherwise get 'NO SUCH AIRPORT'

⑤ get value if key in dict, otherwise get 'Palm Springs' AND set key

⑥ check for key in dict

***getting_dict_values.py***

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

# Iterating through a dictionary

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

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

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

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

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

## Example

**iterating_over_dicts.py**

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

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

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

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

***iterating_over_dicts.py***

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

# Reading file data into a dictionary

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

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

## Example

**read_into_dict_of_tuples.py**

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

from pprint import pprint

knight_info = {}  ①

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

pprint(knight_info)
print()

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

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

① create empty dict

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

***read_into_dict_of_tuples.py***

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

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

Not Sure
```

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

# Counting with dictionaries

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

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

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

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

## Example

**count_with_dict.py**

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

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

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

① create empty dict

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

***count_with_dict.py***

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

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

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

# About sets

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

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

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

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

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

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

# Creating Sets

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

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

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

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

# Working with sets

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

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

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

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

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

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

## Example

### set_examples.py

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

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

set1.add('taupe')  ②

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

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

① create literal set

② add element to set (ignored if already in set)

③ intersection of two sets

④ union of two sets

⑤ XOR (symmetric difference); items in one set but not both

⑥ Remove items in right set from left set

⑦ Create set from iterable (e.g., list)

### *set_examples.py*

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

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

*Table 14. Set functions and methods*

| Function | Description |
|---|---|
| m in S | True if s contains member m |
| m not in S | True if S does not contain member m |
| len(s) | the number of items in S |
| S.add(m) | Add member m to S (if S already contains m do nothing) |
| S.clear() | remove all members from S |
| S.copy() | a (shallow) copy of S |
| S - S2<br>S.difference(S2) | Return the set of all elements in S that are not in S2 |
| S.difference_update(S2) | Remove all members of S2 from S |
| S.discard(m) | Remove member m from S if it is a member. If m is not a member, do nothing. |
| S & S2<br>S.intersection(S2) | Return new set with all unique members of S and S2 |
| S.isdisjoint(S2) | Return True if S and S2 have no members in common |
| S.issubset(S2) | Return True is S is a subset of S2 |
| S.issuperset(S2) | Return True is S2 is a subset of S |
| S.pop() | Remove and return an arbitrary set element. Raises KeyError if the set is empty. |
| S.remove(m) | Remove member m from a set; it must be a member. |
| S ^ S2<br>S.symmetric_difference(S2) | Return all members in S or S2 but not both. |
| S.symmetric_difference_update(S2) | Update a set with the symmetric difference of itself and another. |
| S | S2<br>S.union(S2) | Return all members that are in S or S2 |
| S.update(S2) | Update a set with the union of itself and S2 |

# Chapter 7 Exercises

## Exercise 7-1 (scores.py)

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

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

## Exercise 7-2 (shell_users.py)

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

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

## Exercise 7-3 (common_fruit.py)

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

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

## Exercise 7-4 (set_sieve.py)

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

# Chapter 8: Functions

## Objectives

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

# Defining a function

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

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

Functions can take fixed or variable parameters and return single or multiple values.

Functions must be defined before they can be called.

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

## Example

### function_basics.py

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


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


say_hello()  ③


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


h = get_hello()  ⑤
print(h)
print()


def sqrt(num):  ⑥
    return num ** .5


m = sqrt(1234)  ⑦
n = sqrt(2)

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

① Function takes no parameters

② If no **return** statement, return None

③ Call function (arguments, if any, in () )

④ Function returns value

⑤ Store return value in h

⑥ Function takes exactly one argument

⑦ Call function with one argument

***function_basics.py***

```
Hello, world

Hello, world

m is 35.128 n is 1.414
```

# Returning values

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

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

**return** without a value returns None.

## Example

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

① return None

② return integer 5

③ return object x

④ return tuple of values

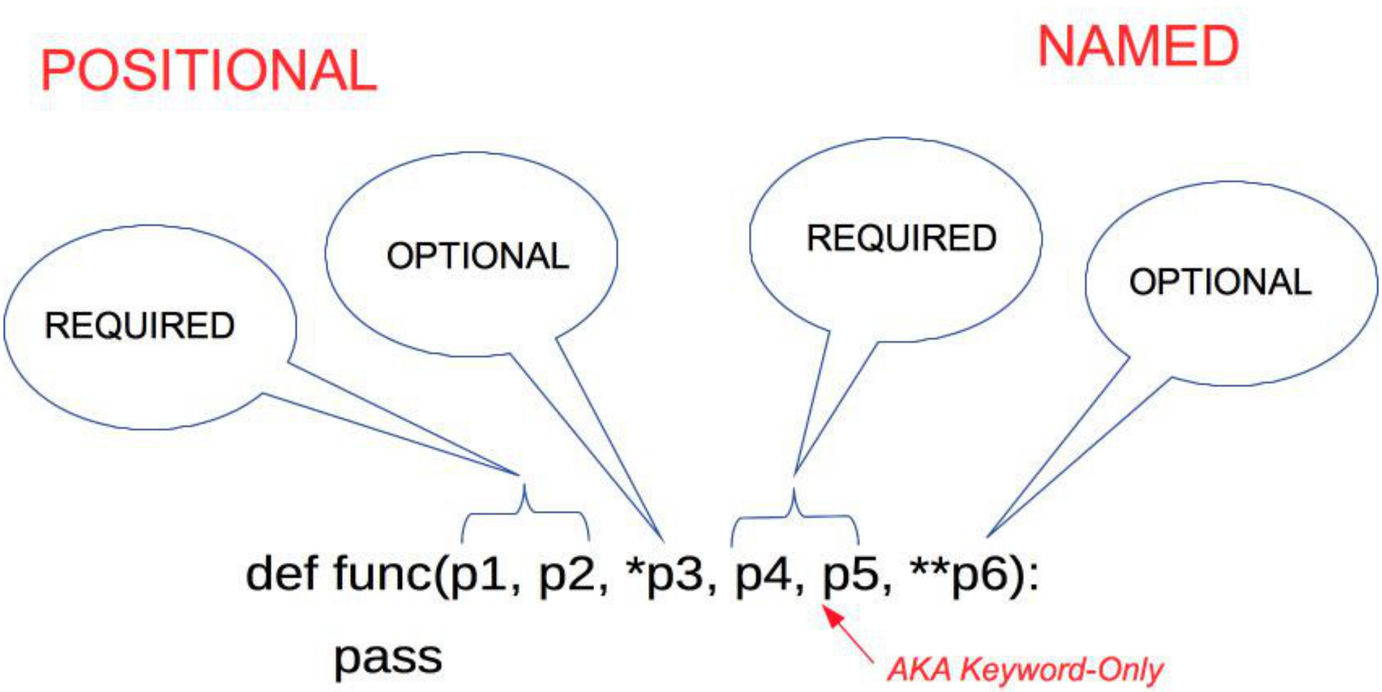
**TIP** Remember that **return** is a statement, not a function.

# Function parameters

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

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

Required parameters may have default values.



## Positional parameters

### Required positional parameters

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

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

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

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

### Optional positional parameters

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

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

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

## Named parameters

### Keyword-only parameters (required named parameters)

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

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

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

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

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

### Keyword parameters (optional named parameters)

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

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

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

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

## Example

### function_parameters.py

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

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


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


def fun_two(n):  ②
    return n ** 2


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


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


fun_three()
fun_three(10)
print()


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


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


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

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


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


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


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

① no parameters

② one required parameter

③ one required parameter with default value

④ one fixed, plus optional parameters

⑤ keyword-only parameters

⑥ keyword (named) parameters

***function_parameters.py***

```
fun_one(): Hello, world

fun_two(5) is 25

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

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

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

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

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

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

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

# Variable scope

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

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

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

## Example

### variable_scope.py

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

x = 5


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


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


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

① Local variable; does not modify global x

② Local variable

③ Uses global x since there is no local x

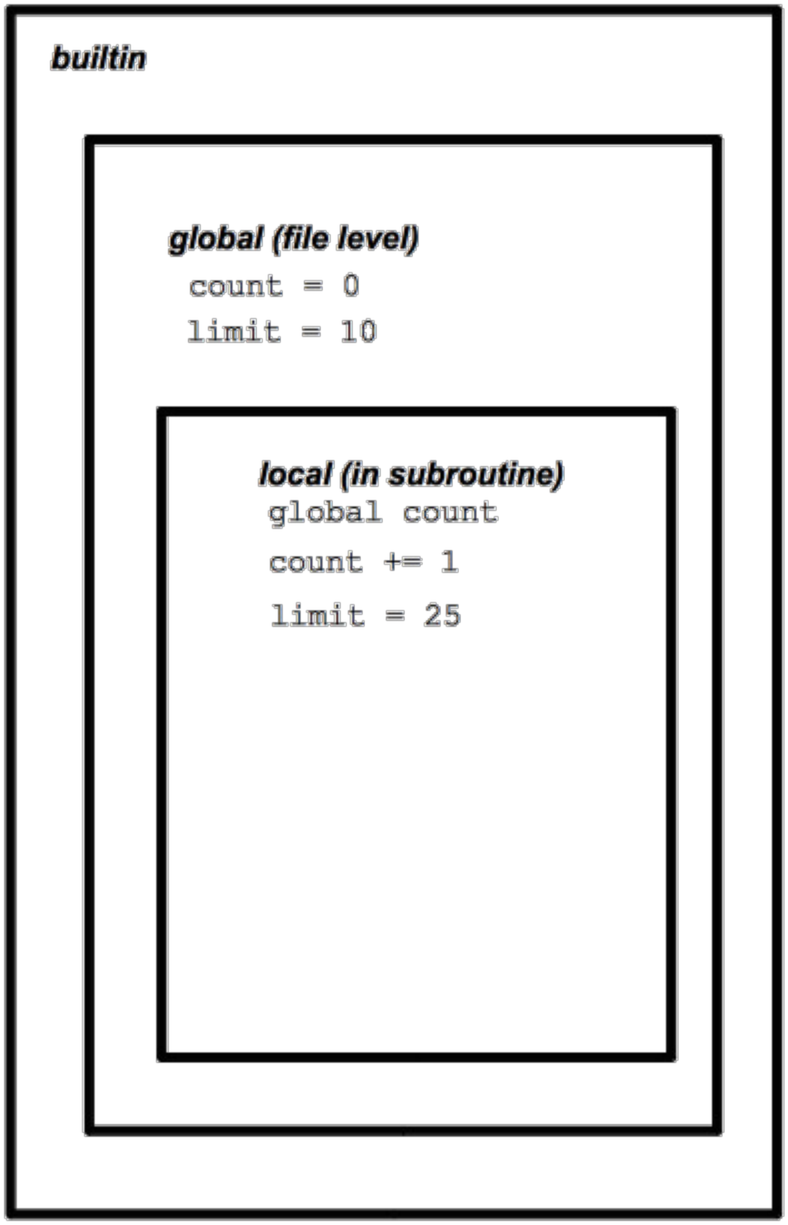
***variable_scope.py***

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

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

main: x is  5
```


builtin
global (file level)
count = 0
limit = 10
local (in subroutine)
global count
count += 1
limit = 25

## The global statement

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

What happens when you want to change a global variable? The **global** statement allows you to declare a global variable within a function. That is, when you assign to the variable, you are assigning to the global variable, instead of a local variable.

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

## Example

### global_statement.py

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

x = 5


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


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

① Mark x as global, not local

② Modify global variable x

***global_statement.py***

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

# Chapter 8 Exercises

## Exercise 8-1 (dirty_strings.py)

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

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

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

## Exercise 8-2 (c2f_func.py)

Define a function named **c2f** that takes one number as a parameter, and then returns the value converted from Celsius to Fahrenheit. Test your function by calling it with the values 100, 0, 37, and -40 (one at a time, not all at once).

**Example**

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

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

## Exercise 8-3 (calc.py)

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

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

**NOTE**

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

*FOR ADVANCED STUDENTS*

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

# Chapter 9: Sorting

## Objectives

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

# Sorting Overview

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

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

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

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

What can you sort?

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

# The sorted() function

- Returns a sorted copy of any collection
- Customize with named keyword parameters

```
key=
reverse=
```

The sorted() builtin function returns a sorted copy of its argument, which can be any iterable.

You can customize sorted with the **key** parameter.

## Example

**basic_sorting.py**

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

"""Basic sorting example"""

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

sorted_fruit = sorted(fruits)  ①

print(sorted_fruit)
```

① sorted() returns a list

***basic_sorting.py***

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

# Custom sort keys

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

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

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

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

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

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

## Example

### custom_sort_keys.py

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

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

def ignore_case(item):  ①
    return item.lower()  ②

fs1 = sorted(fruit, key=ignore_case)  ③
print("Ignoring case:")
print(" ".join(fs1), end="\n\n")

def by_length_then_name(item):
    return (len(item), item.lower())  ④

fs2 = sorted(fruit, key=by_length_then_name)
print("By length, then name:")
print(" ".join(fs2))
print()

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

n1 = sorted(nums)  ⑤
print("Numbers sorted numerically:")
for n in n1:
    print(n, end=' ')
print("\n")

n2 = sorted(nums, key=str)  ⑥
print("Numbers sorted as strings:")
for n in n2:
    print(n, end=' ')
print()
```

① Parameter is *one* element of iterable to be sorted

② Return value to sort on

③ Specify function with named parameter **key**

④ Key functions can return tuple of values to compare, in order

⑤ Numbers sort numerically by default

⑥ Sort numbers as strings

***custom_sort_keys.py***

```
Ignoring case:
Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon lime
lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon

By length, then name:
FIG date Kiwi lime pear Apple grape guava lemon peach banana cherry lychee ORANGE papaya
apricot Tamarind BLUEberry persimmon elderberry Watermelon pomegranate

Numbers sorted numerically:
5 32 80 255 400 800 1000 5000

Numbers sorted as strings:
1000 255 32 400 5 5000 80 800
```

## Example

### sort_holmes.py

```
#!/usr/bin/env python
"""Sort titles, ignoring leading articles"""
books = [
    "A Study in Scarlet",
    "The Sign of the Four",
    "The Hound of the Baskervilles",
    "The Valley of Fear",
    "The Adventures of Sherlock Holmes",
    "The Memoirs of Sherlock Holmes",
    "The Return of Sherlock Holmes",
    "His Last Bow",
    "The Case-Book of Sherlock Holmes",
]


def strip_articles(title):  ①
    title = title.lower()
    for article in 'a ', 'an ', 'the ':
        if title.startswith(article):
            title = title[len(article):]  ②
            break
    return title


for book in sorted(books, key=strip_articles):  ③
    print(book)
```

① create function which takes element to compare and returns comparison key

② remove article by using a slice that starts after article + space`

③ sort using custom function

***sort_holmes.py***

```
The Adventures of Sherlock Holmes
The Case-Book of Sherlock Holmes
His Last Bow
The Hound of the Baskervilles
The Memoirs of Sherlock Holmes
The Return of Sherlock Holmes
The Sign of the Four
A Study in Scarlet
The Valley of Fear
```

# Lambda functions

- Shortcut for function definition
- Create function on-the-fly
- Body must be an expression
- May take any number of parameters
- For sorting, takes one parameter

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

```
lambda parameters: expression
```

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

When using a lambda function with the **key** parameter of **sorted()**, it expects a single parameter, which is one element of the list being sorted. Lambda functions are particularly useful for sorting nested collections, such as lists of tuples.

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

Thus, the following can be used as a template:

```
lambda e: expression
```

## Example

### lambda_sort.py

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

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

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

fs1 = sorted(fruit, key=lambda e: e.lower())  ①
print("Ignoring case:")
print(' '.join(fs1))
print()

fs2 = sorted(fruit, key=lambda e: (len(e), e.lower()))  ②
print("By length, then name:")
print(' '.join(fs2))
print()

fs3 = sorted(nums)
print("Numbers sorted numerically:")
for n in fs3:
    print(n, end=' ')
print()
print()
```

① lambda returns key function that converts each element to lower case

② lambda returns tuple

***lambda_sort.py***

```
Ignoring case:
Apple apricot banana BLUEberry cherry date elderberry FIG grape guava Kiwi lemon lime
lychee ORANGE papaya peach pear persimmon pomegranate Tamarind Watermelon

By length, then name:
FIG date Kiwi lime pear Apple grape guava lemon peach banana cherry lychee ORANGE papaya
apricot Tamarind BLUEberry persimmon elderberry Watermelon pomegranate

Numbers sorted numerically:
5 32 80 255 400 800 1000 5000
```

# Sorting nested data

- Collections sorted item-by-item
- Only same kind of items can be compared

You can sort a collection of collections, for instance a list of tuples. For each tuple, sorted() will compare the first element of the tuple, then the second, and so forth.

All of the items in the collection must be the same — they all must be tuples, or lists, or dicts, or strings, or anything else.

Use a lambda function, and index each element as necessary. To sort a list of tuples by the third element of each tuple, use

```
list2 = sorted(list1,key=lambda e: e[2])
```

## Example

### nested_sort.py

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

computer_people = [
    ('Melinda', 'Gates', 'Gates Foundation', '1964-08-15'),
    ('Steve', 'Jobs', 'Apple', '1955-02-24'),
    ('Larry', 'Wall', 'Perl', '1954-09-27'),
    ('Paul', 'Allen', 'Microsoft', '1953-01-21'),
    ('Larry', 'Ellison', 'Oracle', '1944-08-17'),
    ('Bill', 'Gates', 'Microsoft', '1955-10-28'),
    ('Mark', 'Zuckerberg', 'Facebook', '1984-05-14'),
    ('Sergey','Brin', 'Google', '1973-08-21'),
    ('Larry', 'Page', 'Google', '1973-03-26'),
    ('Linus', 'Torvalds', 'Linux', '1969-12-28'),
]



# sort by first name (default)
for first_name, last_name, organization, dob in sorted(computer_people):
    print(first_name, last_name, organization, dob  )
print('-' * 60)

# sort by last name
for first_name, last_name, organization, dob in sorted(computer_people, key=lambda e: e[
1]):  ①
    print(first_name, last_name, organization, dob)
print('-' * 60)

# sort by company
for first_name, last_name, organization, dob in sorted(computer_people, key=lambda e: e[
2]): ②
    print(first_name, last_name, organization, dob)
```

① Select element of nested tuple for sorting

② Select different element of nested tuple for sorting

***nested_sort.py***

```
Bill Gates Microsoft 1955-10-28
Larry Ellison Oracle 1944-08-17
Larry Page Google 1973-03-26
Larry Wall Perl 1954-09-27
Linus Torvalds Linux 1969-12-28
Mark Zuckerberg Facebook 1984-05-14
Melinda Gates Gates Foundation 1964-08-15
Paul Allen Microsoft 1953-01-21
Sergey Brin Google 1973-08-21
Steve Jobs Apple 1955-02-24
------------------------------------------------------------
Paul Allen Microsoft 1953-01-21
Sergey Brin Google 1973-08-21
Larry Ellison Oracle 1944-08-17
Melinda Gates Gates Foundation 1964-08-15
Bill Gates Microsoft 1955-10-28
Steve Jobs Apple 1955-02-24
Larry Page Google 1973-03-26
Linus Torvalds Linux 1969-12-28
Larry Wall Perl 1954-09-27
Mark Zuckerberg Facebook 1984-05-14
------------------------------------------------------------
Steve Jobs Apple 1955-02-24
Mark Zuckerberg Facebook 1984-05-14
Melinda Gates Gates Foundation 1964-08-15
Sergey Brin Google 1973-08-21
Larry Page Google 1973-03-26
Linus Torvalds Linux 1969-12-28
Paul Allen Microsoft 1953-01-21
Bill Gates Microsoft 1955-10-28
Larry Ellison Oracle 1944-08-17
Larry Wall Perl 1954-09-27
```

# Sorting dictionaries

- Use dict.items()
- By default, sorts by key
- Use a lambda function or itemgetter() to sort by value

While a dictionary can't be sorted, the keys to a dictionary can. Better yet, the list of tuples returned by DICT.items() can be sorted. This list will be sorted by keys, unless you specify a key function.

Use a lambda function or operator.itemgetter() to specify the 2nd element of the key,value tuple to sort by values.

Sorting dictionary.items() is really just sorting a list of tuples.

## Example

**sorting_dicts.py**

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

count_of = dict(red=5, green=18, blue=1, pink=0, grey=27, yellow=5)

# sort by key
for color, num in sorted(count_of.items()):  ①
    print(color, num)

print()

# sort by value
for color, num in sorted(count_of.items(), key=lambda e: e[1]): ②
    print(color, num)
```

① No special sort needed to sort by key

② Sorting by value uses second element of nested (key, value) pairs returned by items()

***sorting_dicts.py***

```
blue 1
green 18
grey 27
pink 0
red 5
yellow 5

pink 0
blue 1
red 5
yellow 5
green 18
grey 27
```

# Sorting in reverse

- Use reverse=True

To sort in reverse, add the **reverse** parameter to sorted() or list.sort() with a true value (e.g. True).

## Example

**reverse_sort.py**

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

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

print("reverse, case-sensitive:")
sorted_fruits = sorted(fruits, reverse=True)  ①
print(" ".join(sorted_fruits))
print()

print("reverse, case-insensitive:")
sorted_fruits = sorted(fruits, reverse=True, key=lambda e: e.lower())  ②
print(" ".join(sorted_fruits))
print()
```

① Set **reverse** to True to reverse sort

② **reverse** can be combined with key functions

### *reverse_sort.py*

```
reverse, case-sensitive:
pomegranate persimmon pear peach papaya lychee lime lemon guava grape elderberry date
cherry banana apricot Watermelon Tamarind ORANGE Kiwi FIG BLUEberry Apple

reverse, case-insensitive:
Watermelon Tamarind pomegranate persimmon pear peach papaya ORANGE lychee lime lemon Kiwi
guava grape FIG elderberry date cherry BLUEberry banana apricot Apple
```

# Sorting lists in place

- Use list.sort()
- Only for lists (not strings or tuples

To sort a list in place, use the list's sort() method. It works exactly like sorted(), except that the sort changes the order of the items in the list, and does not make a copy.

## Example

**sort_in_place.py**

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

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

fruit.sort(key=str.lower)  ①

print(" ".join(fruit))
```

① List is sorted in place; cannot be undone

***sort_in_place.py***

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

# Chapter 9 Exercises

## Exercise 9-1 (scores_by_score.py)

Redo **scores.py**, printing out the students in descending order by score.

**TIP** You will not need to change anything in **scores.py** other than the loop that prints out the names and scores.

## Exercise 9-2 (alt_sorted.py)

Read in the file alt.txt. Put all the words that start with 'a' in to a file named a_sorted.txt, in sorted order. Put all the words that start with 'b' in b_sorted.txt, in reverse sorted order.

**TIP** Read through the file once, putting lines into two lists.

## Exercise 9-3 (sort_fruit.py)

Using the file fruit.txt, print it out:

- sorted by name case-sensitively (the default)
- sorted by name case-insensitively (ignoring case)
- sorted by length of name, then by name
- sorted by the 2nd letter of the name, then the first letter

## Exercise 9-4 (sort_presidents.py)

Using the file presidents.txt, print out the presidents' first name, last name, and state of birth, sorted by last name, then first name.

**TIP** Use the split() method on each line to get the individual fields.

# Chapter 10: Errors and Exception Handling

## Objectives

- Understanding syntax errors
- Handling exceptions with try-except-else-finally
- Learning the standard exception objects

# Syntax errors

- Generated by the parser
- Cannot be trapped

Syntax errors are generated by the Python parser, and cause execution to stop (your script exits). They display the file name and line number where the error occurred, as well as an indication of where in the line the error occurred.

Because they are generated as soon as they are encountered, syntax errors may not be handled.

## Example

```
  File "<stdin>", line 1
    for x in bargle
                  ^
SyntaxError: invalid syntax
```

**TIP** When running in interactive mode, the filename is <stdin>.

# Exceptions

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

Even if code is syntactically correct, errors can occur. A common run-time error is to attempt to open a non-existent file. Such errors are called exceptions, and cause the interpreter to stop with an error message.

Python has a hierarchy of builtin exceptions; handling an exception higher in the tree will handle any children of that exception.

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

## Example

**exception_unhandled.py**

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

x = 5
y = "cheese"

z = x + y  ①
```

① Adding a string to an int raises **TypeError**

***exception_unhandled.py***

```
Traceback (most recent call last):
  File "exception_unhandled.py", line 6, in <module>
    z = x + y  ①
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Handling exceptions with try

- Use try/except clauses
- Specify expected exception

To handle an exception, put the code which might generate an exception in a try block. After the try block, you must specify a except block with the expected exception. If an exception is raised in the try block, execution stops and the interpreter looks for the exception in the except block. If found, it executes the except block and execution continues; otherwise, the exception is treated as fatal and the interpreter exits.

## Example

**exception_simple.py**

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

try:  ①
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")

except TypeError as err:    ②
    print("Naughty programmer! ", err)

print("After try-except")  ③
```

① Execute code that might have a problem

② Catch the expected error; assign error object to **err**

③ Get here whether or not exception occurred

***exception_simple.py***

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

# Handling multiple exceptions

- Use a tuple of exception names, but with single argument

If your try clause might generate more than one kind of exception, you can specify a tuple of exception types, then the variable which will hold the exception object.

## Example

**exception_multiple.py**

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except (IOError, TypeError) as err:  ①
    print("Naughty programmer! ", err)
```

① Use a tuple of 2 or more exception types

***exception_multiple.py***

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

# Handling generic exceptions

- Use **Exception**
- Specify except with no exception list
- Clean up any uncaught exceptions

As a shortcut, you can specify **Exception** or an empty exception list. This will handle any exception that occurs in the try block.

## Example

**exception_generic.py**

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except Exception as err: ①
    print("Naughty programmer! ", err)
```

① Will catch *any* exception

***exception_generic.py***

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

# Ignoring exceptions

- Use the **pass** statement

Use the **pass** statement to do nothing when an exception occurs

Because the except clause must contain some code, the pass statement fulfills the syntax without doing anything.

## Example

**exception_ignore.py**

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

try:
    x = 5
    y = "cheese"
    z = x + y
    f = open("sesame.txt")
    print("Bottom of try")

except(TypeError, IOError): ①
    pass
```

① Catch exceptions, and do nothing

***exception_ignore.py***

```
_no output_
```

This is probably a bad idea...

# Using else

- executed if no exceptions were raised
- not required
- can make code easier to read

The last except block can be followed by an else block. The code in the else block is executed only if there were no exceptions raised in the try block. Exceptions in the else block are not handled by the preceding except blocks.

The else lets you make sure that some code related to the try clause (and before the finally clause) is only run if there's no exception, without trapping the exception specified in the except clause.

```
try:
    something_that_can_throw_ioerror()
except IOError as e:
    handle_the_IO_exception()
else:
# we don't want to catch this IOError if it's raised
    something_else_that_throws_ioerror()
finally:
    something_we_always_need_to_do()
```

## Example

### exception_else.py

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

numpairs = [(5, 1), (1, 5), (5, 0), (0, 5)]

total = 0

for x, y in numpairs:
    try:
        quotient = x / y
    except Exception as err:
        print("uh-oh, when y = {}, {}".format(y, err))
    else:
        total += quotient  ①
print(total)
```

① Only if no exceptions were raised

***exception_else.py***

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

# Cleaning up with finally

- Executed whether or not exception occurs
- Code executed whether or not exception raised
- Code runs even if **exit()** called
- For cleanup

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**

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

try:
    x = 5
    y = 37
    z = x + y
    print("z is", z)
except TypeError as err:    ①
    print("Caught exception:", err)
finally:
    print("Don't care whether we had an exception")  ②

print()

try:
    x = 5
    y = "cheese"
    z = x + y
    print("Bottom of try")
except TypeError as err:
    print("Caught exception:", err)
finally:
    print("Still don't care whether we had an exception")
```

① Catch **TypeError**

② Print whether **TypeError** is caught or not

***exception_finally.py***

```
z is 42
Don't care whether we had an exception

Caught exception: unsupported operand type(s) for +: 'int' and 'str'
Still don't care whether we had an exception
```

## The Standard Exception Hierarchy (Python 3.7)

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
```

```
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
```

# Chapter 10 Exercises

## Exercise 10-1 (c2f_loop_safe.py)

Rewrite c2f_loop.py to handle the error that occurs if the user enters non-numeric data. The script should print a message and keep going if an error occurs.

## Exercise 10-2 (c2f_batch_safe.py)

Rewrite c2f_batch.py to handle the ValueError that occurs if sys.argv[1] is not a valid number.

# Chapter 11: Creating and using modules

## Objectives

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

# What is a module?

- Sharable library of python code
- Can have initialization code

A module is just a Python script that is imported in some other Python script. In some languages it would be called a library. Any time a function, class, or variable might be useful in more than one place, it should be put in a module, in order to avoid cut-and-paste disease.

If the function or class need to change, you only need to change it in one place.

Any code in a module that is not inside a function definition will be executed the first time a module is loaded, and thus can provide initialization code.

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

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

# Creating Modules

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

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

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

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

## Example

### breakfast.py

```
#!/usr/bin/env python
'''
Spam -- provides a tasty breakfast
'''

# separator for toast toppings
TOPSEP = " and "


def eggs(how):
    '''cook some eggs'''
    print("Cooking up some lovely {} eggs".format(how))


def toast(*toppings):
    '''cook some toast'''
    print("Toasting up some toast with ", TOPSEP.join(toppings))
```

### eat.py

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

from breakfast import eggs, toast

eggs("fried")
toast("butter", "strawberry jam")

print()
help(eggs)
print()
help(toast)
```

### *eat.py*

```
Cooking up some lovely fried eggs
Toasting up some toast with  butter and strawberry jam

Help on function eggs in module breakfast:

eggs(how)
    cook some eggs


Help on function toast in module breakfast:

toast(*toppings)
    cook some toast
```

# The import statement

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

The import statement is used to load modules. It can be used in several ways:

```
import breakfast
```

Load module breakfast.py. To call function **eggs** which is defined in breakfast.py, use breakfast.eggs() breakfast.eggs("scrambled")

```
from breakfast import eggs [, toast, coffee, ...]
```

Load module breakfast.py and add eggs to the current namespace, so it can be called without the module name:

```
    eggs("fried")
```

To import multiple objects, separate them with commas.

```
from breakfast import *
```

Load module breakfast.py and add all objects (except those than begin with "_") to the current namespace. eggs("poached") toast("butter", "jam")

This is generally not considered a **Good Thing** unless you know what you're doing, or the module documentation suggests it.

# Where did __pycache__ come from?

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

After running a script which uses module cheese.py in the current directory, you will notice that a folder named __pycache__ appeared. The Python interpreter saves a compiled version of each imported module into the __pycache__ folder. For portability, each compiled version embeds a version string (such as "cpython-64") into the name, so that the same __pycache__ folder can contain compiled modules for different versions of Python.

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

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

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

# Module search path

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

When you specify a module to load with the import statement, it first looks in the current directory, and then searches the directories listed in sys.path.

```
>>> import sys
>>> sys.path
['', '/usr/lib/python24.zip', '/usr/lib/python2.4', '/usr/lib/python2.4/plat-linux2',
'/usr/lib/python2.4/lib-tk', '/usr/lib/python2.4/lib-dynload',
'/usr/local/lib/python2.4/site-packages', '/usr/lib/python2.4/site-packages',
'/usr/lib/python2.4/site-packages/...]
```

To add locations, put one or more directories to search in the PYTHONPATH environment variable. Separate multiple paths by semicolons for Windows, or colons for Unix/Linux. This will add them to sys.path.

**Windows**

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

**non-Windows**

export PYTHONPATH="/home/bob/python"

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

# Packages

- Collection of modules
- Corresponds to a directory
- Can have subpackages

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

Load modules in a package with **import package**.

Access subpackages via **package.sub**.

Access functions via **package.sub.function()**.

# Example

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

```
media
media/{dunder}init{dunder}.py
media/dvd.py
media/videotape.py
media/cd.py
```

To use the function Search that is defined in module dvd, import media.dvd:

```
import media.dvd
media.dvd.Search("Uma Thurman")
```

Other variations are:

```
from media import dvd
dvd.Search("Uma Thurman")
_or_
from media.dvd import Search
Search("Uma Thurman")
```

If a module named __init__.py is present in the package, it is executed when the package or any of its modules are imported.

# Module Aliases

- Provide alternate name for module
- Save typing
- Make code more readable

To load a module with a different name, use the syntax

```
import module as alias
```

This is useful to save typing, especially when using packages. It can also make code more readable.

## Example

```
import media.dvd as dvd
found = dvd.search('beatles')

import xml.etree.ElementTree as ET
import Tkinter as tk
```

# When the batteries aren't included

- Python Package Index (PyPI) has over 99,000 packages
- Install with the **pip** utility

Although the Python distribution claims to be "batteries included", functionality beyond the standard library is provided by so-called third party modules. There are about 99,000 such packages in the Python Package Index (http://pypi.python.org/pypi).

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

# Chapter 11 Exercises

## Exercise 11-1 (temp_conv.py)

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

Conversion formulas

```
F = ((9 * C) / 5.0 ) + 32
C = (F - 32) * (5.0/9)
```

**TIP** write some simple code in the `if __name__ == "__main__"` code to do a simple test of the functions.

## Exercise 11-2 (c2f_mod.py)

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

## Exercise 11-3 (f2c.py)

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

## Exercise 11-4 (c2f_loop_mod.py)

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

# Chapter 12: 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.

Objects may inherit attributes (data and methods) from other objects.

Methods are not polymorphic; i.e., you can't define multiple versions of a method, with different signatures, and have the corresponding method selected at runtime. However, because Python has dynamic typing, this is seldom needed.

# Defining classes

- Use the **class** statement
- **Syntax**

```
class ClassName(baseclass):
    pass
```

To create a class, declare the class with the **class** statement. Any base classes may be specified in parentheses, but are not required.

Classes are conventionally named with Camel Case (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, most Python programmers expect it.

The basic layout is this:

```
class ClassName(baseclass):
    classvar = value

    def __init__:(self,...):
        self._attrib = instancevalue;
        ClassName.attrib = classvalue;

    def method1:(self,...):
        self._attrib = instancevalue

    def method2:(self,...):
        x = self.method1()
```

## Example

### simple_class.py

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

class Simple():  ①
    def __init__(self, message_text):  ②
        self._message_text = message_text  ③

    def text(self):  ④
        return self._message_text


if __name__ == "__main__":
    msg1 = Simple('hello')  ⑤
    print(msg1.text())  ⑥

    msg2 = Simple('hi there')  ⑦
    print(msg2.text())
```

① default base class is **object**

② constructor

③ message text stored in instance object

④ instance method

⑤ instantiate an instance of Simple

⑥ call instance method

⑦ create 2nd instance of Simple

***simple_class.py***

```
hello
hi there
```

# Constructors

- Constructor is named **__init__**
- AKA initializer
- Passed *self* plus any parameters

A class's constructor (also known as the initializer) is named __init__. It receives the object being created, and any parameters passed into the initializer in the code as part of instantiation.

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

It is also normal to name data elements (variables) of a class with a leading underscore to indicate (in a non-mandatory way) that the variable is *private*. Access to private variables should be provided via public access methods (AKA getters) or properties.

# Instance methods

- Expect the object as first parameter
- Object conventionally named *self*
- Otherwise like normal Python functions
- Use *self* to access instance attributes or methods
- Use class name to access class data

Instance methods are defined like normal functions, but like constructors, the object that the method is called from is passed in as the first parameter. As with the constructor, the parameter should be named *self*.

## Example

### animal.py

```
class Animal():
    count = 0  ①

    def __init__(self, species, name, sound):
        self._species = species
        self._name = name
        self._sound = sound
        Animal.count += 1

    @property
    def species(self):
        return self._species

    @classmethod
    def kill(cls):
        cls.count -= 1

    @property
    def name(self):
        return self._name

    def make_sound(self):
        print(self._sound)

    @classmethod
    def remove(cls):
        cls.count -= 1  ②

    @classmethod
    def zoo_size(cls):  ③
        return cls.count


if __name__ == "__main__":
    leo = Animal("African lion", "Leo", "Roarrrrrrr")
    garfield = Animal("cat", "Garfield", "Meowwwww")
    felix = Animal("cat", "Felix", "Meowwwww")

    for animal in leo, garfield, felix:
        print(animal.name, "is a", animal.species, "--", end=' ')
        animal.make_sound()
```

① class data

② update class data from instance

③ zoo_size gets class object when called from instance or class

***animal.py***

```
Leo is a African lion -- Roarrrrrrr
Garfield is a cat -- Meowwwww
Felix is a cat -- Meowwwww
```

# 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 properties? Consider that you had a

## Example

### properties.py

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

class Person():

    def __init__(self, firstname=None, lastname=None):
        self._first_name = None
        self._last_name = None
        self.first_name = firstname  ①
        self.last_name = lastname

    @property
    def first_name(self):  ②
        return self._first_name

    @first_name.setter  ③
    def first_name(self, value):  ④
        if value is None or value.isalpha():
            self._first_name = value
        else:
            raise ValueError("First name may only contain letters")

    @property
    def last_name(self):
        return self._last_name

    @last_name.setter
    def last_name(self, value):
        if value is None or value.isalpha():
            self._last_name = value
        else:
            raise ValueError("Last name may only contain letters")


if __name__ == '__main__':
    person1 = Person('Ferneater', 'Eulalia')

    person2 = Person()
    person2.last_name = 'Pepperpot'  ⑤
    person2.first_name = 'Hortense'

    print("{} {}".format(person1.first_name, person1.last_name))
    print("{} {}".format(person2.first_name, person2.last_name))
```

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

① calls property

② getter property

③ decorator comes from getter property

④ setter property

⑤ access property

***properties.py***

```
Ferneater Eulalia
Hortense Pepperpot
First name may only contain letters
```

# Class methods and data

- Defined in the class, but outside of methods
- Defined as attribute of class name (similar to self)
- Define class methods with @classmethod
- Class methods get the class object as 1st parameter

Most classes need to store some data that is common to all objects created in the class. This is generally called class data.

Class attributes can be created by using the class name directly, or via class methods.

A class method is created by using the @classmethod decorator. Class methods are implicitly passed the class object.

Class methods can be called from the class object or from an instance of the class; in either case the method is passed the class object.

## Example

**class_methods_and_data.py**

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

class Rabbit:
    LOCATION = "the Cave of Caerbannog"  ①

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

    def display(self):
        print("This rabbit guarding {} uses {} as a weapon".
              format(self.LOCATION, self.weapon))  ②

    @classmethod  ③
    def get_location(cls):  ④
        return cls.LOCATION  ⑤


r = Rabbit("a nice cup of tea")
print(Rabbit.get_location())  ⑥
print(r.get_location())  ⑦
```

① class data (not duplicated in instances)

② instance method

③ the **@classmethod** decorator makes a function receive the class object, not the instance object

④ *get_location() is a *class* method

⑤ class methods can access class data via **cls**

⑥ call class method from class

⑦ call class method from instance

***class_methods_and_data.py***

```
the Cave of Caerbannog
the Cave of Caerbannog
```

# Static Methods

- Define with @staticmethod

A static method is a utility method that is included in the API of a class, but does not require either an instance or a class object. Static methods are not passed any implicit parameters.

Many classes do not need any static methods.

Define static methods with the @staticmethod decorator.

## Example

```
class Spam():

    @staticmethod
    def format_as_title(s):   # no implicit parameters
        return s.strip().title()
```

# Private methods

- Called by other methods in the class
- Not visible to users of the class
- Conventionally named with leading underscore

Private methods are those that are called only within the class. They are not part of the API – they are not visible to users of the class. Private methods may be instance, class, or static methods.

Name private methods with a leading underscore. This does not protect it from use, but gives programmers a hint that it's for internal use only.

# Inheritance

- Specify base classes after class name
- Multiple inheritance OK
- Depth-first, left-to-right search for methods not in derived class

Classes may inherit methods and data. 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 sought in the first base class in the list. If not found, it is sought in the base class of that class, if any, and so on. This is usually called a depth-first search.

The derived class inherits all attributes of the base class. If the base class initializer takes the same arguments as the derived class, then no extra coding is needed. Otherwise, to explicitly call the initializer in the base class, use super().__init__(args).

The simplest derived class would be:

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

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

## Example

### mammal.py

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

from animal import Animal


class Mammal(Animal):  ①
    def __init__(self, species, name, sound, gestation):
        super(Mammal, self).__init__(species, name, sound)
        self._gestation = gestation

    @property
    def gestation(self):  ②
        """Length of gestation period in days"""
        return self._gestation


if __name__ == "__main__":
    mammal1 = Mammal("African lion", "Bob", "Roarrrr", 120)
    print(mammal1.name, "is a", mammal1.species, "--", end=' ')
    mammal1.make_sound()

    print("Number of animals", mammal1.zoo_size())

    mammal2 = Mammal("Fruit bat", "Freddie", "Squeak!!", 180)
    print(mammal2.name, "is a", mammal2.species, "--", end=' ')
    mammal2.make_sound()

    print("Number of animals", mammal2.zoo_size())
    print("Number of animals", Mammal.zoo_size())

    mammal1.kill()
    print("Number of animals", Mammal.zoo_size())

    print("Gestation period of the", mammal1.species, "is", mammal1.gestation, "days")
    print("Gestation period of the", mammal2.species, "is", mammal2.gestation, "days")
```

① inherit from Animal

② add property to existing Animal properties

***mammal.py***

```
Bob is a African lion -- Roarrrr
Number of animals 1
Freddie is a Fruit bat -- Squeak!!
Number of animals 2
Number of animals 2
Number of animals 1
Gestation period of the African lion is 120 days
Gestation period of the Fruit bat is 180 days
```

# Untangling the nomenclature

There are many terms to distinguish the various parts of a Python program. This chart is an attempt to help you sort out what is what:

*Table 15. Objected-oriented Nomenclature*

| | |
|---|---|
| attribute | A variable or method that is part of a class or object |
| base class | A class from which other classes inherit |
| child class | Same as derived class |
| class | A Python module from which objects may be created |
| class method | A function that expects the class object as its first parameter. Such a function can be called from either the class itself or an instance of the class. Created with @classmethod decorator. |
| derived class | A class which inherits from some other class |
| function | An executable subprogram. |
| instance method | A function that expects the instance object, conventionally named 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 12 Exercises

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

*Part 1:*

Create a module which defines a class named **Knight**.

The initializer for the class should expect the knight's name as a parameter. Get the information from the file **knights.txt** to initialize the object.

The object should have these (read-only) properties:

*name*
*title*
*favorite_color*
*quest*
*comment*

**Example**

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

*Part 2:*

Create an application to use the **Knight** class created in part one. For each knight specified on the command line, create a **knight** object and print out the knight's name, favorite color, quest, and comment. Precede the name with the knight's title.

**Example output:**

```
Arthur Bedevere
Name: King Arthur
Favorite Color: blue
Quest: The Grail
Comment: King of the Britons

Name: Sir Bedevere
Favorite Color: red, no, blue!
Quest: The Grail
Comment: AARRRRRRRGGGGHH
```

# Appendix A: Where do I go from here?

## Resources for learning Python

These are from Jessica Garson, who, among other things, teaches Python classes at NYU. (Used with permission).

Run the script **where_do_i_go.py** to display a web page with live links.

Resources for Learning Python [https://dev.to/jessicagarson/resources-for-learning-python-hd6]

### Just getting started

Here are some resources that can help you get started learning how to code.

- Code Newbie Podcast [https://www.codenewbie.org/podcast]
- Dive into Python3 [http://www.diveintopython3.net]
- Learn Python the Hard Way [https://learnpythonthehardway.org/python3]
- Learn Python the Hard Way [https://learnpythonthehardway.org/python3]
- Automate the Boring Stuff with Python [https://automatetheboringstuff.com]
- Automate the Boring Stuff with Python [https://automatetheboringstuff.com]

### So you want to be a data scientist?

- Data Wrangling with Python [https://www.amazon.com/Data-Wrangling-Python-Tools-Easier/dp/1491948817]
- Data Analysis in Python [http://www.data-analysis-in-python.org/index.html]
- Titanic: Machine Learning from Disaster [https://www.kaggle.com/c/titanic/discussion/5105]
- Deep Learning with Python [https://www.manning.com/books/deep-learning-with-python]
- How to do X with Python [https://chrisalbon.com/]
- Machine Learning: A Probabilistic Prospective [https://www.amazon.com/Machine-Learning-Probabilistic-Perspective-Computation/dp/0262018020]

### So you want to write code for the web?

- Learn flask, some great resources are listed here [https://www.fullstackpython.com/flask.html]
- Django Polls Tutorial [https://docs.djangoproject.com/en/2.0/intro/tutorial01/]
- Hello Web App [https://www.amazon.com/Hello-Web-App-Learn-Build-ebook/dp/B00U5MMZ2E/ref=sr_1_1?ie=UTF8&qid=1510599119&sr=8-1&keywords=hello+web+app]
- Hello Web App Intermediate [https://www.amazon.com/Hello-Web-App-Intermediate-Concepts/dp/0986365920]

- Test-Driven-Development for Web Programming [https://www.obeythetestinggoat.com/pages/book.html#toc]
- 2 Scoops of Django [https://www.amazon.com/Two-Scoops-Django-1-11-Practices-ebook/dp/B076D5FKFX/ref=sr_1_1?s=books&ie=UTF8&qid=1510598897&sr=1-1&keywords=2+scoops+of+django]
- HTML and CSS: Design and Build Websites [https://www.amazon.com/HTML-CSS-Design-Build-Websites/dp/1118008189/ref=sr_1_1?ie=UTF8&qid=1510599157&sr=8-1&keywords=css+and+html]
- JavaScript and JQuery [https://www.amazon.com/JavaScript-JQuery-Interactive-Front-End-Development/dp/1118531647]

## Not sure yet, that's okay!

Here are some resources for self guided learning. I recommend trying to be very good at Python and the rest should figure itself out in time.

- Python 3 Crash Course [https://www.amazon.com/Python-Crash-Course-Hands-Project-Based/dp/1593276036]
- Base CS Podcast [https://www.codenewbie.org/basecs]
- Writing Idiomatic Python [https://www.amazon.com/Writing-Idiomatic-Python-Jeff-Knupp-ebook/dp/B00B5VXMRG]
- Fluent Python [https://www.amazon.com/dp/1491946008?aaxitk=o7.Y1C9z7oJp87fs3ev30Q&pd_rd_i=1491946008&hsa_cr_id=1406361870001]
- Pro Python [https://www.amazon.com/Pro-Python-Marty-Alchin/dp/1484203356/ref=sr_1_1?s=books&ie=UTF8&qid=1510598874&sr=1-1&keywords=pro+python]
- Refactoring [https://www.amazon.com/Refactoring-Improving-Design-Existing-Code/dp/0201485672/ref=sr_1_1?ie=UTF8&qid=1510598784&sr=8-1&keywords=refactoring+martin+fowler]
- Clean Code [https://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?s=books&ie=UTF8&qid=1510598926&sr=1-1&keywords=clean+code]
- Write music with Python, since that's my favorite way to learn a new language [https://github.com/reckoner165/soundmodular]

# Appendix B: Python Bibliography

| Title | Author | Publisher |
|---|---|---|
| **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 |

| Title | Author | Publisher |
|---|---|---|
| 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, 3nd. 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 | T J O'Connor | Syngress |
| Web Scraping with Python | Ryan Mitchell | O'Reilly & Assoc. |
| **Testing** | | |

| Title | Author | Publisher |
|---|---|---|
| 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: Best Practices for Django 1.11 | Daniel Roy Greenfeld, Audrey Roy Greenfeld | Two Scoops Press |
| Web Development with Django Cookbook | Aidas Bendoraitis | Packt Publishing |

# 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

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

person = 'Bob'
age = 22

print("{0} is {1} years old.".format(person, age))  ①
print("{0}, {0}, {0} your boat".format('row'))  ②
print("The {1}-year-old is {0}".format(person, age))  ③
print("{name} is {age} years old.".format(name=person, age=age))  ④
print()
print("{} is {} years old.".format(person, age))  ⑤
print("{name} is {} and his favorite color is {}".format(22, 'blue', name='Bob'))  ⑥
```

① Placeholders can be numbered

② Placeholders can be reused

③ They do not have to be in order (but usually are)

④ Selectors can be named

⑤ Empty selectors are autonumbered (but all selectors must either be empty or explicitly numbered)

⑥ 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**

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

person = 'Bob'
age = 22

print(f"{person} is {age} years old.")
print(f"The {age}-year-old is {person}.")
print()
```

*fmt_fstrings.py*

```
Bob is 22 years old.
The 22-year-old is Bob.
```

# 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

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

person = 'Bob'
value = 488
bigvalue = 3735928559
result = 234.5617282027

print('{:s}'.format(person))    ①
print('{name:s}'.format(name=person))    ②
print('{:d}'.format(value))    ③
print('{:b}'.format(value))    ④
print('{:o}'.format(value))    ⑤
print('{:x}'.format(value))    ⑥
print('{:X}'.format(bigvalue))    ⑦
print('{:f}'.format(result))    ⑧
print('{:.2f}'.format(result))    ⑨
```

① String

② String

③ Integer (displayed as decimal)

④ Integer (displayed as binary)

⑤ Integer (displayed as octal)

⑥ Integer (displayed as hex)

⑦ Integer (displayed as hex with uppercase digits)

⑧ Float (defaults to 6 places after the decimal point)

⑨ Float rounded to 2 decimal places

***fmt_types.py***

```
Bob
Bob
488
111101000
750
1e8
DEADBEEF
234.561728
234.56
```

*Table 16. Formatting Types*

| | |
|---|---|
| b | Binary – converts number to base 2 |
| c | Character – converts to corresponding character, like chr() |
| d | Decimal – outputs number in base 10 |
| e, E | Exponent notation. 'e' prints the number in scientific notation using the letter 'e' to indicate the exponent. 'E' is the same, except it uses the letter 'E' |
| f,F | Floating point. 'F' and 'f' are the same. |
| g | General format. For a given precision p >= 1, rounds the number to p significant digits and then formats the result in fixed-point or scientific notation, depending on magnitude. This is the default for numbers |
| G | Same as g, but upper-cases 'e', 'nan', and 'inf" |
| n | Same as d, but uses locale setting for number separators |
| o | Octal – converts number to base 8 |
| s | String format. This is the default type for strings |
| x, X | Hexadecimal – convert number to base 16; A-F match case of 'x' or 'X' |
| % | Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign. |

# Field Widths

- Specified as {0:width.precision}
- Width is really minimum width
- Precision is either maximum width or # decimal points

Fields can specify a minimum width by putting a number before the type. If the parameter is 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

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

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))     ①
print('[{:10s}]'.format(name))   ②
print('[{:3s}]'.format(name))    ③
print('[{:3.3s}]'.format(name))  ④
print()
print('[{:8d}]'.format(value))       ⑤
print('[{:8f}]'.format(value))       ⑥
print('[{:8f}]'.format(airspeed))    ⑦
print('[{:.2f}]'.format(airspeed))   ⑧
print('[{:8.3f}]'.format(airspeed))  ⑨
```

① Default format — no padding

② Left justify, 10 characters wide

③ Left justify, 3 characters wide, displays entire string

④ Left justify, 3 characters wide, truncates string to max width

⑤ Right justify, decimal, 8 characters wide (all numbers are right-justified by default)

⑥ Right justify int as float, 8 characters wide

⑦ Right justify float as float, 8 characters wide

⑧ Right justify, float, 3 decimal places, no maximum width

⑨ 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

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

name = 'Ann'
value = 12345
nvalue = -12345

①
print('[{0:10s}]'.format(name))     ②
print('[{0:<10s}]'.format(name))    ③
print('[{0:>10s}]'.format(name))    ④
print('[{0:^10s}]'.format(name))    ⑤
print()
print('[{0:10d}] [{1:10d}]'.format(value, nvalue))    ⑥
print('[{0:>10d}] [{1:>10d}]'.format(value, nvalue))  ⑦
print('[{0:<10d}] [{1:<10d}]'.format(value, nvalue))  ⑧
print('[{0:^10d}] [{1:^10d}]'.format(value, nvalue))  ⑨
print('[{0:=10d}] [{1:=10d}]'.format(value, nvalue))  ⑩
```

① note: all of the following print in a field 10 characters widedd

② Default (left) alignment

③ Explicit left alignment

④ Right alignment

⑤ Centered

⑥ Default (right) alignment

⑦ Explicit right alignment

⑧ Left alignment

⑨ Centered

⑩ 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**

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

name = 'Ann'
value = 123

print('[{:>10s}]'.format(name))     ①
print('[{:.>10s}]'.format(name))    ②
print('[{:->10s}]'.format(name))     ③
print('[{:.10s}]'.format(name))     ④
print()
print('[{:10d}]'.format(value))     ⑤
print('[{:010d}]'.format(value))    ⑥
print('[{:_>10d}]'.format(value))   ⑦
print('[{:+>10d}]'.format(value))   ⑧
```

① Right justify string, pad with space (default)

② Right justify string, pad with '.'

③ Right justify string, pad with '-'

④ Left justify string, pad with '.'

⑤ Right justify number, pad with space (default

⑥ Right justify number, pad with zeroes

⑦ Right justfy, pad with '_' ('>' required)

⑧ Right justfy, pad with '+' ('>' required)

### *fmt_fill.py*

```
[       Ann]
[.......Ann]
[-------Ann]
[Ann]

[       123]
[0000000123]
[_______123]
[+++++++123]
```

# Signed numbers

- Can pad with any character except '{}'
- Sign can be '+', '-', or space
- Only appropriate for numeric types

The sign character follows the alignment character, and can be plus, minus, or space.

A plus sign means always display + or – preceding non-zero numbers.

A minus sign means only display a sign for negative numbers.

A space means display a – for negative numbers and a space for positive numbers.

## Example

### fmt_signed.py

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

values = 123, -321, 14, -2, 0

for value in values:
    print("default: |{:d}|".format(value))  ①
print()

for value in values:
    print("   plus: |{:+d}|".format(value))  ②
print()

for value in values:
    print("  minus: |{:-d}|".format(value)) ③
print()

for value in values:
    print("  space: |{: d}|".format(value)) ④
print()
```

① default (pipe symbols just to show white space)

② plus sign puts '+' on positive numbers (and zero) and '-' on negative

③ minus sign only puts '-' on negative numbers

④ space puts '-' on negative numbers and space on others

***fmt_signed.py***

```
default: |123|
default: |-321|
default: |14|
default: |-2|
default: |0|

   plus: |+123|
   plus: |-321|
   plus: |+14|
   plus: |-2|
   plus: |+0|

  minus: |123|
  minus: |-321|
  minus: |14|
  minus: |-2|
  minus: |0|

  space: | 123|
  space: |-321|
  space: | 14|
  space: |-2|
  space: | 0|
```

# Parameter Attributes

- Specify elements or properties in template
- No need to repeat parameters
- Works with sequences, mappings, and objects

When specifying container variables as parameters, you can select elements in the format rather than in the parameter list. For sequences or dictionaries, index on the selector with []. For object attributes, access the attribute from the selector with . (period).

## Example

**fmt_attrib.py**

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

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))  ①
print('{f[0]} {f[2]}'.format(f=fruits))  ②
print()
print('{0[0]} {0[2]}'.format(values))  ③
print()
print('{0[Palin]} {0[Cleese]}'.format(pythons))  ④
print('{names[Palin]} {names[Cleese]}'.format(names=pythons))  ⑤
print()
print('{0.month}-{0.day}-{0.year}'.format(dday))  ⑥
```

① select from tuple

② named parameter + select from tuple

③ Select from list

④ select from dict

⑤ named parameter + select from dict

⑥ 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**

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

from datetime import datetime

event = datetime(2016, 1, 2, 3, 4, 5)

print(event)  ①
print()

print("Date is {0:%m}/{0:%d}/{0:%y}".format(event))  ②
print("Date is {:%m/%d/%y}".format(event))  ③
print("Date is {:%A, %B %d, %Y}".format(event))  ④
```

① Default string version of date

② Use three placeholders for month, day, year

③ Format month, day, year with a single placeholder

④ 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 17. 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 instrptime() 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

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

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(  ①
    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(  ②
        FIRST_NAME,
        LAST_NAME,
        width=WIDTH,
        pad=PAD,
        align=alignment,
    ))
```

① value of WIDTH used in format spec

② values of PAD, WIDTH, ALIGNMENTS used in format spec

***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**

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

'''Demonstrate misc formatting'''

big_number = 2303902390239

print("Big number: {:,d}".format(big_number))  ①
print()

value = 27

print("Binary: {:#010b}".format(value))  ②
print("Octal:  {:#010o}".format(value))  ③
print("Hex:    {:#010x}".format(value))  ④
print()
```

① Add commas for readability

② Binary format with leading 0b

③ Octal format with leading 0o

④ Hexadecimal format with leading 0x

***fmt_misc.py***

```
Big number: 2,303,902,390,239

Binary: 0b00011011
Octal:  0o00000033
Hex:    0x0000001b
```

# Index