Red Hat
Training and Certification

Student Workbook (ROLE)

Red Hat Enterprise Linux 9.0 AD141

Red Hat Training Presents - Introduction to Python Programming

Edition 2

Red Hat
Learning Community

Join a community dedicated to learning open source

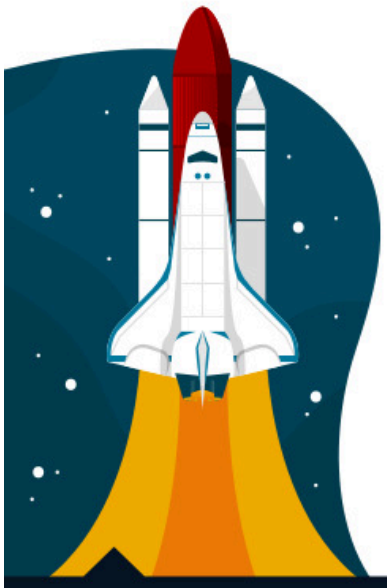
The Red Hat® Learning Community is a collaborative platform for users to accelerate open source skill adoption while working with Red Hat products and experts.

Network with tens of thousands of comunity members

Engage in thousands of active conversations and posts

Join and interact with hundreds of certified training instructors

Unlock badges as you participate and accomplish new goals



This knowledge-sharing platform creates a space where learners can connect, ask questions, and collaborate with other open source practitioners.

Access free Red Hat training videos

Discover the latest Red Hat Training and Certification news

Connect with your instructor - and your classmates - before, after, and during your training course.

Join peers as you explore Red Hat products

Join the conversation learn.redhat.com

Red Hat

Red Hat Training Presents - Introduction to Python Programming

Red Hat Enterprise Linux 9.0 AD141
Red Hat Training Presents - Introduction to Python Programming
Edition 2 20221213
Publication date 20221213

Authors:	Antonio Mari Romero, Aykut Bulgu, Eduardo Ramírez Ronco, Jaime Ramírez Castillo, Pablo Solar Vilariño, Randy Thomas
Course Architects:	Ravi Srinivasan, Zachary Gutterman
DevOps Engineer:	Richard Allred
Editor:	Sam Ffrench

Contributors: **Jordi Sola, Nicolette Lucas, Shatakshi Jain, Marek Czernek**

Document Conventions

This section describes various conventions and practices that are used throughout all Red Hat Training courses.

Admonitions

Red Hat Training courses use the following admonitions:

References

These describe where to find external documentation that is relevant to a subject.

Note

Notes are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on something that makes your life easier.

Important

Important sections provide details of information that is easily missed: configuration changes that apply only to the current session, or services that need restarting before an update applies. Ignoring these admonitions will not cause data loss, but might cause irritation and frustration.

Warning

Do not ignore warnings. Ignoring these admonitions will most likely cause data loss.

Inclusive Language

Red Hat Training is currently reviewing its use of language in various areas to help remove any potentially offensive terms. This is an ongoing process and requires alignment with the products and services that are covered in Red Hat Training courses. Red Hat appreciates your patience during this process.

Introduction

Red Hat Training Presents - Introduction to Python Programming

Python is a popular programming language used by system administrators, developers, and data scientists to create web applications, custom Red Hat Ansible Automation modules, perform statistical analysis, and train AI/ML models. This course introduces the Python language and teaches fundamental concepts like control flow, loops, data structures, functions, file I/O, regular expressions, parsing JSON, and debugging. This course is based on Python 3 and RHEL 9.

Course Objectives

- How to install Python and set up an IDE to work with scripts efficiently.
- Basics of Python syntax, functions and data types.
- How to debug Python scripts using the Python debugger (pdb).
- Use Python data structures like dictionaries, sets, tuples and lists to handle compound data.
- Learn Object-oriented programming in Python and Exception Handling.
- How to read and write files in Python and parse JSON data.
- Use powerful regular expressions in Python to manipulate text.
- How to effectively structure large Python programs using modules and namespaces.
- How to use third-party libraries using the pip CLI tool.

Audience

- System administrators and DevOps personnel who want to use Python to automate operating system tasks.
- Developers from other programming languages who want to learn Python for writing applications.
- AI/ML, data scientists, and engineers who want to use Python for data analysis and machine learning.

Prerequisites

- The are no prerequisites for this course.

Orientation to the Classroom Environment

In this chapter, a high-level overview is provided of the Python programming environment. This chapter presents various ways of creating and executing Python scripts and navigating the documentation and help system.

Python can be described as:

- Interpreted, as opposed to compiled.
- Object oriented, as opposed to procedure oriented.
- Dynamically typed, as opposed to statically typed.

Some of Python's strengths include the following.

- It is easy to learn.
- It is efficient at processing text data.
- It is modular and easily extensible via Python Modules.
- It supports object-oriented programming.

Some use cases of Python include:

- Creation of web applications.
- Writing of applications that interact with the underlying operating system.
- Building of tools that process and analyze big amounts of data.
- Writing of Ansible modules.
- Creation of machine learning workflows you can test, train, and deploy to Red Hat OpenShift Data Science.

Python was created by Guido van Rossum in 1990 and released to the public domain in 1991.

- In 1994, comp.lang.python was formed.
  - This is a dedicated Usenet newsgroup for general discussions and questions about Python.

References

- A brief history and the terms and conditions of the use of Python can be found at the following URL
  `https://docs.python.org/3/license.html`
- A brief history, written by Guido himself, of what started it all can be found under the section "Why was Python created in the first place?" at the following URL
  `https://docs.python.org/3/faq/general.html`

Installing Python

This course uses Python 3.9, which is the default, preinstalled Python version on Red Hat Enterprise Linux (RHEL) 9.

Python 3 includes significant changes to the language that make it incompatible with Python 2.

References

An overview of the what's new can be found at the following URL

`https://docs.python.org/3/whatsnew/index.html`

Some of the less disruptive changes in Python 3 have been backported to Python 2.6.x and 2.7.x.

- Therefore, Python 2 will correctly interpret Python 3 code in some, but not all, cases.
  - Throughout this course, various differences in the two versions of the language may be pointed out by the instructor.

Python runs on Windows, Linux/Unix, Mac OS X.

References

Python can be downloaded from the following URL

`http://www.python.org/download`

- Microsoft Windows users can download a standard installer to install Python on their machine.
- Most Linux distributions come with Python preinstalled. However, the preinstalled version is often an outdated version.
- Python also comes preinstalled on the Mac OS X operating system. However, similar to Linux distributions, it is often an old version.

For this course, the classroom environment has a working version of the RHEL 9 operating system with the following version of Python installed, and ready for use.

- Python 3.9.10 (main, Feb 9 2022, 00:00:00)

Orientation to the Classroom Environment

The Red Hat Online Learning (ROL) platform provides a RHEL 9 workstation environment in the cloud, which you can connect to remotely from your browser. To use the cloud workstation, click the **CREATE** button in the **Lab Environment** tab in the ROL interface.

In this environment, the main computer system used for learning activities is `workstation`. All student computer systems have a standard user account, `student`, which has the password `student`. The root password on all student systems is `redhat`.

The required tools are preinstalled in the `Cloud Workstation` classroom environment, which also includes VSCodium, a text editor that includes useful development features.

Executing Python from the Command Line

Executing Python code can be done from the command line, Python's interactive interpreter, or an Integrated Development Environment (IDE). To execute a Python program from the command line, do the following.

1. First create a Python script with a text editor and name it with the standard `.py` file extension.

2. Once this file has been created, execute the file by using one of the various python commands.

On a Linux system, the python command may refer to a version of Python 2 or Python 3.

- Python recommends the following convention to ensure that Python scripts can continue to be portable across Linux systems, regardless of the version of the Python interpreter.
  - `python2` will refer to some installed version of Python 2.
  - `python3` will refer to some installed version of Python 3.
  - `python` will refer to either version of Python for a given Linux distribution and its configuration.
- The above recommendation comes in the form of a Python Enhancement Proposal (PEP).
  - The specific PEP for the above recommendation is PEP 394.
  - More information about PEPs can be found at the following URL `https://www.python.org/dev/peps/`

The scripts provided in this course are written for Python 3 and as such rely on the `python3` command.

- Many of the scripts will include the following character sequence (known as a "shebang") as the first line of each source file.

```
#!/usr/bin/env python3
```

A simple example script is shown below.

hello.py

```
#!/usr/bin/env python3
print("Hello World")
```

- The script would then be executed as follows:

```
$ python3 hello.py
Hello World
$
```

A more Linux-like approach is to first make the file executable and then simply execute it from the command line as shown below.

```
$ chmod u+x hello.py
$ ./hello.py
Hello World
$
```

- The `chmod u+x hello.py` is used to grant only the user (owner) of the file execution permissions.
- The `./hello.py` is used to execute the file.

This first program is offered merely to demonstrate the execution of a Python program from the command line.

- The `print` function sends data to the standard output.
  - In Python 3, parentheses are required for function arguments.
  - This is not the case in Python 2 versions where `print` was a statement as opposed to a function.

Executing Python from an Interactive Python Shell

The Python interpreter, sometimes called an interactive Python shell, allows Python commands to be executed interactively.

- Interactively entering an expression will output the result of executing the expression without the need to call the `print()` function.
- This behavior is only available in the interactive shell and would not produce output if the same statement was run within a script.

Here is a small example session executing the Python interpreter in interactive mode.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Characters inside double quotes are a string")
Characters inside double quotes are a string
>>> print('Single quotes are also used to represent a string')
Single quotes are also used to represent a string
>>> 3 + 6
9
>>> result = 3 + 6
>>> value = 2 ** 10
>>> print("Passing multiple arguments", result, value)
Passing multiple arguments 9 1024
>>> print(value / result, value // result)
113.77777777777777 113
>>> # The pound sign makes this a comment
    #     The comment is here to indicate that the following
    #     defines a function that takes a parameter
    def some_function(param):
        print("The param is:", param)
```

```
>>> # The next statement calls the above function
    some_function(value)
The param is: 1024
>>> exit()
$
```

IDLE

Python provides a graphical tool named **IDLE** (Integrated Development and Learning Environment).

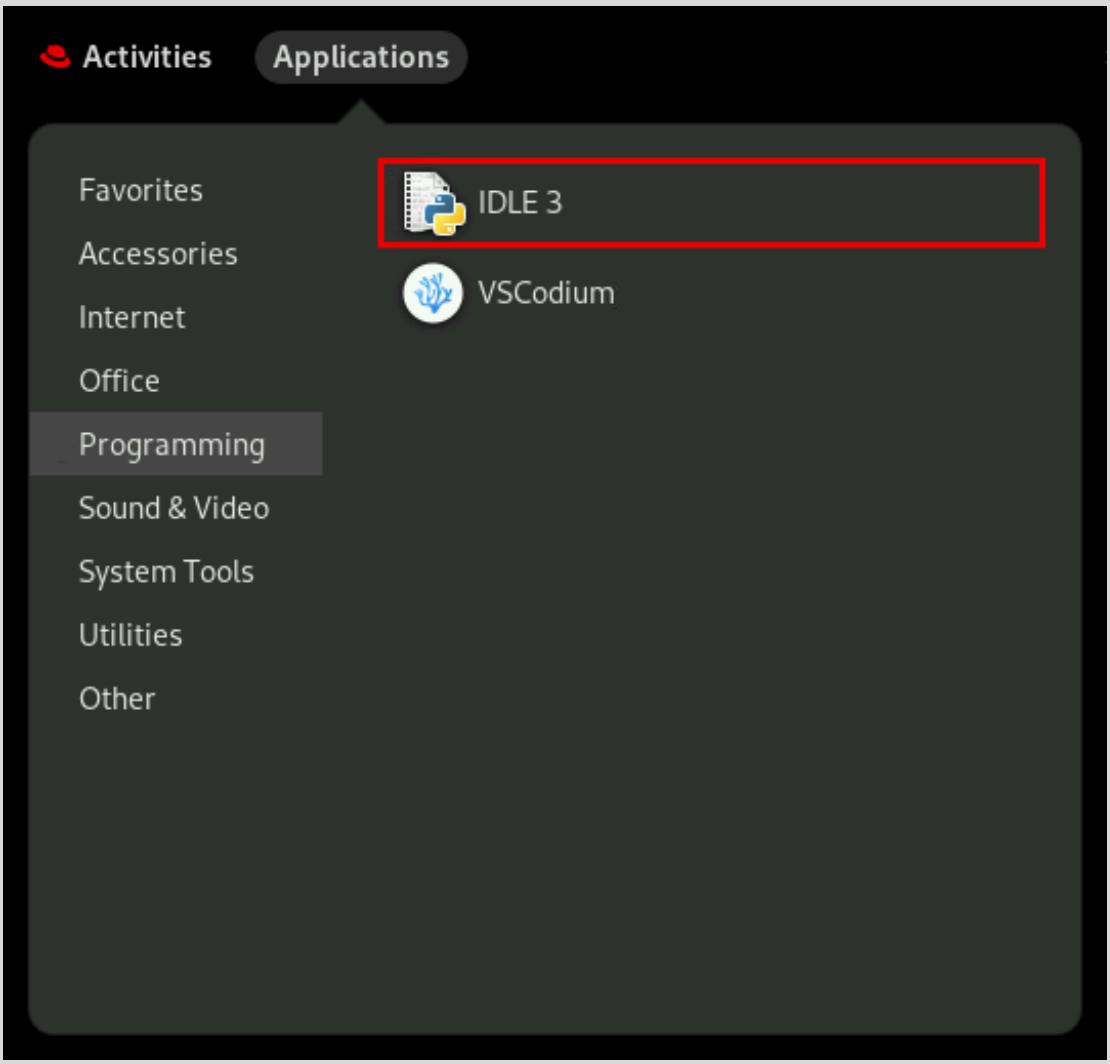
- Some of the features offered with IDLE are:
  - It is coded in 100% pure Python, using the `tkinter` GUI toolkit.
  - It is cross-platform: works on Windows, Unix, and Mac OS X.
  - It offers an interactive Python shell window with colorizing of code input, output, and error messages.

IDLE can be started in various ways:

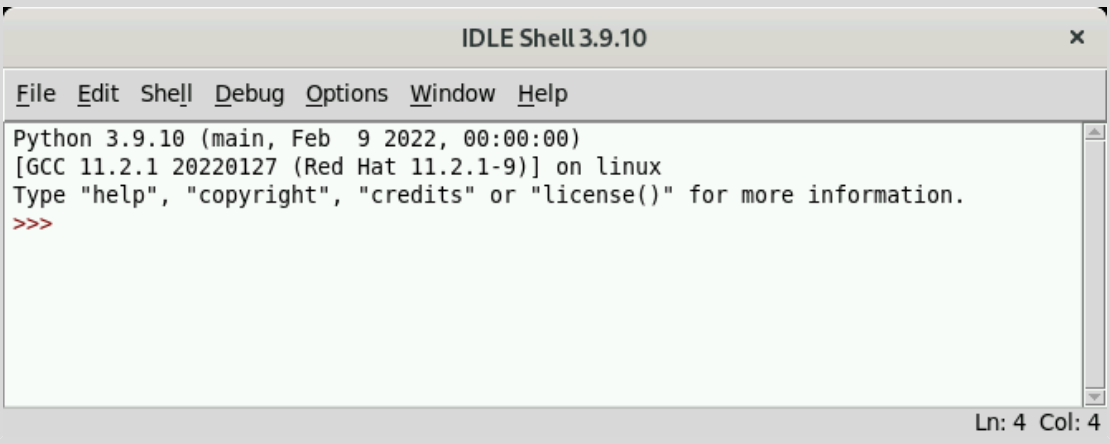
- Starting IDLE from the command line.

```
$ idle3
$
```

- Starting IDLE from the Applications menu.



- The IDLE application will appear as shown below.



Additional Editors and IDEs

The choice of a Python editor is largely a personal choice.

References

A list of common Python editors for various operating systems can be found at the following URL

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

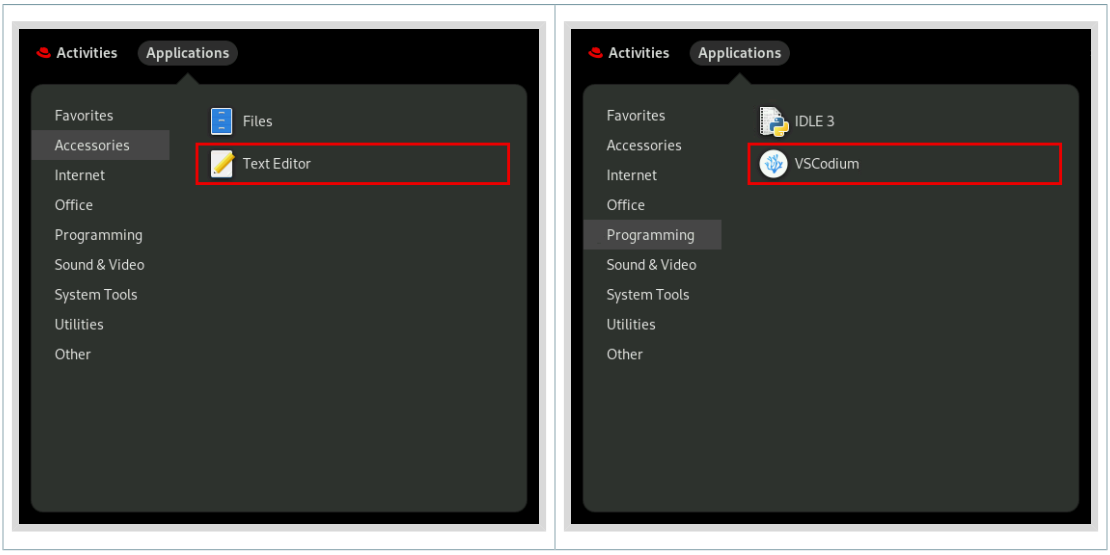
As we will see throughout the course, Python relies heavily on the indention requirements of the source code to define blocks of code.

- Integrated Development Environments (IDEs) are usually configured to handle the spacing and indenting issues automatically for the developer.

The following editors are available for use on the classroom machines.

- Text editors such as `vim`, and `gedit` are available.
- And a more robust IDE, named VSCodium, is available.
  - The Python extension has been added to the VSCodium installation.

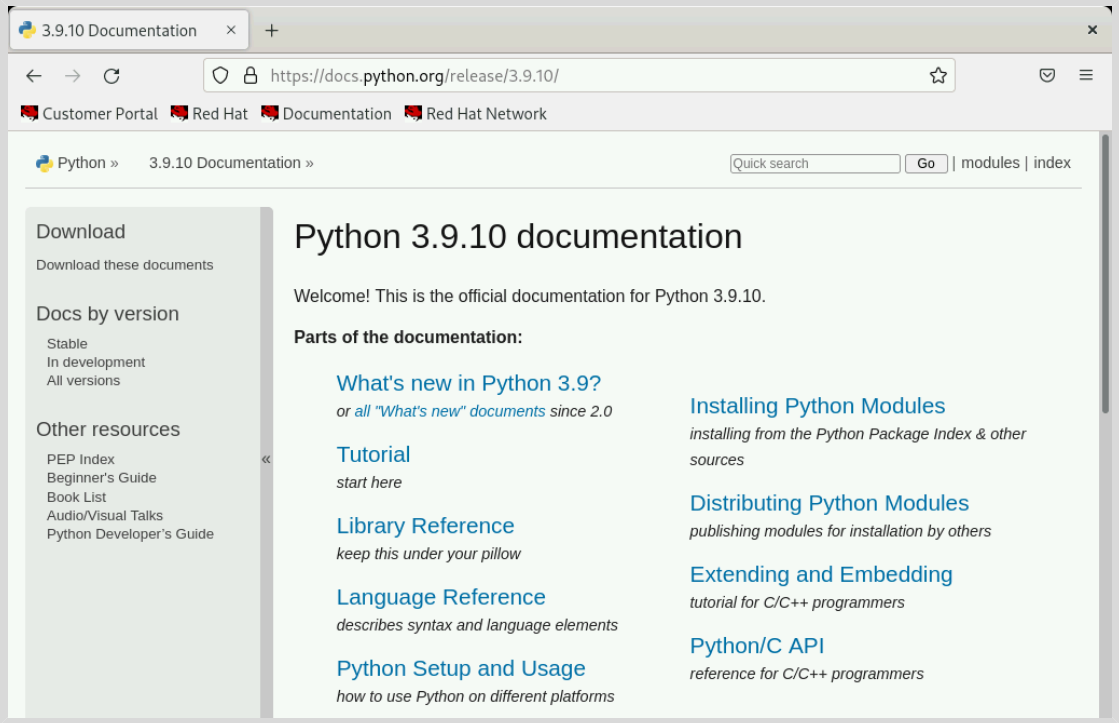
Many of the above programs can be launched in the command line or from the menu launcher in the top left corner of the screen as shown below.



Python Documentation

There are various ways in which the Python programmer can get help from the Python documentation.

- A good starting point is one of the following URLs:
  - The most recent version can always be found here `https://docs.python.org/3/`
  - The documentation for the specific version of Python being used can be found here `https://www.python.org/doc/versions/`
- At the time of this writing, choosing version 3.9 from the list above will present the following:



- The **Library Reference** and **Language Reference** links above are often useful to both new and seasoned Python developers.
- The **Python Setup and Usage** provides additional information for using Python on a specific operating system.

Getting Help

In addition to the online Python documentation, the interactive Python shell can provide additional help.

- Recall the Python shell can be started with either of the following.
  - The `python3` command for a text based environment.
  - The `idle` command for a graphical environment.
- Once the Python shell is available, typing `help()` will start the Python help utility.
  - To see the math functions, type `math`.
  - To see the string methods, type `str`.
  - To see all documented topics, type `topics`.

Here is an example of using the help utility.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> help()
```

```
Welcome to Python 3.9's help utility!

If this is your first time using Python, you should definitely check out
the tutorial on the Internet at https://docs.python.org/3.9/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing
Python programs and using Python modules.  To quit this help utility and
return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type
"modules", "keywords", "symbols", or "topics".  Each module also comes
with a one-line summary of what it does; to list the modules whose name
or summary contain a given string such as "spam", type "modules spam".

help> math
```

- Upon typing math at the `help>` prompt, the documentation will be displayed as shown on the following page:

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available. It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the inverse hyperbolic cosine of x.

    asin(...)
        asin(x)

:
```

- Linux users might recognize the above view as a "man page", a form of documenting software.
  - The up, down, page up and page down keys on the keyboard can be used to scroll through the help screen shown above.
  - Typing the letter 'q' will quit out of the help screen above.
  - Typing `quit` at the `help>` prompt will exit the help utility back to the interactive Python shell prompt >>>

- From there, typing `quit()` or `exit()` will exit the Python shell.

Alternatively, help can be obtained at the interactive Python prompt >>> by passing information to the help function as shown below.

```
help('math')
help('str')
```

Python Keywords

Several keywords shown so far are used for special purposes in the Python language.

- These words cannot be used as the names of variables or functions.
- They are listed here for reference.

Python Keywords

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

The list above can be generated by executing:

- `help('keywords')` at the >>> prompt or
- `keywords` at the `help>` prompt.

Only a few of these keywords have been introduced, but all will be used in this course. Each will be explained when introduced.

Naming Conventions

A Python **identifier** is a name that is used to identify any of the following:

- Variables
- Functions/Methods
- Classes
- Modules

An identifier must start with

- A letter of the alphabet or

- The underline _ character.

This can be followed by any number of letters, digits and/or the _ character.

- Identifier names cannot consist of any other characters.

Variable names and function names typically begin with a lowercase letter, while class names typically start with an uppercase letter.

There are a lot of different naming styles within the language.

- PEP 8 is designed to standardize coding conventions for Python.

References

A complete list of naming conventions can be found within PEP 8 here
`http://www.python.org/dev/peps/pep-0008/#naming-conventions`

Dynamic Types

The following program emphasizes the dynamic type system of Python.

- In a dynamically typed language, a variable is simply a value bound to a name.
- The value of what is being referenced by the variable has a type like `int` or `float` or `str`, but the variable itself has no type.

datatypes.py

```
#!/usr/bin/env python3
x = 10
tab = " \t"

print(x, tab, type(x), tab, id(x))
y = x
print(y, tab, type(y), tab, id(y))
x = 25.7
print(x, tab, type(x), tab, id(x))
x = "Hello"
print(x, tab, type(x), tab, id(x))
```

- The output from running the program above is shown below.

```
$ python3 datatypes.py
10      <class 'int'>           10943296
10      <class 'int'>           10943296
25.7    <class 'float'>         139761989022296
Hello   <class 'str'>           139761987959416
$
```

The built-in `type()` function returns the data type of the object that is referenced by a particular variable.

- Notice that the type of the reference is bound dynamically as the program is executed.

The built-in `id()` function returns a unique identifier of an object that is referenced by a particular variable.

- There is only a single `int` object `10` being created and both the variables x and y reference the same object. This is evident in that they both produce the same id.

Exercises

Exercise 1

Create a folder called `AD141` in your home directory. Within the `~/AD141` directory, clone the `AD141-apps` repository to your workstation.

```
[student@workstation AD141]$ git clone \
https://github.com/RedHatTraining/AD141-apps.git
Cloning into 'AD141-apps'...

...output omitted...
```

Exercise 2

If not already done so, run the `python3` command to open an interactive Python Shell at the command line and experiment with some Python statements.

Exercise 3

Launch IDLE and experiment some more with some Python statements.

- Also experiment with IDLE in opening an existing file and editing it such as the `hello.py` or `datatypes.py` from this chapter. You can find those files in the `~/AD141/AD141-apps/overview/examples` directory.

Exercise 4

Within the `~/AD141` directory, create a subdirectory named `mywork`. That directory would be a good place to create all files pertaining to the exercises throughout the course.

- Start by creating a file named `first.py`.
  - In that file, assign values to variables and then perform a few operations with them, storing the results in new variables.
  - Print the values of those variables.
  - Basically, create similar statements in this exercise as done in exercises 1 and 2 above, but this time place them in a single file that can easily be edited and run over and over as a script.

The solution files for this exercise are in the `AD141-apps` repository, within the `overview/solutions` directory.

Exercise 5

Run the script created in exercise 3 above in the following two ways:

1. Run it from the command line as: `python3 first.py`

2. Make the file executable such that it can be run as follows from the command line: `./first.py`

References

- A brief history and the terms and conditions of the use of Python can be found at the following URL
  `https://docs.python.org/3/license.html`
- A brief history, written by Guido himself, of what started it all can be found under the section "Why was Python created in the first place?" at the following URL
  `https://docs.python.org/3/faq/general.html`
- An overview of the what's new can be found at the following URL
  `https://docs.python.org/3/whatsnew/index.html`
- Python can be downloaded from the following URL
  `http://www.python.org/download`
- A list of common Python editors for various operating systems can be found at the following URL
  `http://wiki.python.org/moin/PythonEditors`
- The most recent version of the documentation always be found here
  `https://docs.python.org/3/`
- The documentation for the specific version of Python being used can be found here
  `https://www.python.org/doc/versions/`
- A complete list of naming conventions can be found within PEP 8 here
  `http://www.python.org/dev/peps/pep-0008/#naming-conventions`

Chapter 1

Basic Python Syntax

Goal

Describe the fundamental syntax of the Python language, and develop simple applications.

Objectives

- Use correct Python syntax in Python programs.
- Use basic Python input and output functions properly.
- Write Python programs using the standard numeric data types and their operators.
- Use strings and their methods in Python programs.
- Convert between numeric and string data types.

Sections

- Basic Python Syntax

Basic Python Syntax

Objectives

- Use correct Python syntax in Python programs.
- Use basic Python input and output functions properly.
- Write Python programs using the standard numeric data types and their operators.
- Use strings and their methods in Python programs.
- Convert between numeric and string data types.

Basic Syntax

This chapter deals with many of the issues relating to the fundamental syntax of the Python language.

Python is case sensitive. Therefore, the following two variables are different.

```
the_Person = "Him"
the_person = "Her"
```

Python statements do not end with a semicolon, but you can use one to separate two statements on the same line.

```
length = 10; width = 5
area = length * width
print(area)
```

All lines in a Python application must begin in the first column, unless the statement is in the body of a loop, conditional, function, or class definition.

- Python relies on indentation to determine the control flow structure of an application.
- As we introduce control structures, we will revisit the indentation rules.
- Line indentation is typically a multiple of four spaces.

You can continue a long statement by placing the \ character at the end of the line.

- When used in this fashion it is often referred to as the line continuation character.
- Several examples of its use are shown below.

```
message = "This is a long string just to \
illustrate continuation across multiple lines"

result = 500 * 2 + \
400 * 3
```

Statements continue onto the following line(s) when reaching the end of the line before the closing character for any of the following pairs of characters.

```
() [] {}
```

The following snippet of code uses the `print()` function with multiple arguments.

```
data = "This is just a string of text"
print("This is the first argument",
      "This is the second argument",
      data)
```

We indicate the arguments to the `print()` function using commas, and Python ignores the additional white spaces.

Although not recommended, Python allows any number of white space characters between the opening and closing parenthesis, which do not need to comply with indentation rules.

Simple Output

We have demonstrated the `print()` function in several forms in previous examples.

- This function takes zero or more arguments, separated by commas, and displays the data.
- The default behavior of the function is:
  - When there is more than one argument, `print()` displays each of the arguments separated by a single space.
  - `print()` displays a new line after the last parameter.

You can alter the default behavior by providing the following optional named arguments.

- `sep`
  - Separator character to display between arguments.
- `end`
  - Character to display after the last parameter.
- `file`
  - File to output the arguments.
  - Use of this named argument will be shown in a later chapter.

Although a more detailed explanation of named arguments will be discussed in a later chapter, the example below demonstrates their use with the `print()` function.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello", 123, "Goodbye", 5 + 9, sep=":", end="###")
Hello:123:Goodbye:14###>>> exit()
$
```

Simple Input

The `input()` function provides a mechanism to obtain user input at runtime from the keyboard.

This function has several characteristics:

- Accepts an optional prompt string as an argument.
- Adds no spaces or other formatting characters.
- Automatically removes the trailing newline from the keyboard input.
- Always returns a string.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> response = input("Please enter some text:")
Please enter some text:Hello
>>> response = input("Please enter some text:\n")
Please enter some text:
Goodbye
>>> print("The last response is", response, sep=":")
The last response is:Goodbye
>>> exit()
$
```

The following example shows how the function returns a string even when the user introduces numbers:

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> response = input("Please enter a number: ")
Please enter a number: 456
>>> print("The response of", response, "is of type", type(response))
The response of 456 is of type <class 'str'>
>>> exit()
$
```

Note

Converting the string to a number will be discussed later in this chapter.

Comments

Comments make source code more understandable and maintainable. Comments follow these general rules:

- Single-line comments begin with the # character and extend to the end of the physical line.
- An inline comment is a comment on the same line as a statement.
- PEP 8 suggests inline comments should be separated by at least two spaces from the statement.
- You can use triple-quoted strings (""") to write multi-line comments.

The following code demonstrates a few examples of comments within the source code.

comments.py

```
#!/usr/bin/env python3
#
# This line and the single # character above are comments.
# This line is another comment.

number = 10                  # This is an inline comment
text = "hello there!"        # A simple greeting
data = "This # is not a comment "

# Sometimes comments are used to temporarily comment out
# sections of code as shown below. Although if the
# statements are actually not needed it would be better
# practice to remove the statements from the code entirely
# as opposed to commenting them out

# print(number)
# print(text)
# print(data)
```

Numbers

Python supports three distinct numeric data types.

- Integers, Floating Point Numbers, and Complex Numbers

Use the following table to verify the numerical operators precedence:

Numerical Operations

Operation	Result
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

Operation	Result
`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
`divmod(x,y)`	The pair `(x // y, x % y)`
`pow(x, y)` or `x ** y`	x to the power y

Python can encode literal values in bases different than decimal. The following example shows some examples:

Base	Value	Decimal Value
Hexadecimal	`x = 0xff`	255
Octal	`y = 0o77`	63
Binary	`z = 0b111`	7

References

You can find more information about numeric data types in section 4.4 of the following URL:

`https://docs.python.org/3/library/stdtypes.html`

Strings

A string in Python is an immutable sequence of zero or more characters.

- Literal string values may be enclosed in single or double quotes.
- Literal strings can span multiple lines in several ways.
  - Using the line continuation character (`\`) as the last character.
  - Surrounding the string within a pair of matching triple-quotes: either double quotes(`"""`) or single quotes(`'''`).
- Literal strings may also be prefixed with a letter `r` or `R`.
  - These are referred to as raw strings and use different rules for backslash escape sequences.

The following is a list that represents all the escape sequences you can use in literal strings to represent special characters:

Escape Sequences

Sequence	Character/Meaning
\newline	Line continuation
\\	Backslash
\'	Single quote
\"	Double quote
\a	ASCII Bell (BEL)
\b	Backspace
\f	Form feed
\n	Linefeed
\r	Carriage Return
\t	Horizontal Tab
\v	Vertical Tab
\ooo	ASCII character (octal value ooo)
\xhhh	ASCII character (hex value hhh)
\uxxxx	Unicode Character with 16-bit hex value xxxx
\Uxxxxxxxx	Unicode Character with 32-bit hex value xxxxxxxx

The following example demonstrates the creation of literal strings by using the various techniques described in this chapter.

string_literals.py

```
#!/usr/bin/env python3
print("This is a literal string", 'and so is this')
print('"Double quotes" inside of single quotes')
print("'Single quotes' inside of double quotes")
print("A double quote \" inside double quotes")
print(r"A double quote \" inside a raw literal string")
print("A as unicode: \x41")

spades = """Royal Straight Flush \
\U0001F0A1 \U0001F0AE \U0001F0AD \U0001F0AB \U0001F0AA
"""
print(spades)

diamonds = """Royal Straight Flush \
```

```
\U0001F0C1 \U0001F0CE \U0001F0CD \U0001F0CB \U0001F0CA
"""
print(diamonds)
```

Running the above program produces the following output:

Note that while Python 3 fully supports Unicode characters, you still need a font supporting Unicode characters to produce this output.

String Methods

Python represents strings by using a class named `str`.

- If you are not familiar with Object Oriented Programming, you can think of a class as a new data type, and an object as an instance of that data type.
- Later in this course we will discuss Object Oriented Programming in more depth.

The `str` class has an abundance of methods defined within it.

- While some of the methods return Boolean values such as `True` or `False`, other methods return a modified version of the string on which they operate.

string_methods.py

```
#!/usr/bin/env python3
url = "www.redhat.com"
result = url.startswith("www")
print(result)
print(url.endswith(".org"))
new_url = url.upper()
print("ORIGINAL", url, "RETURNED", new_url)
```

When run, the preceding program outputs:

```
$ python3 string_methods.py
True
False
ORIGINAL www.redhat.com RETURNED WWW.REDHAT.COM
$
```

The following string methods return a Boolean indicating if the characters in the string are of a specific kind.

string_types.py

```
#!/usr/bin/env python3
print("AbCDe".isalpha(), "AbCd123".isalpha())   # True False
print("123".isnumeric(), "12.3".isnumeric())    # True False
print("  \t\n".isspace(), "a b\t\n".isspace())  # True False
print("ABCD".isupper(), "abcd".isupper())       # True False
```

The following snippet shows methods to search, replace, and split strings.

more_strings.py

```
#!/usr/bin/env python3

word = "is"
sentence = "The capital of Mississippi is Jackson."
position = sentence.find(word)
print("First:", position, "\t2nd: ", sentence.find(word, position + 1))
print(sentence.find(word, 8, 12))
print("The word '", word, "' appears", sentence.count(word), "times.\n")
print("Right Justified:", word.rjust(15), "|")
print(" Left Justified:", word.ljust(15, "*"), "|")

data = "1 4 1 1abc"
print("data:", data)
print("replace all:", data.replace("1", "0"))
print("replace two:", data.replace("1", "0", 2))

pieces = data.split(' ')
print(data)
print("pieces is of type:", type(pieces))
print(pieces)
```

The preceding programs outputs this:

```
$ python3 more_strings.py
First: 16  2nd:  19
-1
The word ' is ' appears 3 times.

Right Justified:              is |
 Left Justified: is************* |
data: 1 4 1 1abc
replace all: 0 4 0 0abc
replace two: 0 4 0 1abc
1 4 1 1abc
pieces is of type: <class 'list'>
['1', '4', '1', '1abc']
$
```

The following methods strip white space characters from a string.

whitespace.py

```
#!/usr/bin/env python3
# The following String has 3 sets of 2 spaces in it
data = "\t  \nabc  def\t  \n"

# The strip method removes leading and trailing whitespace
result = data.strip()
print(len(data), ":", len(result))

# The rstrip method removes trailing whitespace
```

```
result = data.rstrip()
print(len(data), ":", len(result))

# The lstrip method removes leading whitespace
result = data.lstrip()
print(len(data), ":", len(result))
```

The preceding code utilizes the built-in `len()` function to determine the length of each string and produces the following output:

```
$ python3 whitespace.py
16 : 8
16 : 12
16 : 12
$
```

References

You can find a complete list of string methods and their definitions in the Python documentation at the following URL:

`https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str`

Sequence Operations

Python strings are a sequence type (positionally ordered), and as such, share certain operations available to all types of sequences.

We will introduce additional sequence types, such as lists and tuples, in a later chapter.

The next example demonstrates the use of the following sequence operations with strings:

- The concatenation (+) operator.
- The repetition (*) operator.
- The membership testing (`in`) operator.

string_operations.py

```
#!/usr/bin/env python3
# The concatenation operator (+)
first_name = "Casey"
last_name = "Jackson"
full_name = first_name + " " + last_name
print(full_name)  # Casey Jackson

# Note the automatic string concatenation below
fullName = "Casey" " " "Jackson"
print(fullName)

# The asterisk (*) operator
stars = "*" * 12
pounds = 5 * "#"
```

```
print(stars, ":", pounds)  # ********** : #####

# The in operator is convenient for membership tests
x = "Hello there"
print('t' in x, 'ell' in x, 'hell' in x)  # True True False
```

Note that when you place literal strings next to each other, as seen on line 9, they are automatically concatenated even though there is no + operator between them.

Indexing and Slicing

Strings, like other sequence types, have a set of operations that utilize square brackets (`[]`) in the syntax.

All three of the following types of operations return a substring of the sequence the operation is performed on.

Indexing

Use one integer inside the brackets to get the element in that position, starting from zero. Negative values start the positional counting from the end of the sequence.

Slicing

Use two integers separated by a colon in the brackets to get the elements between the given positions.

Negative values also start from the end of the sequence, and omitted values mean `From the beggining` or `til the end` depending if you forgo the first or the second value.

Extended Slicing

Extended slicing behaves like regular slicing, but adds a third integer separated by a colon, which indicates the iteration step size to use when picking elements from the sequence. By default, the value of this parameter is one.

For example, `my_list[3:10:2]` starts from index 3 and stops at index `10`, selecting every two elements, because the step is 2.

The following example demonstrates these three operations:

indexes_and_slices.py

```
#!/usr/bin/env python3
spam = "Spam and eggs"
delim = " | "
# Indexing
print(spam[0], spam[3], spam[-1], spam[-4], sep=delim)

# Slicing
print(spam[2:7], spam[5:], spam[:8], sep=delim)

# Slicing from end
print(spam[-3:-1], spam[-3:], spam[:-1], sep=delim)
print()
```

```
# Extended Slicing
alphabet = "abcdefghijklmnopqrstuvwxyz"
print(alphabet[2:18:3])
start = 18
print(alphabet[start::1])
print(alphabet[::1])
```

Running the preceding program gives the following output:

```
$ python3 indexes_and_slices.py
S | m | s | e
am an | and eggs | Spam and
gg | ggs | Spam and egg

cfilor
stuvwxyz
abcdefghijklmnopqrstuvwxyz
$
```

Formatting Strings

Strings have a built in % operator that you can use to format stings similar to the `sprintf()` function in the C language.

- Using the % operator with strings may lead to a number of common errors.

References

You can find an in depth description of the % operator at the following URL:
`https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting`

- The newer `format()` method is the recommended way to avoid these errors and provide a more powerful way to format text.

The `format()` is a method of the `str` class that returns a copy of the string with each placeholder replaced with the value of the corresponding argument.

You can specify placeholders within the string by using the `{}` characters.

- If the placeholder characters surround an integer number, then it replaces the value by position.
- If the placeholder characters surround a word, then the substitution value is passed by keyword argument.

References

You can find a complete description of the various formatting options at the following URL:
`https://docs.python.org/3/library/string.html#formatstrings`

The following program demonstrates a few simple examples of using the `format()` method.

- We will introduce more complex examples later in the course.

format.py

```
#!/usr/bin/env python3
separator = "----------------------------"
name = "First: {} \tLast Name: {} \tMiddle Initial: {}"
formatted = name.format("John", "Smith", "C.")
print(formatted)
formatted = name.format("Melony", "Jones", "A.")
print(formatted)
print(separator)

name = "{1},  {0}"
print(name.format("First", "Last"))
print(name.format("John", "Smith"))
print(name.format("Melony", "Jones"))
print(separator)

dimensions = "Type: {type}\nHeight:{height}, Width:{width}"
result = dimensions.format(height=50, width=25, type="Box")
print(result)
```

The preceding program outputs:

```
$ python3 format.py
First: John   Last Name: Smith    Middle Initial: C.
First: Melony       Last Name: Jones     Middle Initial: A.
----------------------------
Last, First
Smith, John
Jones, Melony
----------------------------
Type: Box
Height:50, Width:25
$
```

Conversion Functions

When you need to treat a string as a numeric type, or a number as a string, you must convert the values.

Python provides several functions that perform these types of conversions.

The following table contains some of the most common built-in conversion functions:

`int()`	`str()`	`chr()`	`oct()`
`float()`	`ord()`	`hex()`	`bin()`

The `int()`, `float()`, and `str()` are constructors that initialize an object, as opposed to a function call.

We will discuss object initialization in more detail later in the course.

conversions.py

```
#!/usr/bin/env python3
# conversions to an integer
result = input("Please enter an integer: ")
number = int(result)
print("Your number plus 10 equals:", number + 10)
print("String of", result, "converted to various bases as int:")
fmt = "Base 10: {}\tBase 2: {}\tBase 8: {}\tBase 16: {}"
print(fmt.format(int(result), int(result, 2), int(result, 8), int(result, 16)))
print()
# conversions to a float
a_float = input("Please enter a decimal number: ")
sum_of_input = float(a_float) + float(result)
print(a_float, "+", result, "=", sum_of_input)
print()
print(number, "as string in the following bases:")
fmt = "Binary: {}\tOctal: {}\tHex: {}"
print(fmt.format(bin(number), oct(number), hex(number)))

print('ord("A") =', ord("A"), '     chr(66) =', chr(66))

print("The sum of the input equals", sum_of_input)
```

The preceding program outputs this:

```
$ python3 conversions.py
Please enter an integer: 101
Your number plus 10 equals: 111
String of 101 converted to various bases as int:
Base 10: 101  Base 2: 5
Base 8: 65    Base 16: 257
Please enter a decimal number: 23.45
23.45 + 101 = 124.45
101 as string in the following bases:
Binary: 0b1100101
 Octal: 0o145
   Hex: 0x65
ord("A") = 65      chr(66) = B
The sum of the input equals 124.45
$
```

If the arguments passed to the conversion functions do not match the expected format then Python raises an exception.

The following examples show the most common conversion errors:

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> result = int(input("Please enter a number: "))
Please enter a number: Hello
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hello'
>>> result = int("123", 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 2: '123'
>>> print(float("987"))     # While this works
987.0
>>> print(int(98.76))       # And this works also
98
>>> print(int("98.76"))     # This one does not work
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '98.76'
>>> exit()
$
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `basics/solutions` directory.

Exercise 1

Write a program that prompts to enter a string of text.

- The program should print the original text followed on a second line in the output by the number of characters entered.

Exercise 2

Write a program that prompts twice for text from the user.

- The first input should be a first name.
- The second input should be a last name.
- The program should print the full name on one line and the person's initials on the second line.

Exercise 3

Write a program that accepts a string from the user.

- Determine and print the following information about the string:
  - Does it end in a period?
  - Does it contain all alphabetic characters?
  - Is there an 'x' in the string?
- Create and print a new string that has all occurrences of `e` changed to `E`.

Exercise 4

Write a program that asks the user to enter a sentence.

- The program should determine and print the following information:
  - The first character in the string of text and the number of times it occurs in the string.
  - The last character in the string of text and the number of times it occurs in the string.

Exercise 5

Write an application that prompts to enter the radius of a circle.

- Accept the user input into a variable.
- Compute and print the area of the circle whose radius was input.
  - The formula for the area of a circle is πr^2 (pi times the square of the radius).
  - Use `3.14159` for pi.

Exercise 6

Write a program that prompts twice for an integer.

- Print the product of the two numbers.
- Once this works properly, try entering numbers with a decimal point.
  - What happens? Why?
- Now try entering data that is nonnumerical.
  - What happens? Why?

Exercise 7

Write a program that prompts the user for a string and then prompts again for a number.

- The program should create and print a new string by using the repetition operator on the string and the number.
  - For example, if the string is `hello` and the number is 3, then `hellohellohello` should be printed.

Exercise 8

Write a program that prompts the user twice for a number.

- The first number will be the base, and the second number will be the exponent.
  - Print the result of raising the base to the exponent.

References

- You can find more information about numeric data types in section 4.4 of the following URL:
  `https://docs.python.org/3/library/stdtypes.html`

- You can find a complete list of string methods and their definitions in the Python documentation at the following URL:
  `https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str`

- You can find an in depth description of the % operator at the following URL:
  `https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting`

- You can find a complete description of the various formatting options at the following URL:
  `https://docs.python.org/3/library/string.html#formatstrings`

Summary

- Python is a high-level, interpreted, dynamically-typed programming language that relies on indentation to determine the control flow structure of your applications.
- The `input()` function obtains user input at runtime from the keyboard, and the `print()` function displays data to the user.
- Comments make your code more understandable and maintainable.
- Python supports the numeric and string data types that you can use, operate, and transform to fulfill the requirements of your application.

Chapter 2

Language Components

Goal

Use fundamental decision-making structures, logical operators, and iteration statements.

Objectives

- Use the indenting requirements of Python properly.
- Use the Python control flow constructs correctly.
- Understand and use Python's various relational and logical operators.
- Use `if` statements to make decisions within the Python code.
- Use `while` and `for` loops to perform repetitive operations.

Sections

- Language Components

Language Components

Objectives

- Use the indenting requirements of Python properly.
- Use the Python control flow constructs correctly.
- Understand and use Python's various relational and logical operators.
- Use `if` statements to make decisions within the Python code.
- Use `while` and `for` loops to perform repetitive operations.

Indenting Requirements

Python provides a robust set of keywords and related items that control the flow of execution within an application.

- In this section, we will explore the various conditional execution and looping options that Python provides.
- In addition, we will look at the various operators used by Python in control flow constructs.

Python mandates the use of indenting within a compound statement.

- The first line of the compound statement is referred to as the header.
- All other statements within the compound statement are referred to as the suite or body and must be indented the same number of columns to be part of the same suite.
- The suite ends with the first statement that is unindented to the column of the header.

One such type of compound statement is a control structure.

Here is the general syntax for an `if` statement.

```
if some_condition:
    suite_statement_1
    suite_statement_2
    suite_statement_3  # suite ends here

print("some output")
```

- Suites must be indented the same amount of white space from the starting column of the header.
- When using tabs in the source code, a single tab is not equal to the number of spaces used in a tab.
- The recommendation is to use spaces over tabs.

- Typically, many editors and IDEs will automatically indent for you.

The if Statement

The fundamental decision making control structure in many programming languages is the `if` statement.

- The following examples demonstrate proper indenting when using the `if` statement and its variants.
- Also, notice the required use of the colon (`:`) to end the header portion of the `if` and the `else`.

if_else.py

```
#!/usr/bin/env python3
number = int(input("Enter a number between 1 and 100: "))
target = 50
if number < target:
    print(number, "is less than", target)
else:
    print(number, "is greater than or equal to ", target)
```

In the above example, `number` and `target` are being printed in both the `if` and the `else` statement.

- It is as if they are being printed unconditionally. Such variables might be better printed outside of the `if else` statement as shown in the rewrite example below.
- It also helps make the code **DRY**, an acronym for **D**on't **R**epeat **Y**ourself.

if_else_rewritten.py

```
#!/usr/bin/env python3
number = int(input("Enter a number between 1 and 100: "))
target = 50
if number < target:
    result = "is less than"
else:
    result = "is greater than or equal to "

print(number, result, target)
```

When writing an `if` statement, there can be zero or more `elif` blocks, and the `else` block is optional.

- The keyword `elif` can be used to prevent the testing of multiple `if` statements when only one of the `if` statements can ever be `True` at a time.
- It can also sometimes prevent the need for nesting `if` statements inside of an `else`.
- Because Python does not have a `switch` statement found in other languages, the `elif` is often a suitable substitute.

elif_example.py

```
#!/usr/bin/env python3
value = int(input("Please enter a whole number: "))

print(value, end=" is ")
if value <= 5:
    print("less than or equal to 5")
elif value <= 10:
    print("between 6 and 10 inclusively")
elif value <= 15:
    print("between 11 and 15 inclusively")
else:
    print("greater than 15")

print("This statement is not part of the above if else")
```

Relational and Logical Operators

Many decisions in a programming language depend upon how one value relates to another.

The Relational Operators in Python

Relational Operators

Operator	Meaning
<	Strictly less than
<=	Less than or equal
>	Strictly greater than
>=	Greater than or equal
==	Equal
!=	Not equal
is	Object identity
is not	Negated object identity

The Logical Operators in Python

Logical Operators

Operator	Unary or Binary	The Result is `True` When ...
not	unary	Operand is `False`

Operator	Unary or Binary	The Result is `True` When ...
`and`	binary	Both operands must be `True`
`or`	binary	Only one operand must be `True`

- The logical operators allow expressions consisting of compound conditions such as the ones shown next.

logicals.py

```
#!/usr/bin/env python3
x = y = 0
if x == 0 and y == 0:
    print("x and y are zero")

if x == 0 or y == 0:
    print("x or y or both are zero")

if not (x >= 10 and x <= 20):
    print("x not between 10 and 20")
```

- Both the `and` and the `or` are *short-circuited* operators.
  - This means that when the first part of an `or` evaluates to `True`, the second part of the `or` is not evaluated.
  - Likewise, when the first part of an `and` evaluates to `False`, the second part of the `and` is not evaluated.
- The relational and logical operators yield results of either `True` or `False`.
- You can write Python statements that check for these values.

```
x = 10
y = 20
z = x < y
if z:
    print("Result is", z)
```

- In addition to the literal values `True` and `False`, there are other expressions that evaluate to `True` or `False` when used where a conditional statement is expected.
  - `True`
    - Any non-zero value
  - `False`
    - `0`
    - Empty sets, dictionaries, tuples, or lists
    - Empty strings

- `None` – A built-in constant frequently used to represent the absence of a value

The while Loop

The `while` statement causes Python to loop through a suite of statements if the test expression evaluates to `True`.

The `while` statement uses the same evaluations for `True` and `False` as the `if` statement.

- Likewise, the same indentation rules are used for a `while` as they are for an `if`.

The following example demonstrates the use of a while loop to add the integers from 5 to 10.

calculate_sum.py

```
#!/usr/bin/env python3

counter = 5
total = 0

while counter <= 10:
    total += counter
    counter += 1
    print("Running Total =", total, "Counter =", counter)

print()
print("Final Total =", total)
```

- The output of the above program is shown below.

```
$ python3 calculate_sum.py
Running Total = 5 Counter = 6
Running Total = 11 Counter = 7
Running Total = 18 Counter = 8
Running Total = 26 Counter = 9
Running Total = 35 Counter = 10
Running Total = 45 Counter = 11
Final Total = 45
$
```

- Each time through the loop, if the condition in the `while` is `True`, then the body the `while` loop is executed.
- When the condition is `False`, then the loop is complete and first statement after the `while` loop is executed.

The break and continue Statements

Any looping construct can have its control flow changed through a `break` or `continue` statement within it.

When a `break` statement is executed, control of the program jumps to the first statement beyond the loop.

A `break` statement is often used when searching through a collection for the occurrence of a particular item.

When a `continue` statement is executed, the rest of the suite is skipped for that iteration of the loop and control goes to the next iteration.

The next example incorporates both `break` and `continue` statements.

continue_and_break.py

```
#!/usr/bin/env python3
cnt = 0
total = 0
while cnt <= 100:
    cnt += 1
    if cnt % 4 == 0:
        continue         # skip even multiples of 4
    if cnt * cnt > 400:
        break            # will happen at cnt = 21
    total += cnt

print("Total is:", total, "    Count is:", cnt)
```

- The output of the above program is shown next.

```
$ python3 continue_and_break.py
Total is: 150 Count is: 21
$
```

The for Loop

The `for` loop in Python is used to iterate over the items of any sequence, such as a *list* or a *string*.

A `range` object can also be used to represent a sequence of numbers and then iterate over the `range` as a sequence with a `for` loop.

The example below demonstrates using `for` loops to loop through a string and several sequences via a `range` object.

for_loops.py

```
#!/usr/bin/env python3
word = "Hello"
print(word)
for each_character in "Hello":
    print(each_character, end="\t")

delim = "\n\t"
print("\nrange(5):", end=delim)
for i in range(5):
    print(i, end=" ")

print("\nrange(5, 10)", end=delim)
```

```
for i in range(5, 10):
    print(i, end=" ")

print("\nrange(-5, 9, 3)", end=delim)
for i in range(-5, 9, 3):
    print(i, end=" ")

print()
```

Notice the third optional `step` parameter of the `range` function. This parameter indicates the increment between each value of the generated sequence.

The output of the above program is shown below.

```
$ python3 for_loops.py
Hello
H    e    l    l    o
range(5):
       0 1 2 3 4
range(5, 10)
       5 6 7 8 9
range(-5, 9, 3)
      -5 -2 1 4 7
$
```

A *list* is another type of sequence that a `for` loop is often used to loop through.

- While a formal discussion of lists does not come until the next chapter, recall that the `split` method of a string returns a *list* of strings.
  - The `split` method accepts an optional argument indicating the delimiter to use when splitting the string.
  - If no argument is passed, white space is used as the delimiter.

splitting_strings.py

```
#!/usr/bin/env python3
text = """Each   word   is   separated
          by     whitespace"""
data = text.split()
for value in data:
    print(value)

print("*" * 50)

text = "This,is,comma,separated,text"
data = text.split(",")
for value in data:
    print(value)
```

The output of the above program is shown next.

```
$ python3 splitting_strings.py
Each
word
is
separated
by
whitespace
**************************************************
This
is
comma
separated
text
$
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `language/solutions` directory.

Exercise 1

Write a program that prompts for a lucky number. The program should print out a message if the number entered is not an integer.

Exercise 2

Rewrite the preceding exercise to additionally print out how many digits are in the number, if the number is an integer.

Exercise 3

Write a program that prompts twice for an integer.

- The program should print the larger of the two numbers.
- If the numbers are equal, then the program should indicate it as such.

Exercise 4

Write a program that prompts twice for an integer.

- The program should output the sum of the integers within the range of those two numbers inclusively.
- For example, if the user inputs the numbers *10* and *15*, then the sum would be *75*.

```
10 + 11 + 12 + 13 + 14 + 15 = 75
```

Exercise 5

Ask the user to input multiple numbers on one input line.

- Split the numbers into a *list*.

- Write a loop that examines each element of the list and displays the ones that are greater than zero.

Exercise 6

Ask the user to input three numbers representing a lower limit, a higher limit, and a step value.

- The program should use a *range* object to loop through and print the numbers from low to high (inclusive), taking into consideration the step.

Exercise 7

Use a *range* to loop through and print each number from 0 to 49 to produce the following output.

- Each number should be printed individually as opposed to concatenating them as a string.

```
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
```

Exercise 8

Rewrite exercise # 4 such that the program takes into account the case where the first number entered is bigger than the last.

- For example, if the user inputs the numbers *10* and *15*, then the sum would be *75*.

```
10 + 11 + 12 + 13 + 14 + 15 = 75
```

- If the user inputs the numbers *15* and *10*, then the sum would be still be *75*.

Summary

- You can use decision-making structures to execute specific blocks of code.
- You can iterate through a suite of statements by using iteration statements.

Chapter 3

Collections

Goal

Use standard and efficient data access types to store data.

Objectives

- Write Python programs by using the various Python collection types.
- Use lists to store and manipulate ordered collections of objects.
- Use tuples to store and retrieve immutable collections of objects.
- Use sets to store and manipulate unordered collections of unique objects.
- Use dictionaries to perform quick lookups on collections of objects.
- Sort the various collection data types in a variety of ways.

Sections

- Collections

Collections

Objectives

- Write Python programs by using the various Python collection types.
- Use lists to store and manipulate ordered collections of objects.
- Use tuples to store and retrieve immutable collections of objects.
- Use sets to store and manipulate unordered collections of unique objects.
- Use dictionaries to perform quick lookups on collections of objects.
- Sort the various collection data types in a variety of ways.

Introduction

Python provides the following general purpose built-in classes that represent standard collection data types.

- `list`
  - An ordered collection of elements
  - Similar to an array in other languages
  - Dynamic. A list can grow or shrink based on the needs of the application
- `tuple`
  - An immutable `list`
- `set`
  - An unordered collection of elements
  - Does not permit duplicates
- `dict`
  - A dictionary is a collection of key-value pairs
  - The keys of a dictionary have to be unique
  - As of Python 3.7, dictionary values are stored in insertion order.

The `list` and `tuple` data structures are also called sequences, because they support efficient element access using integer indices (slice notation).

Each of the preceding collections has a constructor that can be used to create an empty instance of the data type or as a conversion method from one collection to another.

- `list()`
- `tuple()`

- `set()`
- `dict()`

Lists

A `list` is an ordered sequence with zero or more objects.

You can create lists in several ways.

- By using the `list()` constructor.
- By using a pair of square brackets to denote the empty list (`[]`).
- By using square brackets, separating items with commas.

The elements in a list can be a mix of any types.

Once created, the slice notation can be used to reference element(s) of a `list`.

The next example demonstrates creating and accessing several lists.

creating_lists.py

```
#!/usr/bin/env python3
listA = list()
listB = []
listC = [20, 3, 7, 82, -3, 456, 3, 65, 23]
listD = ["James", "Heather", "Monica", "Eugene"]
listE = listC + listD
print(listA, listB, listC, listD, listE, sep="\n")
print("Indexing:", listC[0], listC[-1], listD[2], listE[4])
sub_list = listC[0:5]
print(type(sub_list), sub_list)
print(listE[-5:])
```

The output of the above program is shown next.

```
$ python3 creating_lists.py
[]
[]
[20, 3, 7, 82, -3, 456, 3, 65, 23]
['James', 'Heather', 'Monica', 'Eugene']
[20, 3, 7, 82, -3, 456, 3, 65, 23, 'James', 'Heather', 'Monica', 'Eugene']
Indexing: 20 23 Monica -3
<class 'list'> [20, 3, 7, 82, -3]
[23, 'James', 'Heather', 'Monica', 'Eugene']
$
```

A `list` is dynamic and as such can grow and/or shrink as needed.

- Adding, deleting, and updating elements are common operations performed on a `list`.
- The next example demonstrates various methods that you can use on a list to modify its contents.

working_with_lists.py

```
#!/usr/bin/env python3
numbers = [10, 20, 30, 40, 20, 50]
fmt = "{0:>24} {1}"
print(fmt.format("Original:", numbers))
print(fmt.format("Pop Last Element:", numbers.pop()))
print(fmt.format("Pop Element at pos# 2:", numbers.pop(2)))
print(fmt.format("Resulting List:", numbers))

numbers.append(100)
print(fmt.format("Appended 100:", numbers))
numbers.remove(20)
print(fmt.format("Removed First 20:", numbers))
numbers.insert(1, 1000)
print(fmt.format("Inserted 1000 at pos# 1:", numbers))

numbers.reverse()
print(fmt.format("Reversed:", numbers))
numbers.sort()
print(fmt.format("Sorted:", numbers))
numbers[0] = -99
print(fmt.format("Modified:", numbers))
```

The output of the above program is shown next.

```
$ python3 working_with_lists.py
               Original: [10, 20, 30, 40, 20, 50]
       Pop Last Element: 50
  Pop Element at pos# 2: 30
         Resulting List: [10, 20, 40, 20]
           Appended 100: [10, 20, 40, 20, 100]
       Removed First 20: [10, 40, 20, 100]
Inserted 1000 at pos# 1: [10, 1000, 40, 20, 100]
               Reversed: [100, 20, 40, 1000, 10]
                 Sorted: [10, 20, 40, 100, 1000]
               Modified: [-99, 20, 40, 100, 1000]
$
```

The elements of a list can be unpacked into individual variables. Referencing specific elements as variables is often easier to understand than referencing list elements by index.

unpacking_lists.py

```
#!/usr/bin/env python3
days = ["Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday"]
mon, tue, wed, thu, fri = days
print(mon, fri)
```

```
$ python3 unpacking_lists.py
Monday Friday
$
```

Looping through a `list` is typically accomplished with a `for` loop.

list_loops.py

```
#!/usr/bin/env python3
numbers = [10, 20, 30, 40, 50]

# Looping by element
for number in numbers:
    print(number, end="\t")
print()

# Looping by index and
# updating list's values at the same time
for index in range(len(numbers)):
    numbers[index] *= 10

for number in numbers:
    print(number, end="\t")
print()
```

The output of the preceding program is shown next.

```
$ python3 list_loops.py
10   20   30   40   50
100  200  300  400  500
$
```

Tuples

A `tuple` is an ordered immutable collection.

- A `tuple` is somewhat like a `list`, but it is typically more efficient, because it cannot shrink, grow, or be changed in any other way.
- All of the immutable operations that can be performed on a `list` can also be performed on a `tuple`.

You can create tuples in several ways.

- By using the `tuple()` constructor.
- By using a pair of parenthesis to denote the empty `tuple` (`()`).
- By using parenthesis, separating items with commas.

The `enumerate()` constructor can be useful when looping through a collection and needing the index of the element being processed at the same time.

- Each time through the loop, the `enumerate` object returns a tuple containing a count (from start, which defaults to 0) and the values obtained from iterating over the iterable object passed to it.

- This is often simpler than obtaining the length of the iterable object with the `len()` function and passing it to the `range()` constructor to loop through by index as in the previous example.

The following demonstrates `tuple` objects and using `enumerate()`.

tuple_loops.py

```
#!/usr/bin/env python3
grades = ("A", "B", "C", "D", "F")
points = ("90-100", "80-89", "70-79", "60-69", "00-59")
for grade in grades:
    print(grade, end="\t")
print()

for a_tuple in enumerate(grades):
    print(a_tuple[0], ":", a_tuple[1], end="\t")
print()

# Unpacking the tuple from enumerate
for i, grade in enumerate(grades, start=1):
    print(i, ":", grade, end="\t")
print()

# Process the two tuples in parallel
for index, grade in enumerate(grades):
    print(grade, ":", points[index])
```

The output of the preceding program is as follows:

```
$ python3 tuple_loops.py
A       B       C       D       F
0 : A   1 : B   2 : C   3 : D   4 : F
1 : A   2 : B   3 : C   4 : D   5 : F
A : 90-100
B : 80-89
C : 70-79
D : 60-69
F : 00-59
```

Sets

A `set` is an unordered collection with no duplicates.

- A `set` can be created by using the `set()` constructor.

- A `set` can also be initialized using curly braces `{}`, separating items with commas.

- Any duplicate entries will be removed when the `set` is created.

If you pass a string to the `set()` constructor, then each character of the string will become an element of the `set`, with duplicates removed.

The following example demonstrates various ways of creating a set.

creating_sets.py

```
#!/usr/bin/env python3
setA = set()
print(len(setA), ":", setA)
setB = set("mississippi")
print(len(setB), ":", setB)
setC = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"}
print(len(setC), ":", setC)
```

The output of the preceding example is shown next.

```
$ python3 creating_sets.py
0 : set()
4 : {'i', 'p', 's', 'm'}
7 : {'Sun', 'Thu', 'Mon', 'Wed', 'Fri', 'Sat', 'Tue'}
$
```

The `in` operator is often used for membership testing in a `set`.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> fruit = {"apple", "orange", "pear", "kiwi"}
>>> print("orange" in fruit, ":", "crabgrass" in fruit)
True : False
>>> exit()
$
```

The following methods can also be called on a `set`.

- The `add()` method adds an element to the `set`.
- The `update()` method adds the elements of any iterable object passed to it as a parameter.
- The `remove()` method removes an element from the `set`.
  - Raises a `KeyError` exception if element to remove is not present.
- The `discard()` method also removes an element from the `set`.
  - This method does not raise any exception if an attempt is made to discard an element that does not exist.
- The `pop()` method removes and returns an arbitrary element.
  - Raises a `KeyError` exception if the set is empty.
- The `clear()` method removes all the elements from a `set`.

- The `issuperset()` and `issubset()` methods can be used to test membership between two different sets.

The following mathematical set operators and equivalent methods are available, as well, when working with sets.

Mathematical Set Operators and Methods

Operator	Method	Description
¦ (vertical bar)	`union()`	Returns a new `set` with elements from the `set` and all others.
& (ampersand)	`intersection()`	Returns a new `set` with elements common to the `set` and all others.
- (dash)	`difference()`	Returns a new `set` with elements in the `set` that are not in the others.
^ (carrot)	`symmetric_difference()`	Returns a new `set` with elements in either the `set` or other but not both.

In the preceding table, following each of the operators with an = will update the `set` rather than return a new `set`.

The following example demonstrates working with sets.

working_with_sets.py

```
#!/usr/bin/env python3
# Define the strings that are used multiple times
prompts = ["Please enter some first names ", "again ",
           "to delete "]
end = "\n" * 2  # define value to use as end in prints

# Using str.join to efficiently join together strings by
# avoiding string concatentation
msg1 = prompts[0]
msg2 = "".join(prompts[:2])
msg3 = "".join(prompts)

name_list = input(msg1).split()
unique_names = set(name_list)
backedup_names = set(unique_names)
print(unique_names, end=end)

# Check to see if name exists prior to adding
name_list = input(msg2).split()
for name in name_list:
    if name not in unique_names:
```

```
        unique_names.add(name)
    else:
        print("\t", name, "already exists and is ignored")
print(unique_names, end=end)

# Update contents of set with contents of a list
name_list = input(msg2).split()
unique_names.update(name_list)
print(unique_names, end=end)

# Check to see if name exists prior to removing
name_list = input(msg3).split()
for name in name_list:
    if name in unique_names:
        unique_names.remove(name)

# Discard words without checking first
name_list = input(msg3).split()
for name in name_list:
    unique_names.discard(name)
print(unique_names, end=end)

# Check the relationship of one set to another
print()
print("Original:", backedup_names)
print("Current :", unique_names, "\n")
print("Original is subset of Current ? ",
      backedup_names.issubset(unique_names))
print("Current is superset of Original ? ",
      unique_names.issuperset(backedup_names))

# Pop each element from the set until it is empty
print("Popping each name from set: ", unique_names)
while unique_names:
    print(unique_names.pop(), end=" ")
print()
```

The output of running the above program is shown below.

```
$ python3 working_with_sets.py
Please enter some first names Emma Declan
{'Emma', 'Declan'}

Please enter some first names again Emma Ava Mia Gavin Declan
        Emma already exists and is ignored
        Declan already exists and is ignored
{'Gavin', 'Ava', 'Declan', 'Mia', 'Emma'}

Please enter some first names again Ava Declan Jasmine Cole
{'Cole', 'Gavin', 'Ava', 'Declan', 'Mia', 'Emma', 'Jasmine'}

Please enter some first names again to delete Ava Mason Jasmine Sally
Please enter some first names again to delete Gavin Sam Joey
{'Cole', 'Declan', 'Mia', 'Emma'}
```

```
Original: {'Emma', 'Declan'}
Current : {'Cole', 'Declan', 'Mia', 'Emma'}

Original is subset of Current ? True
Current is superset of Original ? True
Popping each name from set: {'Cole', 'Declan', 'Mia', 'Emma'}
Cole Declan Mia Emma
$
```

The following example demonstrates the mathematical operators and methods of sets.

set_operations.py

```
#!/usr/bin/env python3
a, b = [set('efgy'), set('exyz')]
fmt = "Set a:{}\t\tSet b:{}"
tab = "\t"
print("The following operators return a new set")
print("Leaving the original two sets unchanged")
print(fmt.format(a, b))
print(tab, "a | b:", a | b)  # union
print(tab, "a & b:", a & b)  # intersection
print(tab, "a - b:", a - b)  # difference
print(tab, "b - a:", b - a)  # difference
print(tab, "a ^ b:", a ^ b)  # symmetric difference
print(fmt.format(a, b))
print("\n", "*" * 75)

print("\nThe '|=' operator modifies the original set")
a, b = [set('efgy'), set('exyz')]
a |= b
print(tab, "a |= b:", a)  # union
print(fmt.format(a, b))
```

The output of the above program is shown next.

```
$ python3 set_operations.py
The following operators return a new set
Leaving the original two sets unchanged
Set a:{'e', 'y', 'g', 'f'}        Set b:{'e', 'y', 'z', 'x'}
         a | b: {'g', 'f', 'y', 'z', 'e', 'x'}
         a & b: {'e', 'y'}
         a - b: {'f', 'g'}
         b - a: {'z', 'x'}
         a ^ b: {'z', 'g', 'f', 'x'}
Set a:{'e', 'y', 'g', 'f'}        Set b:{'e', 'y', 'z', 'x'}

***********************************************************

The '|=' operator modifies the original set
```

```
       a |= b: {'g', 'f', 'y', 'z', 'e', 'x'}
Set a:{'g', 'f', 'y', 'z', 'e', 'x'}         Set b:{'e', 'y', 'z', 'x'}
$
```

Dictionaries

A `dict`, or dictionary, is an unordered collection of entries.

- Each entry contains a key-value pair.
- Other languages refer to this data type as a hash, map, or associative array.

Dictionary keys have to be both unique and hashable.

- All of Python's immutable built-in objects are hashable and can be used as keys in a dictionary.

Dictionaries can be created in several ways.

- By placing a comma-separated list of key-value pairs within braces.

```
ages = {'Jack': 51, 'Casey': 43, 'Derek': 27}
positions = {1: "First", 2: "Second", 3: "Third"}
empty = {}
```

- By calling the `dict()` constructor.

```
empty_also = dict()
something = dict(key1='value 1', key2='value 2')
fancy = dict([[8, 2], [5, 4], [3, 9]])
```

Adding a key-value pair is done through assignment.

adding_to_dictionary.py

```
#!/usr/bin/env python3
fruits = {"a": "Apples", "b": "Bananas"}
print(fruits)
fruits["s"] = "Strawberries"    # Add a new key-value pair
print(fruits)
```

The output of the above program is shown next.

```
$ python3 adding_to_dictionary.py
{'a': 'Apples', 'b': 'Bananas'}
{'a': 'Apples', 'b': 'Bananas', 's': 'Strawberries'}
$
```

Dictionaries are designed as a data structure that provides fast lookups into the structure by key.

- You can fetch a value from a dictionary by its key, as shown in the following example.
- Note that attempting to supply a key that does not exist in the dictionary will result in a `KeyError` exception being raised.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> products = {'p1': 'Product 1', 'p2': 'Product 2', "p3": "Product 3"}
>>> value = products["p1"]
>>> print(value)
Product 1
>>> value = products["p4"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'p4'
>>> exit()
$
```

- Dictionaries provide the `get()` method, as an alternative to using the `[]` syntax to retrieve a value by key.
- In contrast to the `[]` syntax, the `get()` method does not generate a `KeyError` if the key passed as a parameter does not exist.
- Instead, the `get()` method returns `None` by default if a key is not present.
- If you pass a second argument to the `get()` method, then the method uses that argument as the return value if the key is not present.

The following example demonstrates the use of the `get()` method with dictionaries.

getting_from_dictionary.py

```
#!/usr/bin/env python3
reward_pts = {"Bryce": 500, "Heather": 2000, "Kaylie": 750}
points = reward_pts.get("Bryce")      # returns 500
print("Bryce:", points)
points = reward_pts.get("Stephanie")  # returns None
print("Stephanie:", points)

# Supplying a different value to return other than None
points = reward_pts.get("Stephanie", 0)  # returns 0
print("Stephanie:", points)
print("Current Dictionary Contents:", reward_pts)
```

```
$ python3 getting_from_dictionary.py
Bryce: 500
Stephanie: None
Stephanie: 0
Current Dictionary Contents: {'Bryce': 500, 'Heather': 2000, 'Kaylie': 750}
$
```

To remove elements from a dictionary, you can use the `pop()` and `popitem()` methods.

The general syntax for the `pop()` method is as follows.

```
value = obj.pop(key[,default])
```

- `obj` represents the dictionary from which to attempt to remove `key` and its associated value.
- `key` represents the key of the key-value pair to attempt to remove.
- `default` represents an optional value to return if `key` does not exist. A `KeyError` exception is raised if no default is given and the `key` does not exist.

The general syntax for the `popitem()` method is as follows.

```
a_tuple = obj.popitem()
```

- `obj` represents the dictionary from which to remove something.
- The `popitem()` method removes and returns an arbitrary key-value pair as a two-element `tuple`, but raises a `KeyError` if `obj` is empty.

The following program demonstrates removing elements from a dictionary.

removing_from_dictionary.py

```
#!/usr/bin/env python3
unknown = "Make is Unknown"
cars = {'Mustang': 'Ford', 'Falcon': 'Ford',
        'Camaro': 'Chevy', 'Corvette': 'Chevy',
        'Eclipse': 'Mitsubishi', 'Integra': 'Acura'}

make = cars.pop("Corvette")
print(make)

make = cars.pop("Accord", unknown)
print(make)

a_tuple = cars.popitem()
print(a_tuple[0], a_tuple[1])

model, make = cars.popitem()
print(model, make)
print(cars)
```

The output of the above program is shown next.

```
$ python3 removing_from_dictionary.py
Chevy
Make is Unknown
Mustang Ford
Camaro Chevy
{'Falcon': 'Ford', 'Integra': 'Acura', 'Eclipse': 'Mitsubishi'}
$
```

When there is a need to loop through a dictionary, the following methods are useful.

- The `keys()` method returns a view of the dictionary keys.
- The `values()` method returns a view of the dictionary values.
- The `items()` method returns a view of the dictionary items.
- Each item in the view returned by `items()` is a two-element `tuple` consisting of a key and a value.

The objects returned by the preceding methods are called view objects.

- A view object provides a dynamic view of the dictionary's entries, which means that when the dictionary changes, the view reflects these changes.

The views returned by the preceding methods are iterable, enabling them to be used in a for loop to iterate through all of its members.

- Each of the views can also be converted to a list or a tuple by passing the view to a `list()` or `tuple()` constructor.
- The views returned by `keys()` and `items()` are set-like views that can be converted to sets with the `set()` constructor.

In addition to the preceding methods, which return views, a dictionary itself is iterable. When iterating over a dictionary, you iterate over the dictionary keys.

You can also pass a dictionary to the `len()` function to determine how many key-value pairs exist in the dictionary.

The example on the following page shows the various techniques of looping through a dictionary.

processing_dictionaries.py

```
#!/usr/bin/env python3
reward_pts = {"Bryce": 500, "Heather": 2000, "Kaylie": 750,
              "Amanda": 1350, "Casey": 2400, "Jason": 800,
              "Kaylie": 25}
rule, prefix = "-" * 75, "\n"

print("Looping through dictionary", rule, sep=prefix)
for name in reward_pts:
    print(name, reward_pts[name])

print(prefix, "Looping through keys()", rule, sep=prefix)
for name in reward_pts.keys():
    print(name, end=" ")

print(prefix, "Looping through values()", rule, sep=prefix)
for value in reward_pts.values():
    print(value, end=" ")

print(prefix, "Looping through items() as tuples", rule,
      sep=prefix)
for item in reward_pts.items():
    print(item)

fmt = "{:8}~{:>8}"
```

```
print(prefix, "Items() unpacked as keys and values", rule,
      sep=prefix)
print(fmt.format("Name", "Points"))
for name, points in reward_pts.items():
    print(fmt.format(name, points))

print(prefix, "Data types of returned views", rule,
      sep=prefix)
print("  keys()", type(reward_pts.keys()))
print("values()", type(reward_pts.values()))
print(" items()", type(reward_pts.items()))
```

The output of the above program is shown on the following page.

```
$ python3 processing_dictionaries.py
Looping through dictionary
--------------------------------------------------------------------------------
Bryce 500
Heather 2000
Kaylie 25
Amanda 1350
Casey 2400
Jason 800


Looping through keys()
--------------------------------------------------------------------------------
Bryce Heather Kaylie Amanda Casey Jason

Looping through values()
--------------------------------------------------------------------------------
500 2000 25 1350 2400 800

Looping through items() as tuples
--------------------------------------------------------------------------------
('Bryce', 500)
('Heather', 2000)
('Kaylie', 25)
('Amanda', 1350)
('Casey', 2400)
('Jason', 800)


Items() unpacked as keys and values
--------------------------------------------------------------------------------
Name    ~  Points
Bryce   ~     500
Heather ~    2000
Kaylie  ~      25
Amanda  ~    1350
Casey   ~    2400
Jason   ~     800
```

```
Data types of returned views
------------------------------------------------------------------------
  keys() <class 'dict_keys'>
values() <class 'dict_values'>
 items() <class 'dict_items'>
$
```

Sorting Collections

Lists provide the `sort()` method, which sorts the `list` object in place.

Alternatively, you can use the built-in `sorted()` function that takes an iterable object as its parameter and returns a new sorted `list`.

There are many ways to use the `sort()` method and the `sorted()` function to sort data in various types of collections.

- A keyword parameter, `reverse=True`, can be passed to reverse the natural order of the objects (typically being ascending order)
- A customized sort can be achieved by passing another keyword parameter named `key` to specify a function of one argument that is used to extract a comparison key from each list element.

The example that follows demonstrates basic sorting of a list in both ascending and descending order.

basic_sorting.py

```
#!/usr/bin/env python3
numbers = [3, 1, -10, 54, 75, 29]
words = ["Hello", "Goodbye", "goodbye", "hello"]
label1, label2 = ("  Unsorted:", "  Sorted:")
print("Basic sorting of a list.")
print(label1, numbers, words)
numbers.sort()
words.sort()
print(label2, numbers, words, "\n")
numbers = [3, 1, -10, 54, 75, 29]

print("Basic sorting of a list in reverse.")
print(label1, numbers, words)
numbers.sort(reverse=True)
words.sort(reverse=True)
print(label2, numbers, words, "\n")
```

Note that calling `sort()` on a `list` updates the `list` in place.

If the desire is to obtain a sorted version of the list and leave the original list unchanged, then the built-in `sorted()` function can be used instead.

The built-in `sorted()` function returns a new sorted list from the items in iterable object that is passed to it as a parameter.

This allows a new sorted `list` to be obtained from a `list`, `tuple`, `set`, or `dict` object.

- The next example demonstrates sorting each of the above data types in ascending order.
- Note that the program uses the `__name__` class property, obtained by the `type()` function, to print the data type name.

sorting_a_collection.py

```
#!/usr/bin/env python3
originals = [[3, 1, -10, 54, 75, 29],
             ("Cheese", "Pepperoni", "Bacon", "Mushrooms"),
             {"AL", "NY", "MD", "VA", "PA", "KY", "VT"},
             {'New Hampshire': 'NH', 'Maryland': 'MD',
              'Nevada': 'NV', 'Maine': 'ME'}]

print("Original Collections")
for collection in originals:
    print(collection)
print()
for collection in originals:
    sorted_list = sorted(collection)
    print(type(collection).__name__, "sorted:", sorted_list)
print()
print("Original Collections")
for collection in originals:
    print(collection)
```

It is important to realize that each of the original collections remains unchanged because the `sorted()` function always returns a new `list` object, without modifying the original collection being passed as an argument.

The `sorted()` function also supports the `reverse=True` keyword argument to sort in descending order.

Custom Sorting

As mentioned earlier, a customized sort can be achieved by passing a named argument called `key`.

- The value of the `key` keyword parameter should be a reference to a function that takes a single argument.
- The return value of the function, when invoked by the `sort()` method or `sorted()` function, will then be used as the comparison key from each element in the list being sorted.

The next example demonstrates sorting a list of strings by the length of each string.

- This is accomplished by passing the built-in `len()` function as the keyword parameter value to the list's sort method `sort(key=len)`

custom_sorting.py

```
#!/usr/bin/env python3
names = """Smith Johnson Williams Brown Jones Miller Lee
Garcia Rodriguez Wilson Martinez Anderson Taylor
```

```
Thomas Hernandez Moore Martin Jackson Thompson
White Lopez Davis"""
names = names.split()
# Primary sort by name ("Alphabetically)
names.sort()
# Secondary sort by length
names.sort(key=len)
print(names)
```

The output of the above program is shown next.

```
$ python3 custom_sorting.py
['Lee', 'Brown', 'Davis', 'Jones', 'Lopez', 'Moore', 'Smith','White', 'Garcia',
'Martin', 'Miller', 'Taylor', 'Thomas', 'Wilson', 'Jackson', 'Johnson',
 'Anderson',
'Martinez', 'Thompson', 'Williams','Hernandez', 'Rodriguez']
$
```

The two sorts together in the preceding example results in any names of the same length being sorted alphabetically.

The next program shows how a custom function can be used as an argument to the `sort()` method.

- The list to be sorted consists of full names, with the desire to sort the list by last name only.
- This requires a custom function that can separate the last name from the string so it can be used as the sort criteria.

custom_sort_function.py

```
#!/usr/bin/env python3
def the_last_word(a_string):
    fmt = "For Input: {:18}    Sort Using: {}"
    last_word = a_string.strip().split()[-1]
    print(fmt.format(a_string, last_word))
    return last_word


names = """Ava Smith, Ethan Johnson, Abigail Williams, \
Sophia Brown, Michael Jones, Emily Miller, Declan Lee"""
names = names.split(", ")
print(names)
names.sort(key=the_last_word)
print(names)
```

The output of the above program is shown next.

```
$ python3 custom_sort_list.py
['Ava Smith', 'Ethan Johnson', 'Abigail Williams', 'Sophia Brown', 'Michael
 Jones',
'Emily Miller', 'Declan Lee']
For Input: Ava Smith          Sort Using: Smith
```

```
For Input: Ethan Johnson      Sort Using: Johnson
For Input: Abigail Williams   Sort Using: Williams
For Input: Sophia Brown       Sort Using: Brown
For Input: Michael Jones      Sort Using: Jones
For Input: Emily Miller       Sort Using: Miller
For Input: Declan Lee         Sort Using: Lee
['Sophia Brown', 'Ethan Johnson', 'Michael Jones', 'Declan Lee', 'Emily Miller',
'Ava Smith', 'Abigail Williams']
$
```

The following example demonstrates sorting the contents of a dictionary by values instead of keys.

dictionary_sorts.py

```
#!/usr/bin/env python3
def get_value(akey):
    return states[akey]


states = {'New Hampshire': 'NH', 'Maryland': 'MD',
          'Nevada': 'NV', 'Maine': 'ME'}

# Sorted by Values
long_names = list(states.keys())
long_names.sort(key=get_value)
for name in long_names:
    print(name, states[name])
print()

# Sorted again by value without the need for custom function
long_names = list(states.keys())
long_names.sort(key=states.get)
for name in long_names:
    print(name, states[name])
print()
```

The output of the preceding program is shown next.

```
$ python3 dictionary_sorts.py
Maryland MD
Maine ME
New Hampshire NH
Nevada NV

Maryland MD
Maine ME
New Hampshire NH
Nevada NV

$
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `collections/solutions` directory.

Exercise 1

Given the following two lists:

```
first = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]
second = ["day", "day", "sday", "nesday", "rsday", "day", "urday"]
```

- Concatenate the two lists by index into a new list that, when printed, looks as follows:

```
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]
```

Exercise 2

Write a program that creates a loop asking the user to input a number.

- Repeat this process until the user enters the value `end`.
  - The following can be used to loop through the user input.

```
prompt = "Enter a number (or the word 'end' to quit) "
while True:
    data = input(prompt)
    if data == "end":
        break
    #Remainder of while loop goes here
```

- Add each iteration number to a `list`.
- Just before the program ends, print the following:
  - The contents of the list on one line
  - The sum of the elements in the list on the second line

Exercise 3

Write a program that creates a loop asking the user to input a number.

- Repeat this process until the user enters the value `end`.
- Enter each number into a `set`.
  - Before you enter the number, verify if the number is already in the set.
  - If the number is already in the set, then update a counter that tracks how many entries are not added to the set.
- Just before the program ends, print the following:
  - The contents of the set on one line

  - The number of elements that were NOT added to the set on the second line

Exercise 4

Use a single `set` to determine the number of unique words in the user's input.

- You can use the same sample `while` loop from Exercise 1.
  - Each time through the loop, the individual words should be added to the single set.
- When done looping, output the contents of the set sorted alphabetically.
- Also, output the number of unique words.

Exercise 5

Use a dictionary to create a mapping from the digits 0-9 to the words `zero`, `one`, `two`, etc.

- Next, ask the user to input a number.
- If the user enters `1437`, then the program should print `one four three seven.`

Exercise 6

Rewrite Exercise 4 to count the frequency of each word in the user's input.

- A `dict` provides the perfect data structure for this problem.
  - Let the words be the keys, and let the counts be the values.
- Print the results sorted by the words.
- Finally, print the results sorted by the counts.

Summary

- You can use the `list` data type to dynamically persist an ordered collection of elements.
- The `tuple` data type is similar to the `list` type but it is more efficient and immutable.
- When your application requires an unordered collection of elements without duplicates, you can use the `set` data type.
- For fast lookups in data that has unique and hashable keys, you can use the `dict` data type.

Chapter 4

Functions

Goal

Encapsulate code in small and reusable logic blocks.

Objectives

- Define and use custom functions within a Python program.
- Define functions that allow passing of named and optional arguments.
- Define and use functions that allow a variable number of arguments.
- Understand access the various scopes of variables within a Python application.
- Pass references to functions the same way references to other objects are used.
- Use the map and filter datatypes to apply a function to iterable objects.
- Understand and use nested functions and lambdas.
- Define and use recursive functions.

Sections

- Functions

Functions

Objectives

- Define and use custom functions within a Python program.
- Define functions that allow passing of named and optional arguments.
- Define and use functions that allow a variable number of arguments.
- Understand access the various scopes of variables within a Python application.
- Pass references to functions the same way references to other objects are used.
- Use the map and filter datatypes to apply a function to iterable objects.
- Understand and use nested functions and lambdas.
- Define and use recursive functions.

Introduction

Large programs are more easily understood, maintained, and debugged if they are divided into manageable reusable pieces.

One way to do this is by using functions.

- A function represents a body of code that can be written once and then executed whenever the need arises, possibly multiple times.
- The function might be encoded in the program that needs it, or it may be located in a separate file and used by any program(s) that needs it.

Functions can also be grouped together into a library and reused over a series of applications by the entire enterprise.

- Libraries in Python are called **modules**.
- Modules will be discussed in more detail in he next chapter.

We have already seen many functions that are part of the standard Python library.

Here are just some of the functions that have been used.

`input()`	`oct()`	`str()`
`print()`	`bin()`	`sorted()`
`len()`	`int()`	`type()`
`hex()`	`float()`	

However, you can always write your own custom functions, and this chapter explores the details of doing that.

Defining Your Own Functions

The keyword `def` creates a function.

- Following that keyword is the name of the function and a set of parenthesis, followed by a colon.
- Define parameters inside the parenthesis if necessary.

The following example is a simple function that takes some text as a parameter and displays it with a border in the output.

- The definition of a function must precede its invocation.

first_function.py

```
#!/usr/bin/env python3
def print_with_border(some_text):
    border = len(some_text) * "#"
    print(border)
    print(some_text)
    print(border)


print_with_border("Hello")
print_with_border("Goodbye")
data = input("Enter some text: ")
print_with_border(data)
```

```
$ python3 first_function.py
#####
Hello
#####
#######
Goodbye
#######
Enter some text: Functions are very powerful
##########################
Functions are very powerful
##########################
$
```

Parameters and Arguments

In general, functions are more useful when you can pass data to them and have the function operate on the data.

- The function definition declares the parameters you can pass to it.
- Pass arguments in the function call for the function to use in the place of each parameter.
  - Each argument that is sent to the function must have a corresponding parameter defined in the header part of the function.
  - The arguments are passed by reference to the function, which is an important aspect of an object-oriented language.

- Optionally, a function may return a value using the `return` keyword.
  - If no `return` is specified, a function returns a reference to the `None` object, by default.

In the previous example, the `print_with_border` function defines a single parameter named `some_text`.

- The example below modifies the previous function to build a string and return it to the caller as opposed to printing it.
- Ultimately the result is the same as the previous, but now it is the caller of the function that decides to print the result.

second_function.py

```
#!/usr/bin/env python3
def wrap_with_border(some_text):
    result = [len(some_text) * "#"]
    result.append(some_text)
    result.append(result[0])
    return "\n".join(result)


data = wrap_with_border("Hello")
print(data)
print(wrap_with_border("Goodbye"))
```

Function Documentation

To document a function, define a literal string as the first statement within the body of the function.

- The interpreter accepts everything in this string as documentation.
- Accessing the documentation string is then available to programmers through Python's help() function or through a special property of the function object named `__doc__`.

```
help(function_name)
print(function_name.__doc__)
```

Notice the output from the following program.

third_function.py

```
#!/usr/bin/env python3
def wrap_with_border(some_text):
    """ Returns a string of some_text with a line of # signs
        before and after it"""
    result = [len(some_text) * "#"]
    result.append(some_text)
    result.append(result[0])
    return "\n".join(result)
```

```
print("The doc string is:\n", wrap_with_border.__doc__)
```

The output of the above program is shown next.

```
$ python3 third_function.py
The doc string is:
  Returns a string of some_text with a line of # signs
        before and after it
$
```

References

More information about docstring conventions can be found in PEP 257 at the following URL

https://www.python.org/dev/peps/pep-0257/

Named Arguments

Python provides several ways to pass arguments to functions.

- You can always pass the arguments in the traditional way.

traditional_args.py

```
#!/usr/bin/env python3
def volume(length, width, height):
    """Returns the volume of a box for given dimensions"""
    return length * width * height


dim1 = float(input("Enter length of the box: "))
dim2 = float(input("Enter width of the box: "))
dim3 = float(input("Enter height of the box: "))
result = volume(dim1, dim2, dim3)
print("The volume is:", result)
```

- Python also allows passing named arguments to a function, where the name of the argument matches the name of a parameter of the defined function.
- An advantage of the named argument approach is that the arguments can be passed in any order as shown in the next example.
- Named arguments make calling the function easier to read.
- Another advantage is that the named arguments make the purpose of each argument clear.

named_args.py

```
#!/usr/bin/env python3
def volume(length, width, height):
    """Returns the volume of a box for given dimensions"""
    return length * width * height


dim1 = float(input("Enter length of the box: "))
dim2 = float(input("Enter width of the box: "))
dim3 = float(input("Enter height of the box: "))
result = volume(length=dim1, width=dim2, height=dim3)
print("The volume is:", result)

result = volume(height=dim3, length=dim1, width=dim2)
print("The volume is:", result)
```

Optional Arguments

A function can also define default values for parameters.

- When a parameter is defined with a default value, it makes the passing of the value for it as an argument optional as shown next.

optional_args.py

```
#!/usr/bin/env python3
def volume(length=10, width=5, height=2):
    """Returns the volume of a box for given dimensions"""
    return length * width * height


print("1:", volume(2, 3, 4))
print("2:", volume(2, 3))
print("3:", volume(2))
print("4:", volume())

vol = volume(30, height=4, width=20)
print("5:", vol)

print("6:", volume(height=3))
print("7:", volume(height=7, width=2))
```

```
$ python3 optional_args.py
1: 24
2: 12
3: 20
4: 100
5: 2400
6: 150
7: 140
$
```

Whenever positional arguments and named arguments are passed within the same function call, the named arguments must be to the right of all other positional arguments.

The following attempt at calling the volume would be invalid syntax:

```
volume(length=10, width=20, 30)
```

Passing Collections to a Function

Here is a function that takes a list as a parameter named `a_list` and updates the list by multiplying each of its elements by the `qty` parameter.

pass_list.py

```
#!/usr/bin/env python3
def multiply_by(qty, a_list):
    for index, value in enumerate(a_list):
        a_list[index] = value * qty


def process_list(a_list):
    print("Before:", a_list)
    multiply_by(5, a_list)
    print("After:", a_list)


data = [10, 20, 30, 40]
process_list(data)

data = ["Me", "the", "Hello"]
process_list(data)
```

The output of the above program is shown next.

```
$ python3 pass_list.py
Before: [10, 20, 30, 40]
After: [50, 100, 150, 200]
Before: ['Me', 'the', 'Hello']
After: ['MeMeMeMeMe', 'thethethethethe', 'HelloHelloHelloHelloHello']
$
```

In the example, the `data` variable stores each list.

- The first call to `process_list(data)` passes a copy of the object reference to the `process_list` function. The `a_list` variable is assigned to the reference while inside the body of the function.

- At this point in the program, there has only been one list created by the program, but two references (`data` and `a_list`) are referencing the single list of `[10, 20, 30, 40]`.

The previous example worked with a lists that were composed of immutable objects (Strings and integers).

- The function extracts each element of the list and then updates the value, overwriting the previous value.
- The function uses the built-in `enumerate()` function to get the index of the value. The index is then used to write the result back into the same place.

```
for index, value in enumerate(a_list):
    a_list[index] = value * qty
```

The next example, passes a dictionary collection where each value in the dictionary is a list.

- The dictionary contains a collection of lists as its values.
- The goal is to add a new element to every list in the dictionary.
- The example demonstrates updating the value in a collection as opposed to replacing each element in a collection.

The output of the above example is shown next.

The diagram below shows the variables and references in memory that represent the dictionary of students.

- The dark black represents the `students` variable defined on line 13 of the code
- The gray indicates the variables inside the `update_stickers` function.

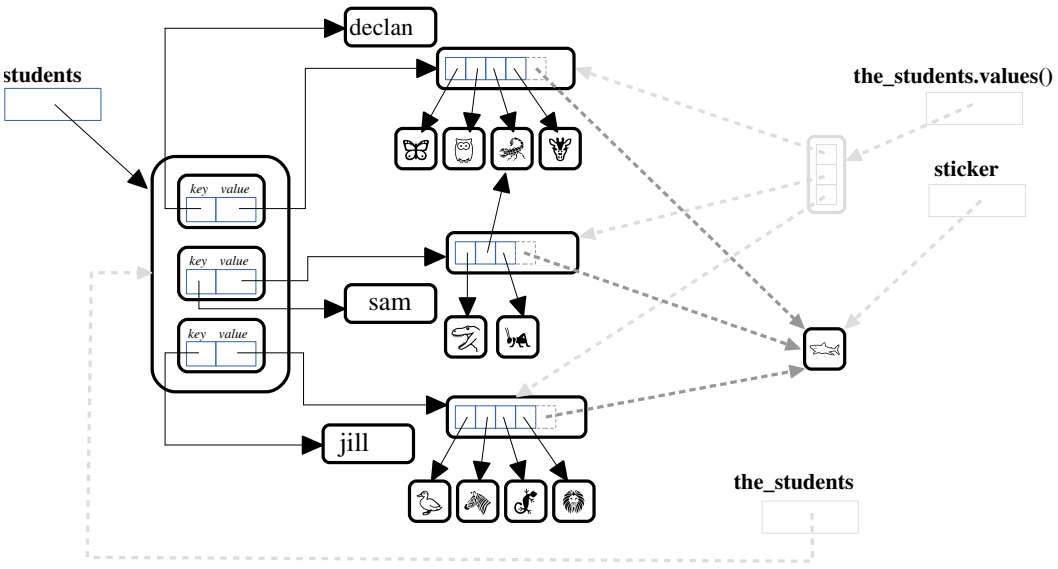


Figure 4.1: Variables and references representation

Variable Number of Arguments

Python functions are capable of accepting a variable number of arguments as a single parameter to the function.

- These arguments are either supplied by position or by a keyword.
  - A single asterisk in front of the parameter is used for a variable number of positional arguments and the data type of the parameter is always a `tuple`.

  - A double asterisk in front of the parameter is used for a variable number of keyword arguments and the data type of the parameter is always a `dict`.

The following example defines a function that defines a parameter that takes a variable number of arguments.

- The parameter uses an asterisk in front of it, and as such requires that the arguments be passed positionally when the function is called.

variable_position.py

```
#!/usr/bin/env python3
def the_sum(*args):
    total = 0
    print("Parameter type:", type(args), end=" ")
    for elem in args:
        total += elem
    return total


total = the_sum(1, 2, 3, 4, 5)
print("Sum is: ", total)
total = the_sum(5, 2, 7)
print("Sum is: ", total)
```

The output of the above program is shown next.

```
$ python3 variable_position.py
Parameter type: <class 'tuple'> Sum is: 15
Parameter type: <class 'tuple'> Sum is: 14
$
```

There may be occasions when you need to define a function that takes a variable number of arguments and several other items.

- Any parameters that are declared after the parameter representing the variable number of arguments must be called as named arguments.

- Such parameters, if not given a default value, are required arguments when the function is called.

varargs.py

```
#!/usr/bin/env python3
def modify(qty, *values, end="\n"):
    for val in values:
        print(qty * val, end=end)


modify(3, "Hello", "Bye", "Sample", end="|")
print()
```

```
modify(4, 10, 20, 30, end=" ~ ")
print()
modify(15, 2, 3, 4)
```

The output of the above program is shown next.

```
$ python3 varargs.py
HelloHelloHello|ByeByeBye|SampleSampleSample|
40 ~ 80 ~ 120 ~
30
45
60
$
```

The example on the following page defines a function that accepts as a parameter a variable number of arguments.

- The parameter uses a double asterisk in front of it, and as such requires that the arguments be passed as keyword arguments when the function is called.

Variable Number of Keyword Arguments

The following example defines a function that accepts as a parameter a variable number of arguments, that when passed will be passed as keyword arguments.

yahtzee_scores.py

```
#!/usr/bin/env python3
def print_score(player, **scores):
    total_score = 0
    print("Player:", player)
    for category, score in scores.items():
        print("{0:>15}: {1}".format(category, score))
        total_score += score
    print("{0:>15}: {1}".format("Total", total_score))


print_score("Aiden", Aces=4, Twos=8, FullHouse=25, LgStraight=40)
print_score("Cindy", Twos=4, LgStraight=40, Chance=24, ThreeOfAKind=21)
```

The output of the above program is shown next.

```
$ python3 yahtzee_scores.py
Player: Aiden
     LgStraight: 40
           Twos: 8
      FullHouse: 25
           Aces: 4
          Total: 77
Player: Cindy
     LgStraight: 40
   ThreeOfAKind: 21
```

```
            Twos: 4
          Chance: 24
           Total: 89
$
```

Scope

In programming languages, the term scope refers to that part of the program where a symbol is known.

Python defines the following scopes.

- **local**
  - *local* scope refers to the same suite of statements.
  - Variables assigned within the same suite are local to the suite.
- **global**
  - Names with the *global* scope are available to all statements in the module.
  - If the developer does not make a variable *global*, then it is assumed *local* to the suite.
- **built-in**
  - Names in the *built-in* scope are defined by Python and are available to all statements in the application.
- **nonlocal**
  - *nonlocal* scope requires nested functions and is introduced separately in the coming topic of nested functions.

When a symbol (variable name, function name, class name, etc.) is referenced in the application, Python will always search for the symbol in the local scope first.

- If the symbol is not found in the local scope, Python then searches the global scope for the symbol.
- The built-in scope is only searched if the symbol is not found in the local or global scopes.

The examples on the next page demonstrate the difference between a global and a local variable.

local.py

```
#!/usr/bin/env python3
def demo():
    count = 0
    print("Inside Function:", count)


count = 5
print("Before Function:", count)
demo()
print("After Function: ", count)
```

The output of the above program is shown next.

```
$ python3 local.py
Before Function: 5
Inside Function: 0
After Function: 5
$
```

The following example makes access to the `count` a global variable inside of the function using the `global` keyword.

global.py

```
#!/usr/bin/env python3
def demo():
    global count
    count = 0


count = 5
print("Before Function:", count)
demo()
print("After Function:", count)
```

The output of the above program is shown next

```
$ python3 global.py
Before Function: 5
After Function: 0
$
```

Functions - "First Class Citizens"

Most programmers are comfortable with the notion of sending data to a function.

In Python, it is just as easy to send a function to a function. To understand this, it is important to recognize the difference between the following two notions.

- `fun()` = invoking the function named fun
- `fun` = referencing the function named fun

The second of the two forms has some special uses.

- The following example defines a generalized `compute` function that, in addition to taking a `list` as an argument, also takes a function.

firstclass.py

```
#!/usr/bin/env python3
def square(p):
    return p * p
```

```
def increment(p):
    return p + 1


def compute(func, lis):
    for index, item in enumerate(lis):
        lis[index] = func(item)


data = [10, 20, 30, 40]
compute(square, data)
print(data)
compute(increment, data)
print(data)
```

The output of the above program is shown next.

```
$ python3 firstclass.py
[100, 400, 900, 1600]
[101, 401, 901, 1601]
$
```

The map Function

Python has several built-in data types that apply a function to each of the elements of an iterable object.

- Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`.

One of these built-in data types is the `map` data type that has the following syntax.

```
map(function_name, an_iterable, ...)
```

The above constructor instantiates an object of type `map`.

- A `map` object is an iterator that applies `function_name` to every item of `an_iterable`, yielding the results.
- If additional iterable arguments are passed, `function_name` must take that many arguments and is applied to the items from all iterables in parallel, the iterator stops when the shortest iterable is exhausted.

The example below uses map to convert a list of strings to an iterator of `int` values.

map_demo.py

```
#!/usr/bin/env python3
data = ["2", "4", "6", "8"]
values = map(int, data)
print(type(values), ":", values)
total = 0
for value in values:
    total += value
print("Sum of numbers in {} = {}".format(data, total))
```

The output of the above example is shown next.

```
$ python3 map_demo.py
<class 'map'> : <map object at 0x7f8d4f11c898>
Sum of numbers in ['2', '4', '6', '8'] = 20
$
```

The following example demonstrates passing multiple iterables as arguments to the `map` constructor.

average_grades.py

```
#!/usr/bin/env python3
tests = {"Sally": (89, 78, 99, 88, 92, 98, 95, 78, 88),
         "Doug": (68, 87, 72, 60, 80, 65),
         "Kesha": (98, 87, 99, 78, 99, 80, 98, 50),
         "John": (89, 78, 99, 88, 92, 99, 95, 88, 95, 99)}


def averages(*grades):
    qty = len(grades)
    return sum(grades)/qty


a, b, c, d = tests.values()
x = map(averages, a, b, c, d)
print("Averages:", list(x))


# Notice that when an asterisk is used on an iterable
# argument when a function is called, that it unpacks the
# iterable automatically into the arguments passed to the
# function as seen below. This * operator when used in
# this manner is often referred to as the "splat" operator
# in other languages.

x = map(averages, *tests.values())
print("Averages:", list(x))
```

The output of the above program is shown next.

```
$ python3 average_grades.py
Averages: [86.0, 82.5, 92.25, 78.5, 90.75, 85.5]
Averages: [86.0, 82.5, 92.25, 78.5, 90.75, 85.5]
$
```

filter

Another built-in data type is the `filter` data type that has the following syntax.

```
filter (function_name, an_iterable)
```

The above constructor instantiates an object of type `filter`.

- A `filter` object is an iterator that applies `function_name` to every item of `an_iterable`, yielding all the results for which the function returns `True`.

filter_demo.py

```
#!/usr/bin/env python3
def multiple_of_three(x):
    return x % 3 == 0


results = filter(multiple_of_three, range(2, 51))
for value in results:
    print(value, end=' ')
print()
```

The output of the above program is shown next.

```
$ python3 filter_demo.py
3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48
$
```

A Dictionary of Functions

The following is an example of creating a menu whose keys map to functions.

function_menu.py

```
#!/usr/bin/env python3
def add():
    val = input("Enter value to add: ")
    data.append(val)


def delete():
    item = data.pop(0)
    print("removing", item)
```

```
def display():
    print("displaying:", data)


def terminate():
    print("terminating")
    exit()


def illegal():
    print("Illegal Selection\n")


data = []
menu = {"1": add, "2": delete, "3": display, "4": terminate}
keys = sorted(menu.keys())
while True:
    print("Make selection:")
    for key in keys:
        print("\t", key, menu[key].__name__)
    key = input(">")

    menu.get(key, illegal)()
```

When a user selects a menu item, that value is used as the key into the dictionary in `menu.get(key, illegal)()`.

The empty parenthesis at the end invoke the function being referenced by the value returned from the `get` method.

Nested Functions

In Python, a function can be nested inside another function.

The inner function will have access to any variables in the scope of the outer function.

nested.py

```
#!/usr/bin/env python3
def outer(a, b):
    x = 15
    y = 20

    def inner(z):
        print(a, b, x, y, z)

    return inner  # a reference to inner function


result = outer(5, 10)
print(type(result))
result(9)  # invoke the returned function
```

The output of the above program is shown next.

```
$ python3 nested.py
<class 'function'>
5 10 15 20 9
$
```

The example above demonstrates a few principles.

- A function can return another function.
- When the nested function is defined, its variables can reference variables from the containing scope.
  - If the inner function needs to modify the value of a variable from the containing scope it must declare access to the variable is being done as `nonlocal` scope.

lambda

Python supports the runtime creation of *anonymous* functions (functions that are not bound to a name) using a `lambda` expression.

This is a powerful concept and one that is often used in conjunction with functions and data types, such as `filter` and `map`.

- `lambda` expressions must be one-liners.
- The parameters are defined before the colon.
- The work of the function is defined after the colon.
- A return statement is unnecessary.

The following example uses a `lambda` as the function to be passed to the filter function.

filter_with_lambda.py

```
#!/usr/bin/env python3
results = filter(lambda x: x % 3 == 0, range(2, 51))

for value in results:
    print(value, end=' ')
print()
```

The next example is a rewrite, of the previous nested functions example, that defines the inner function as a lambda.

nested_lambda.py

```
#!/usr/bin/env python3
def outer(a, b):
    x = 15
    y = 20

    return lambda z: print(a, b, x, y, z)
```

```
result = outer(5, 10)
print(type(result))
result(9)  # invoke the returned function
```

Recursion

One advanced form of defining and using a function is known as recursion.

A *recursive function* is a function that calls itself that has the following two features:

- A call to itself.
- A termination condition, when the function does not call itself.

Recursion is often a conceptually easier way of describing a problem, but sometimes has a tendency to become less efficient due to the possibly large number of function calls involved.

The following example is a recursive function that adds numbers from `1` to and upper limit `n`.

recursive_sum.py

```
#!/usr/bin/env python3
def sum_to(n):
    if not n:     # This is the termination condition
        return n

    return n + sum_to(n-1)  # This is where the recursion is


limit = 6
print("Sum from 1 to", limit, "is:", sum_to(limit))
```

The last line of code inside of the `sum_to` function recursively calls the same function.

- The `if` statement on line 3 of the code acts as the terminating condition where the function no longer calls itself.

The following example rewrites the above code to include additional print statements to get a better understanding of the intermediate steps throughout the recursive calls.

recursive_sum_verbose.py

```
#!/usr/bin/env python3
indent = 0
text = "sum_to"


def sum_to(n):
    global indent
    print(" " * indent, text, n)
    indent += 1
    if not n:
```

```
        indent -= 1
        print(" " * indent, text, n, " => ", n)
        return n
    result = sum_to(n - 1)
    indent -= 1
    print(" " * indent, text, n, end="")
    print(" => {0:2} + {1:2} => {2:2}".format(n, result, n + result))
    return n + result


limit = 6
print("\nSum from 1 to", limit, "is:", sum_to(limit))
```

The added `print` statements show the values of `n` and result `through` each recursive call to `sum_to` as shown next.

```
$ python3 recursive_sum_verbose.py
 sum_to 6
  sum_to 5
   sum_to 4
    sum_to 3
     sum_to 2
      sum_to 1
       sum_to 0
       sum_to 0 => 0
      sum_to 1 => 1 + 0 => 1
     sum_to 2 => 2 + 1 => 3
    sum_to 3 => 3 + 3 => 6
   sum_to 4 => 4 + 6 => 10
  sum_to 5 => 5 + 10 => 15
 sum_to 6 => 6 + 15 => 21

Sum from 1 to 6 is: 21
$
```

Recursion is often a more elegant approach to solving a problem, but can result in lessened performance.

- A call such as `sum_to(50)` will make 50 function calls, whereas an iterative approach may only require one.

Python has a limit called the recursion limit.

- If a function recursively calls itself too many times, it will raise an exception of type `RecursionError`.
- While this limit can be changed, hitting the limit should act as a red flag that the recursive call may be to inefficient to use.

The example below is an iterative approach to the problem as opposed to the previous recursive approach.

iterative_sum.py

```
#!/usr/bin/env python3
def iterative_sum_to(n):
    total = 0
    for i in range(n, 0, -1):
        total += i
    return total


limit = 6
print("Sum from 1 to", limit, "is:", iterative_sum_to(limit))
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `functions/solutions` directory.

Exercise 1

Write and test a function that is designed to validate input.

- The function should prompt the user for a positive integer.
- It should validate the information entered by the user is indeed a positive integer.
  - If number entered is valid, the function should return the number.
  - If the number entered is invalid, the function should return a zero (*0*) instead.
- The application, not the function, should indicate with a message in the output each time an invalid entry is given.

Exercise 2

Write and test a function that takes a collection of strings and returns the length of the longest string in the collection.

- The application should loop through the collection of strings and rely on the value returned by the function to format all of the strings to the output such that they are all right justified to the width of the longest string.

Exercise 3

There is a built-in function in Python called `sum` that will return the sum of all of the numbers of an iterable object.

- Write a similar function, but instead of taking a collection as a parameter, the function should take a variable number of arguments and return the sum of them.

Exercise 4

Rewrite the function in Exercise 3 above to return a tuple instead of a sum.

- The tuple should be the sum and the average of all of the arguments passed to the function.

Exercise 5

Write a calculator application that presents the following menu:

```
Calculator options:
      1. Add
      2. Subtract
      3. Multiply
      4. Divide
      5. Quit
```

- The user is expected to enter a number from the above menu.
  - After choosing the operation, the user should be prompted twice for 2 numbers and the chosen operation performed on them with the result being displayed on the screen.
  - Each of the above options should be implemented as its own function.

Exercise 6

Write and test a function that receives a list as its only parameter and returns a new list of the positive elements only.

Exercise 7

Write and test a function that takes a variable number of arguments as its first parameter and a number as its second parameter.

- The function should return the count of the values in the tuple parameter (the variable number of arguments) that are greater than the second parameter (*num* in the sample below).
- For example, one such call to a function named *positive* is shown below.

```
res = positive(5, -10, 10, -20, 30, num=0)
```

- In this case, the function would return a value of *3*.

Exercise 8

Write a function that returns a nested function.

- When the nested function is executed it should return the sum of two integers.
- The two parameters should be passed to the outer function and used by the inner function.

Exercise 9

Re-write your solution to Exercise 8 such that the outer function receives no parameters, and the nested function is defined as taking the two parameters.

Exercise 10

Re-write your solution to either Exercise 8 or 9 so that it uses a lambda expression as the nested function.

Exercise 11

Write and test a function that takes two lists as parameters and returns a list of the elements that are common to both.

Exercise 12

Write and test a function that takes a number and a dictionary and adds the number to all values in the dictionary.

- You can assume that all the values in the dictionary are numbers.

Exercise 13

While the `index` method of a `list` can be used to find either the first occurrence of an item or the first occurrence of it within a range of the list, it does not allow you to find, say the second or third occurrence by passing in a number as to the one you are looking for.

- Write a function that takes 3 parameters and returns what it finds.
  - One being the list to search.
  - The second being the object to search for.
  - The thrid being an int representing which one you are looking for such as the fist, second, third occurrence.
- The index method raises a `ValueError` exception if the value being searched for does not exit in the list.
  - It is perfectly fine for your function to behave in the same manner.

References

More information about docstring conventions can be found in PEP 257 at the following URL

`https://www.python.org/dev/peps/pep-0257/`

Summary

- You can organize your code into reusable pieces by creating functions with the `def` keyword.
- You can write function calls that are easier to read by combining positional and named arguments.
- Functions are "first-class citizens" in Python. You can pass a function as a parameter to another function.
- Python supports nesting functions. You can define a function inside another function's body.
- You can define anonymous functions in Python by using `lambda` expressions.

Chapter 5

Modules

Goal

Reuse code organized and grouped into independent files.

Objectives

- Use modules to hold Python definitions and statements.
- Define and access modules from other modules.
- Use the `dir` function to list available symbols with a module's global scope.
- Check the index of Python standard library modules for modules that may be helpful.
- Understand and use the various properties and functions of Python's `sys` module.
- Write programs that use various numeric and mathematical modules from Python's standard library.
- Write programs that use the various date and time modules from Python's standard library.

Sections

- Modules

Modules

Objectives

- Use modules to hold Python definitions and statements.
- Define and access modules from other modules.
- Use the `dir` function to list available symbols with a module's global scope.
- Check the index of Python standard library modules for modules that may be helpful.
- Understand and use the various properties and functions of Python's `sys` module.
- Write programs that use various numeric and mathematical modules from Python's standard library.
- Write programs that use the various date and time modules from Python's standard library.

What is a Module?

By using functions, we can better organize our code for reuse within a program.

- As your scripts get longer, you may want to split them into several files for easier maintenance.
- You may also want to use a handy function that you have written in several programs without copying its definition into each program.

Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a **module**.

A **module** is a file containing Python definitions and statements.

- The module name is the file name without the `.py` extension.
- Definitions from a module can be imported into other modules or into the main module.
- The main module (named `__main__`) is the collection of variables that you have access to in a script executed at the top level of a script and in an interactive Python shell.

Within a module, the module's name is accessible via the value of the global variable `__name__`.

The following example defines several functions, that can be used by multiple Python applications, in a file named `reusable.py`.

- When the `reusable` module is run from the command line, its acts as the main module and as such is assigned a `__name__` of `__main__` when loaded.
- The statements in the module that invoke the functions are placed inside of a conditional statement that only runs when the module's name is `__main__`.

The example and its output is shown on the following page.

Modules

reusable.py

```
#!/usr/bin/env python3
def main():
    print("Testing my functions at top level", square(5), cube(10))


def square(p):
    return p ** 2


def cube(p):
    return p ** 3


if __name__ == "__main__":
    main()
```

```
$ python3 reusable.py
Testing my functions at top level 25 1000
$
```

Placing the application code in a function called `main` and calling it conditionally as shown above is a very Pythonic way of defining a Python application.

Note

Pythonic means code that follows the conventions of the Python community. You can learn more about this coding style in the following URL.

`https://peps.python.org/pep-0008/`

Applications desiring to use the above functions must import the `reusable.py` file by its module name.

app.py

```
#!/usr/bin/env python3
import reusable


def main():
    print(reusable.square(5), reusable.cube(5))


if __name__ == "__main__":
    main()
```

When the `app` module in the previous example is run from the command line, its name becomes `__main__`.

- Upon importing the `reusable` module, the `if` statement in the `reusable` module does not execute the `main` function because it's name is `reusable` in this context instead of `__main__`.

Each module has its own symbol table, which contains the identifiers (names) used in that module.

- Many languages refer to these symbol tables as **namespaces**.
- In the absence of any imported modules, Python always looks for names in the namespace of the main module.
- To direct Python to look into another module in order to resolve a name, use the module name ahead of the function name, as in the code next.

```
result = reuseable.square(5)
```

Alternately, you can import the entire `reuseable` symbol table into that of the currently running main module.

alternate.py

```
#!/usr/bin/env python3
from reusable import *


def main():
    print("Module name is: " + __name__)
    print(square(5), cube(5))


if __name__ == "__main__":
    main()
```

- The * wildcard used above does not actually import all of the module's symbols into the top level symbol table.
  - It ignores any symbols that start with an underscore.
  - If a list of strings named `__all__` is found in the module, only the symbols named in the list are imported into the namespace.

The dir Function

The built-in `dir` function determines which symbols a module defines.

- It returns a sorted list of strings.
  - When called with no parameters, it returns the symbols available in the main module.
  - When a module name is passed as an argument, it returns a list of all of the symbols available in that module's symbol table.

The interactive Python shell below demonstrates various calls to the built-in `dir()` function.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__','__name__',
'__package__', '__spec__']
>>> x = 99
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__','__name__',
'__package__', '__spec__', 'x']
>>> import reusable
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__loader__','__name__',
'__package__', '__spec__', 'reusable', 'x']
>>> dir(reusable)
['__builtins__', '__cached__', '__doc__', '__file__','__loader__', '__name__',
'__package__', '__spec__', 'cube', 'main','square']
>>> exit()
$
```

A full list of the built-in symbols available to all modules can be seen by passing the variable `__builtins__` to the `dir()` function.

```
dir(__builtins__)
```

Python Standard Library Modules

Python comes with a library of standard modules, some of which are built into the interpreter.

- These provide access to operations that are not part of the core language but are built in, often either for efficiency or to provide access to operating system specific calls.

References

The modules available from the Python standard library, and their documentation, can be found at the following URL:

`https://docs.python.org/3/library/index.html`

- Many of the modules provide standardized solutions for many problems that occur in everyday programming.
- Some are designed to encourage and enhance the portability of Python programs by providing platform-neutral APIs.

There is a growing public collection of thousands of components, from individual programs to entire frameworks, available from the Python Package Index (PyPi).

- Keep in mind the following if using anything from the PyPi repository.
  - Arbitrary Python code can be executed during the installation.

  - Anyone can register an account and upload Python packages to the PyPi repository requiring trust of the maintainer of the package as to the safety of its contents.

References

The PyPi repository can be found at the following URL:
`https://pypi.python.org/pypi`

- Installation of packages from the repository is often done through a package management system named "pip".

The remainder of this chapter will focus on several commonly used modules that are part of the Python Standard Library.

The sys Module

The `sys` module provides access to variables and functions to interact with the interpreter. The following are several of the available properties and functions available in the module.

- `sys.path`
  - A list of strings that specifies the search path for modules.
  - Includes values from the environment variable PYTHONPATH.
  - The first item of this list, `sys.path[0]`, is the directory containing the script that was used to invoke the Python interpreter.
- `sys.argv`
  - A list of command line arguments passed to a Python script, where `sys.argv[0]` is the script name.
- `sys.version_info`
  - A tuple containing the five components of the version number: `major`, `minor`, `micro`, `releaselevel`, and `serial`.
  - The values can be obtained by index or by name, `sys.version_info[0]` or `sys.version_info.major` for example.
- `sys.exit([arg])`
  - Exits from Python. The optional argument `arg` can be an integer giving the exit status (defaulting to zero), or another type of object.
  - If it is an integer, zero is considered “successful termination” and any nonzero value is considered “abnormal termination” by shells and the like.
  - `sys.exit("Some error message")` would exit a program with the given message and an exit status of `1`.

References

The `sys` module's complete list of properties and functions and their documentation can be found at the following URL:

`https://docs.python.org/3/library/sys.html`

- The properties and functions listed above for the `sys` module are shown in the example on the following page.

sys_testing.py

```
#!/usr/bin/env python3
import sys


def main():
    print("Script Name:", sys.argv.pop(0))
    print("Remaining Command Line Arguments:", sys.argv)

    info = sys.version_info
    print("Python Version:")
    print(info.major, info.minor, info.micro, sep=".")

    print("Python Path:")
    for each_path in sys.path:
        print("\t", each_path)
    print()

    number = input("Please enter an integer")
    if not number.isdecimal():
        sys.exit(number + " is not an integer.")

    print("You entered the number " + number)


if __name__ == "__main__":
    main()
```

Numeric and Mathematical Modules

There are various modules in the Python Standard Library that deal with numeric and math-related functions and data types.

- The `math` module contains functions for floating-point numbers.
- The `decimal` module provides a data type that supports correctly-rounded decimal floating point arithmetic.
- The `random` module provides a pseudo-random number generator.

numbers_and_math.py

```
#!/usr/bin/env python3
import math
import decimal
import random


def math_examples():
    print("Square Root of 10:", math.sqrt(10))
    print("64 to 3/2 pow:", math.pow(64, 1.5))
    print("Hypotenuse of 6 and 8:", math.hypot(6, 8))
    print("Smallest integer >= 2.5", math.ceil(2.5))
    print("Pi", math.pi)


def decimal_examples():
    print(".1 + .2 is not normally thought of as", .1 + .2)
    d1, d2 = decimal.Decimal(".1"), decimal.Decimal(".2")
    print("It is normally:", d1 + d2)


def random_examples():
    data = [9, 8, 7, 6, 5, 4, 3, 2]
    print(" float 0.0 <= x < 1.0:", random.random())
    print(" int from 0 to 9:", random.randrange(10))
    print(" choice ", data, ":", random.choice(data))
    random.shuffle(data)
    print(" shuffled", ":", data)
    print(" sample ", data, ":", random.sample(data, 4))
```

An application that calls all three of the above methods is shown on the following page.

numbers_and_math_test.py

```
#!/usr/bin/env python3
from numbers_and_math import math_examples, decimal_examples, random_examples


def main():
    examples = [math_examples, decimal_examples, random_examples]
    for function in examples:
        print(function.__name__.upper(), "*" * 50)
        function()


if __name__ == "__main__":
    main()
```

The output of the above program is shown next.

```
$ python3 numbers_and_math_test.py
MATH_EXAMPLES ************************************************
Square Root of 10: 3.1622776601683795
64 to 3/2 pow: 512.0
Hypotenuse of 6 and 8: 10.0
Smallest integer >= 2.5 3
Pi 3.141592653589793
DECIMAL_EXAMPLES *********************************************
.1 + .2 is not normally thought of as 0.30000000000000004
It is normally: 0.3
RANDOM_EXAMPLES **********************************************
 float 0.0 <= x < 1.0: 0.9254105346291956
 int from 0 to 9: 2
 choice  [9, 8, 7, 6, 5, 4, 3, 2] : 9
 shuffled : [2, 3, 9, 4, 6, 5, 7, 8]
 sample  [2, 3, 9, 4, 6, 5, 7, 8] : [3, 6, 9, 2]
$
```

The `shuffle()` function of the `random` module updates the sequence passed to it in place as opposed to returning a new sequence of the results.

References

Many of the standard modules that deal with numbers and math can be found in the Python documentation at the following URL:

`https://docs.python.org/3/library/numeric.html`

Time and Date Modules

There are various modules in the Python Standard Library that deal with time and date functions and data types.

- We cover the `datetime`, `calendar` and `time` modules in this section.

The `datetime` module provides data types for manipulating both dates and times in both simple and complex ways.

The `datetime` modules defines many practical data types, some of which are listed here:

- `datetime.date` - represents a date (year, month and day).
- `datetime.time` – represents a local time of day.
- `datetime.datetime` – represents all the information from a `datetime.date` and a `datetime.time` object.
- `datetime.timedelta` - represents a duration between two dates or times.

References

The following URL provides a complete list of data types provided by the `datetime` module:

`https://docs.python.org/3/library/datetime.html`

The example on the following page demonstrates both the `datetime` and `timedelta` data types from the `datetime` module.

- The code demonstrates various ways of creating a `datetime` object through its constructor and other methods of the class.
- It also demonstrates the creation and use of a `timedelta` object to modify a `datetime` object.

dates_and_times.py

```
#!/usr/bin/env python3
from datetime import datetime, timedelta


def main():
    end = "\n\n"
    now = datetime.today()
    print("Today is:", now)
    print(now.month, now.day, now.year, sep="/", end=end)

    delta = timedelta(days=1, hours=12)
    future = now + delta
    print("36 hours from now:", future, end=end)

    past = datetime(2000, 1, 31, 14, 25, 59)
    print("A date in the past:", past, end=end)


if __name__ == "__main__":
    main()
```

The output of the above program is shown next.

```
$ python3 dates_and_times.py
Today is: 2018-09-24 16:50:47.224852
9/24/2018
36 hours from now: 2018-09-26 04:50:47.224852
A date in the past: 2000-01-31 14:25:59
$
```

The `calendar` module provides a several data types and functions related to calendars.

- By default, these calendars have Monday as the first day of the week, and Sunday as the last (the European convention).
- The `setfirstweekday()` function can be used to set the first day of the week to any other weekday.

The example on the next page demonstrates the various data types and functions in the `calendar` module.

calendars.py

```
#!/usr/bin/env python3
import calendar as cal     # Note the use of "import as"


def main():
    cal.setfirstweekday(cal.SUNDAY)
    datasets = [cal.day_name, cal.day_abbr, cal.month_name, cal.month_abbr]
    for calendar_data in datasets:
        print(list(calendar_data))

    print(cal.month(2016, 1))  # Calendar for January 2016
    cal.prmonth(2017, 3, w=5)  # Calendar for March 2017


if __name__ == "__main__":
    main()
```

The output of the above program is shown next.

```
$ python3 calendars.py
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
['', 'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August',
 'September', 'October', 'November', 'December']
['', 'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov',
 'Dec']
    January 2016
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31

               March 2017
 Sun    Mon    Tue    Wed    Thu    Fri    Sat
                      1      2      3      4
   5      6      7      8      9     10     11
  12     13     14     15     16     17     18
  19     20     21     22     23     24     25
  26     27     28     29     30     31

$
```

The `time` module contains various time-related functions and data types.

- Some of these functions refer to an epoch time, which is defined as hour zero on January 1st , 1970 for most systems.

Some of the available functions available in the `time` module are explained below.

- `time.time()` returns the time in seconds since the epoch as a floating point number.
- `time.ctime([secs])` returns a time expressed in seconds since the epoch as a string.
  - If the optional `secs` parameter is not passed as an argument it defaults to the current time.
- `time.sleep(secs)` suspends execution of the program for the given number of seconds.
  - The argument may be a floating point number to indicate a more precise sleep time.
  - The actual suspension time may be more or less than that requested because of a caught signal terminating the `sleep()` or the scheduling of other activities in the system.
- `time.perf_counter()` returns the value (in fractional seconds) of a performance counter (i.e. a clock with the highest available resolution to measure a short duration).
  - It includes time elapsed during sleep and is system-wide.
- `time.process_time()` returns the value (in fractional seconds) of the sum of the system and user CPU time of the current process.
  - It does not include time elapsed during sleep.
  - The reference point of the returned value for both functions is undefined, so that only the difference between the results of consecutive calls is valid.
- `time.strftime(format[,t])` converts a `struct_time` object `t`, or the current time if `t` is not provided, to a string as specified by the `format` argument.

References

The following URL provides a complete syntax for the `format` string:

`https://docs.python.org/3/library/time.html#time.strftime`

- `time.localtime([secs])` returns an object of type `time.struct_time`.
  - If the optional `secs` parameter is not passed as an argument it defaults to the current time.
- The `time.struct_time` object is an object with a named `tuple` interface.
  - The values can be accessed either by index or by attribute name.

The table on the following page lists the values that are available from the `struct_time` object.

struct_time Properties and Values

Index	Attribute Name	Values
0	tm_year	for example 2016
1	tm_mon	Range from 1 to 12
2	tm_mday	Range from 1 to 31
3	tm_hour	Range from 0 to 23

Index	Attribute Name	Values
4	tm_min	Range from 0 to 59
5	tm_sec	Range from 0 to 61
6	tm_wday	Range from 0 to 6 (Monday is 0)
7	tm_yday	Range from 1 to 366
8	tm_isdst	0 (no), 1 (yes), -1 (unknown)
N/A	tm_zone	abbreviation of time zone name
N/A	tm_gmtoff	offset east of UTC in seconds

The following program demonstrates the various functions from the `time` module.

time_testing.py

```
#!/usr/bin/env python3
import time


def counters():
    return(time.perf_counter(), time.process_time())


def main():
    starts = counters()
    print("time.time():", time.time())
    print("time.ctime():", time.ctime())
    print()
    now = time.localtime()
    f = "{0}/{1}/{2}"
    # struct_time values by property
    print("struct_time values by property and index")
    print(f.format(now.tm_mon, now.tm_mday, now.tm_year))

    # struct_time values by tuple index
    print(f.format(now[1], now[2], now[0]))
    print()
    # Formatting of struct_time objects
    print("time.strftime() examples:")
    print("Format: %m/%d/%Y\tResult: ",
          time.strftime("%m/%d/%Y", now))
    print("Format: %A %B %d\tResult: ",
          time.strftime("%A %B %d", now))
    print()
    time.sleep(5)
    ends = counters()
    print()
```

```
    print("Performance:", ends[0] - starts[0], "seconds")
    print("Process:", ends[1] - starts[1], "seconds")


if __name__ == "__main__":
    main()
```

The output of the above program is shown on the following page.

```
$ python3 time_testing.py
time.time(): 1575657424.6517038
time.ctime(): Fri Dec  6 13:37:04 2019

struct_time values by property and index
12/6/2019
12/6/2019

time.strftime() examples:
Format: %m/%d/%Y Result:  12/06/2019
Format: %A %B %d Result:  Friday December 06


Performance: 5.003588916002627 seconds
Process: 0.00015263399999999872 seconds
$
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `modules/solutions` directory.

Exercise 1

Define a few functions and place them in a module.

- Now, write a Python program in a separate file that imports the module and calls the functions.

Exercise 2

Create a new file and define a function in it with the same name (but different behavior) as one of the functions from the previous exercise.

- In a separate file, create an application that imports the module from this exercise that contains the function and the module from the previous exercise.
  - The application should be able to successfully call all of the functions from both of the imported modules.

Exercise 3

Write a program that sorts its command line arguments.

Exercise 4

Write a program that sums the command line arguments.

- The program should print both the sum of the arguments and the average value.

References

- The modules available from the Python standard library, and their documentation, can be found at the following URL:
  `https://docs.python.org/3/library/index.html`
- The PyPi repository can be found at the following URL:
  `https://pypi.python.org/pypi`
- The `sys` module's complete list of properties and functions and their documentation can be found at the following URL:
  `https://docs.python.org/3/library/sys.html`
- Many of the standard modules that deal with numbers and math can be found in the Python documentation at the following URL:
  `https://docs.python.org/3/library/numeric.html`
- The following URL provides a complete lst of data types provided by the `datetime` module:
  `https://docs.python.org/3/library/datetime.html`
- The following URL provides a complete syntax for the `format` string:
  `https://docs.python.org/3/library/time.html#time.strftime`

Summary

- Modules are files that you can use to organize and group your code.
- You can reuse the definitions stored in a module by importing the module in your application.
- You can develop your own modules, use the Python standard library modules, or use modules available in the Python Package Index.

Chapter 6

Classes in Python

Goal

Apply object oriented programming principles by using classes.

Objectives

- Understand the basic principles of an object oriented programming language.
- Use the `class` keyword to define custom data types.
- Use properties and decorators as a Pythonic way of writing classes.
- Use Python's special methods within a class definition to accomplish standard tasks.
- Use class variables as a way of providing shared access to instances of a class.
- Understand inheritance and polymorphism to take simplify class creation.
- Use various built-in functions to obtain information about the relationship between data types.

Sections

- Classes in Python

Classes in Python

Objectives

- Understand the basic principles of an object oriented programming language.
- Use the `class` keyword to define custom data types.
- Use properties and decorators as a Pythonic way of writing classes.
- Use Python's special methods within a class definition to accomplish standard tasks.
- Use class variables as a way of providing shared access to instances of a class.
- Understand inheritance and polymorphism to take simplify class creation.
- Use various built-in functions to obtain information about the relationship between data types.

Principles of Object Orientation

Object-oriented programming is a style of programming that lends itself to the principle that a software solution should closely model the problem domain.

- If your problem domain was banking, then some data types in your program might be `Customer`, `Account`, and `Loan`.
- If your problem domain was system software, then you might require types such as `Process` or `File`.

The aforementioned types would give rise to objects (entities) characterized by the following properties:

- Behavior: The actions the objects can perform.
- Attributes: The characteristics or properties of each object.

Object Orientation is characterized by the following principles:

- Encapsulation: The coupling of data and methods.
- Data abstraction or information hiding: Access to private data is achieved via a public interface (public accessor methods of the class).
- Inheritance: The derivation of specific data types from more general types. A typical object-oriented system contains inheritance hierarchies.
- Polymorphism: In an inheritance hierarchy, there may be a set of classes, each with the same named methods and interfaces, but whose implementations are different. Polymorphism is the ability of a language to differentiate these same named methods at run time.

Defining New Data Types

For domain specific problems, Python permits the defining of custom data types.

- These new data types are defined by using the `class` keyword.

Once a new data type has been defined, programmers can create instances of them.

- Each instance of a data type is called an object.

Suppose a new data type representing a student is desired.

- A minimal `Student` data type would be defined with a `class` definition such as the one below.

student0.py

```
#!/usr/bin/env python3
""" This module defines a Student datatype and a main function to demonstrate
    the instantiation of a Student object """


class Student:
    """ This class represents a Student. This mulitline string will act as
    a doc string since it is declared at the top of the class"""


def main():
    """ This main is basically for testing purposes only.
    The Student class would typically be imported by
    another module as opposed to running it here."""

    jeff = Student()     # Instantiate a Student object
    heather = Student()  # Instantiate another Student
    print(jeff, id(jeff), hex(id(jeff)))
    print(heather, id(heather), hex(id(heather)))


if __name__ == "__main__":
    main()
```

In the preceding example, the code is documented in three places:

- The module is being documented
- The class is being documented
- The main() function is being documented

The following example shows how you can get the documentation from the previous module by using the `help` function.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> help('student0')
```

The preceding execution of the `help` function shows the following information:

```
Help on module student0:

NAME
    student0

DESCRIPTION
    This module defines a Student data type and a main function to demonstrate
    the instantiation of a Student object

CLASSES
    builtins.object
        Student

    class Student(builtins.object)
     |  This class represents a Student. This multiline string will act as
     |  a doc string since it is declared at the top of the class
     |
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)

FUNCTIONS
    main()
        This main is basically for testing purposes only.
        The Student class would typically be imported by
        another module as opposed to running it here.
:
```

The execution output of the preceding program might be as follows:

```
$ python3 student0.py
<__main__.Student object at 0x7f613b7fca90> 140055586785936 0x7f613b7fca90
<__main__.Student object at 0x7f613b8027f0> 140055586809840 0x7f613b8027f0
$
```

Ultimately, the class definition will contain many methods, the collection of which defines the behavior for the class.

- Currently, the `Student` objects created in the example are skeletal because there are no custom properties or methods within them.
  - When designing software, there would typically be an analysis and design phase before any coding was undertaken.
  - In that phase, the behavior and the attributes of class objects would be determined by the designers.

`Student` attributes will consist of a name and a major.

- It would be convenient to pass these attributes as parameters during the creation of a `Student` object.

```
jeff = Student("Jeff", "American History")
heather = Student("Heather", "Mathematics")
```

When an object is created in Python, a special method called `__init__()` is called.

- Therefore, the `Student` class can include this special method.
  - In this case, two explicit arguments are passed to the method.
  - However, Python always also implicitly passes a reference to the object being constructed as the first parameter to the method.
- The declaration of the `__init__()` method might begin like the following:

```
def __init__(self, name, major):
    pass # Remember to put real code here
```

- The `pass` statement does nothing, but is used here to satisfy the syntax requirement for a method.

We still need to add some code to fill the object with its data.

student1.py

```
#!/usr/bin/env python3
class Student:

    def __init__(self, name, major):
        self.name = name
        self.major = major


def main():
    jeff = Student("Jeff", "American History")
    heather = Student("Heather", "Mathematics")
    print(jeff.name, ":", jeff.major)
    print(heather.name, ":", heather.major)
    jeff.name = "Jeffrey"
    heather.major = "Computer Science"
    print(jeff.name, ":", jeff.major)
    print(heather.name, ":", heather.major)


if __name__ == "__main__":
    main()
```

- The first parameter in the `__init__()` method is called `self`.
  - This is a reference to the object being created.
  - While this variable could technically be called anything, traditionally it is called `self`.

- The `__init__()` method can be seen as a constructor of the object.
  - Its typical role is to copy the parameters to the object's data.
  - `self.name` and `self.major` are used to store the local variables `name` and `major` as instance data of the object.
- The application accesses the properties of the object directly in the `main` function.
  - The `print` statements access the value of the properties directly.
  - `jeff.name = "Jeffrey"` shows how to set properties directly.

Typically, a class would have methods to retrieve and modify the instance data.

- Many object-oriented languages emphasize data hiding, where direct access to the data is private.
  - Public access to the variables is then typically provided in the form of methods typically referred to as *getters* and *setters* for the data.
  - While Python has no concept of private, any symbol that starts with a single underscore is understood to be private by convention only.
- The following example shows an updated version of the `Student` class that includes instance methods that have been added to the class.

student2.py

```
#!/usr/bin/env python3
class Student:
    def __init__(self, name, major):
        self._name = name
        self._major = major

    def get_name(self):
        return self._name

    def set_name(self, name):
        self._name = name

    def get_major(self):
        return self._major

    def set_major(self, major):
        self._major = major


def main():
    jeff = Student("Jeff", "American History")
    print(jeff.get_name(), ":", jeff.get_major())
    jeff.set_name("Jeffrey")
    print(jeff.get_name(), ":", jeff.get_major())
```

```
if __name__ == "__main__":
    main()
```

The @property Decorator

The preceding example uses *getter* and *setter* methods. This is a common convention in many object-oriented languages to manage how class attributes are accessed and mutated.

In Python, however, the recommended, Pythonic, way to implement a *getter/setter* mechanism is to use the `@property` decorator to wrap class attributes.

Decorators

Python decorators are a built-in feature of the language and extend the behavior of functions, classes, and methods.

- Decorators wrap existing functions, or classes, to extend their original functionality.
- You can define decorators as functions or classes that take another function, or class, extend its behavior, and return the decorated function/class.
- After creating a decorator, you can use it to decorate functions or classes by using the name of the decorator, preceded by the @ symbol, just before the function/class definition.

Using the `@property` Decorator

Although you can create your own decorators, Python comes with a number of built-in decorators, such as the `@property` decorator.

- A class property is a method that can be accessed as an attribute. The value of the attribute is the return value of the method.
- The following example demonstrates the use of the built-in `@property` decorator to define *getter* methods for the `_name` and `_major` attributes of a student.
- The `_name` and `_major` attributes are not meant to be accessed from outside the class.
- Using the `@property` decorator creates a new property. The property name is the name of the method that `@property` decorates. In the following example, the `def name(self)` method signature defines a new property called `name`.
- The implementation of the `@property`-decorated method is the *getter* of the property. In the example, the `name` property *getter* returns the value of `self._name`.
- Properties include a `setter` attribute that you can use to define a *setter* method.
- To define a *setter* method on a property, use the `@property-name.setter` decorator. For a *setter* method to be valid, you must use the same method name as in the *getter* method. The following example demonstrates defining *setters* by using the `@name.setter` and `@major.setter` decorators.

student3.py

```
#!/usr/bin/env python3
class Student:
```

```
    def __init__(self, name, major):
        self.name = name    # Calls the @name.setter method
        self.major = major  # Calls the @major.setter method

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        # print("Debug:", "name setter being called")
        self._name = name

    @property
    def major(self):
        return self._major

    @major.setter
    def major(self, major):
        # print("Debug:", "major setter being called")
        self._major = major
```

The following example shows how to use the preceding module.

student3_test.py

```
#!/usr/bin/env python3
from student3 import Student


def main():
    jeff = Student("Jeff", "American History")
    heather = Student("Heather", "Mathematics")
    print(jeff.name, ":", jeff.major)
    jeff.name = "Jeffrey"
    print(jeff.name, ":", jeff.major)
    heather.major = "Computer Science"
    print(heather.name, ":", heather.major)


if __name__ == "__main__":
    main()
```

- The following snippet is the output of the preceding example.

```
$ python3 student3_test.py
Jeff : American History
Jeffrey : American History
Heather : Computer Science
$
```

When you access a property as an attribute, e.g. `jeff.name`, you trigger the *getter* method of the property. Similarly, when you mutate a property, e.g. `jeff.name = "Jeff"`, you trigger the *setter* method of the property.

- The added benefit is that if any business logic or validation of the properties needs to be performed on the properties, there are existing methods where the validation can be done.
- More information can be found by reading the documentation for the property class by typing `help(property)` in an interactive Python shell.

Special Methods

In addition to the `__init__()` method, there are other special methods that are commonly defined within a class.

- One such method is called `__str__()`.
  - This method is automatically invoked by the `str(object)` constructor and the built-in functions `format()` and `print()` to compute the nicely printable string representation of an object.
  - The return value must be a string object.
- Another special method is called `__del__()`.
  - Called when the instance is about to be destroyed.
  - The `del` statement `del x` doesn't directly call the `x.__del__()` method. The former decrements the reference count for `x` by one, and the latter is only called when `x`'s reference count reaches zero.
  - It is not guaranteed that `__del__()` methods are called for objects that still exist when the interpreter exits.

A class can also implement certain operations that are invoked by special syntax.

- This is done by defining methods with special names.
- This is Python's approach to operator overloading.
  - For instance, if a class defines a method called `__eq__()`, and `x` is an instance of this class, then `x == someobject` invokes a call to the `x.__eq__()` method of the class and passes `someobject` as a parameter.
  - Typically, any attempt to execute an operation will raise an `AttributeError` or `TypeError` exception when no corresponding method has been defined.

References

The list of special methods that can be defined inside a class can be found in the documentation here:

`http://docs.python.org/3/reference/datamodel.html#special-method-names`

- The following example updates the preceding version of the `Student` class to include several special methods.

student4.py

```
#!/usr/bin/env python3
class Student:

    def __init__(self, name, major):
        self.name = name    # Calls the @name.setter method
        self.major = major  # Calls the @major.setter method

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name

    @property
    def major(self):
        return self._major

    @major.setter
    def major(self, major):
        self._major = major

    def __str__(self):
        return "{} : {}".format(self.name, self.major)

    def __eq__(self, obj):
        if type(obj) != Student:
            return False
        else:
            return self.name == obj.name and \
                   self.major == obj.major
```

- The following example is an application that uses the preceding `Student` class.

student_test.py

```
#!/usr/bin/env python3
from student4 import Student


def main():
    s1 = Student("Elizabeth", "Electrical Engineering")
    s2 = Student("Robert", "Electrical Engineering")
    student_info("Before", s1, s2)
    s2.name = "Elizabeth"
    student_info("After", s1, s2)


def student_info(label, student1, student2):
    print(label)
    print(student1, "id=", id(student1))
```

```
    print(student2, "id=", id(student2))
    fmt = "{0} == {1} : {2}"
    print(fmt.format(student1, student2, student1 == student2))
    print()


if __name__ == "__main__":
    main()
```

- The execution output of the preceding application might be:

```
$ python3 student_test.py
Before
Elizabeth : Electrical Engineering id= 139687194554720
Robert : Electrical Engineering id= 139687194554776
Elizabeth : Electrical Engineering == Robert : Electrical Engineering : False

After
Elizabeth : Electrical Engineering id= 139687194554720
Elizabeth : Electrical Engineering id= 139687194554776
Elizabeth : Electrical Engineering == Elizabeth : Electrical Engineering: True
$
```

The following example defines a class called `Fraction`.

- Although Python already provides a class called `Fraction` in a module called `fractions`, a simplified version of the representation of a fraction is being presented here to show another example of creating a custom data type.
- A `Fraction` will be composed of an integer for the numerator and an integer for the denominator.
- The `Fraction` class also defines several special methods.

fraction.py

```
#!/usr/bin/env python3
class Fraction:
    def __init__(self, numerator=0, denominator=1):
        self.numerator = numerator
        self.denominator = denominator

    def __str__(self):
        return "{}/{}".format(self.numerator, self.denominator)

    def __lt__(self, other):
        left = self.numerator / self.denominator
        right = other.numerator / other.denominator
        return left < right

    def __mul__(self, other):
```

```
        numerator = self.numerator * other.numerator
        denominator = self.denominator * other.denominator
        return Fraction(numerator, denominator)
```

An application that uses the preceding `Fraction` class, and the output of the application are as follows:

fraction_tests.py

```
#!/usr/bin/env python3
from fraction import Fraction


def main():
    fractions = [Fraction(1, 3), Fraction(), Fraction(3, 4)]
    for fraction in fractions:
        print(fraction)

    frac_1_3, frac_0_1, frac_3_4 = fractions
    print(frac_1_3, "<", frac_0_1, ":", frac_1_3 < frac_0_1)  # False
    print(frac_1_3, "<", frac_3_4, ":", frac_1_3 < frac_3_4)  # True
    result = frac_1_3 * frac_3_4
    print(frac_1_3, "*", frac_3_4, "=", result)  # 1/3 * 3/4 = 3/12


if __name__ == "__main__":
    main()
```

```
$ python3 fraction_tests.py
1/3
0/1
3/4
1/3 < 0/1 : False
1/3 < 3/4 : True
1/3 * 3/4 = 3/12
$
```

Class Variables

Class data is a part of the class but not a part of an object.

- Instance variables are typically for data unique to each instance, while class variables are for attributes and methods shared by all instances of the class.

The following example defines a class variable called `quantity`.

car.py

```
#!/usr/bin/env python3
class Car:
```

```
    quantity = 0  # Class variable shared by all instances

    def __init__(self, make, model):
        self.make = make
        self.model = model
        self.odometer = 0
        Car.quantity += 1

    def __del__(self):
        Car.quantity -= 1

    def drive(self, miles):
        self.odometer += miles

    def __str__(self):
        data = [self.make, self.model, " ~ Odometer:", str(self.odometer)]
        return " ".join(data)
```

The following application shows the differences between accessing instance variables and class variables from the `Car` class.

car_test.py

```
#!/usr/bin/env python3
from car import Car


def main():
    malibu = Car("Chevy", "Malibu")
    miata = Car("Mazda", "Miata")
    mustang = Car("Ford", "Mustang")
    soul = Car("Kia", "Soul")
    print("# of existing Cars:", Car.quantity)

    malibu.drive(miles=10000)
    miata.drive(500)
    print(malibu, miata, mustang, soul, sep="\n")
    print("Deleting Malibu")
    del malibu

    print("# of existing Cars:", Car.quantity)


if __name__ == "__main__":
    main()
```

- Note the difference in the syntax between accessing instance variables and accessing class variable.
- Each object of type `Car` has its own `make`, `model`, and `odometer` properties.
- All `Car` objects have access to the single `quantity` variable that exists in memory.

Inheritance

Inheritance is one of the signature characteristics of object-oriented programming.

- In designing a set of classes that is important to a set of applications, there will usually be some classes that are related to one another.
- One kind of relationship, known as inheritance, is described by the words "is a".
  - A `Manager` is an `Employee`.
  - A `Directory` is a `File`.
  - A `Square` is a `Shape`.
  - An `Array` is a `Data Structure`.

The main idea behind inheritance is that the data and methods of the superclass (the base class) are directly reused by a subclass (the derived class).

- For example, a `Directory` object could directly reuse `File` methods since a `Directory` is a `File`.
- The subclass may add newly created methods to implement additional behavior.

The syntax for defining the inheritance is as follows:

```
class SubClassName(SuperClassName)
```

- The following example demonstrates inheritance by defining a subclass called `GradStudent` class that has the `Student` class as its superclass.
- The `GradStudent` is a `Student` that receives a monetary stipend.
- The inheritance is implemented with the following syntax:

```
class GradStudent(Student):
```

- The `GradStudent` class will inherit all the attributes and methods of `Student`.

gradstudent.py

```
#!/usr/bin/env python3
from student4 import Student


class GradStudent(Student):

    def __init__(self, name, major, stipend):
        # Pass the name and major to the __init__() of the
        # super class then store the part specific to the
        # GradStudent object
        super().__init__(name, major)
        self.stipend = stipend

    @property
    def stipend(self):
```

```
        return self._stipend

    @stipend.setter
    def stipend(self, stipend):
        self._stipend = stipend

    def __str__(self):
        # Override the __str__ from the parent class by
        # first getting the parent information:
        # super().__str__()
        # and then concatenating the stipend
        return "{} {}".format(super().__str__(), self.stipend)
```

The `__init__()` method of the `GradStudent` class uses `super()` to pass information to the `__init__()` method of the `Student` class.

- `super()` returns an object that acts as a proxy, and delegates method calls to a parent or sibling class.
- This is useful for accessing inherited methods that have been overridden in a class.

The `__str__()` method of the `GradStudent` class overrides the `__str__()` method of the `Student` class.

- The method reuses the parent classes `__str__()` method with a call to `super()` and incorporates it into its own.

gradstudent_test.py

```
#!/usr/bin/env python3
from gradstudent import GradStudent


def main():
    grad_student = GradStudent("James", "Anthropology", 25000)

    print("  MAJOR:", grad_student.major)
    print("   NAME:", grad_student.name)
    print("STIPEND:", grad_student.stipend)
    print(grad_student)


if __name__ == "__main__":
    main()
```

- The output of the preceding program is as follows:

```
$ python3 gradstudent_test.py
  MAJOR: Anthropology
   NAME: James
STIPEND: 25000
James : Anthropology 25000
$
```

Polymorphism

The word polymorphism literally means "many forms".

- When the word is used in object-oriented programming languages, it means the ability of the language to execute a method based on the run time type of a variable.

For example, suppose we have a `Shape` class with derived classes called `Square`, `Rectangle` and `Circle`.

- It would be simple enough to create objects of these subclasses.

```
s1 = Square()
c1 = Circle()
r1 = Rectangle()
```

- Suppose further that each of the subclasses has an `area()` method.
- Each subclass would need to have its own implementation of the `area()` method since it is dependent upon the specific subclass of `Shape`.
- A collection of various shapes can then be created in a list.

```
shapes = [ s1, c1, r1 ]
```

- Iterating over the list of shapes to calculate the area of each could then be done as follows.

```
for shape in shapes:
    print(shape.area())
```

- Polymorphism ensures that it is the runtime type of the object whose method will be called.
- Python knows the run time type of each `Shape` object and calls the appropriate `area()` method.
- The entire code to implement the preceding example is as follows:

shape.py

```
#!/usr/bin/env python3
class Shape:
    id = 100

    def __init__(self, name):
        self.name = name
        self.number = Shape.id
        Shape.id += 1

    def area(self):
        pass  # Intended to be implemented by subclasses

    @property
    def name(self): return self._name

    @name.setter
```

```
    def name(self, name): self._name = name

    def __str__(self):
        return "Name:{}  id:{}".format(self.name, self.number)
```

shape_circle.py

```
#!/usr/bin/env python3
import math
from shape import Shape


class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def __str__(self):
        fmt = "{}  Radius:{}"
        return fmt.format(super().__str__(), self.radius)

    def area(self):
        return math.pi * self.radius ** 2
```

shape_rectangle.py

```
#!/usr/bin/env python3
from shape import Shape


class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def __str__(self):
        fmt = "{}  Length:{} Width:{}"
        return fmt.format(super().__str__(), self.length,
                          self.width)

    def area(self):
        return self.length * self.width
```

shape_square.py

```
#!/usr/bin/env python3
from shape_rectangle import Rectangle


class Square(Rectangle):
    def __init__(self, name, length):
        super().__init__(name, length, length)
```

In inheritance hierarchies like the `Shape` hierarchy, one can conceive of two kinds of methods.

- Those with behavior invariant over specialization.
  - The getter and setter methods for the `name` attribute are such methods and, therefore, there is no need to override them in the subclasses.
- Those with behavior that varies over specialization.
  - The `area()` method is such a method and, therefore, it is overridden in the subclasses.

The following application tests all the classes in the `Shape` hierarchy.

shape_testing.py

```
#!/usr/bin/env python3
from shape_circle import Circle
from shape_square import Square
from shape_rectangle import Rectangle


def main():
    shapes = [Circle("Circle 1", 10),
              Square("Square 1", 5),
              Rectangle("Rectang1e 1", 5, 10)]

    for shape in shapes:
        print(shape)
        print("AREA:", shape.area())
        print("*" * 50)


if __name__ == "__main__":
    main()
```

- The output of this program might be as follows:

```
$ python3 shape_testing.py
Name:Circle 1 id:100 Radius:10
AREA: 314.1592653589793
**************************************************
Name:Square 1 id:101 Length:5 Width:5
AREA: 25
**************************************************
```

```
Name:Rectang1e 1 id:102 Length:5 Width:10
AREA: 50
************************************************
$
```

Type Identification

The `type` function, which identifies the type of a variable has already been used in examples several times.

- On some occasions, there is the requirement to determine whether an object is of one class or another.
- The `isinstance()` function determines whether a particular object is of a particular class.
- The `issubclass()` function determines whether one class is a subclass (direct or otherwise) of another class.

classes.py

```
#!/usr/bin/env python3
class Employee:
    pass


class Manager(Employee):
    pass


class Executive(Manager):
    pass


def main():
    manager = Manager()

    print(isinstance(manager, Employee))      # True
    print(isinstance(manager, Manager))       # True
    print(isinstance(manager, Executive))     # False

    print(issubclass(Executive, Executive))   # True
    print(issubclass(Executive, Manager))     # True
    print(issubclass(Executive, Employee))    # True
    print(issubclass(Executive, object))      # True


if __name__ == "__main__":
    main()
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `classes/solutions` directory.

Exercise 1

Create a class called `Person`.

- Each `Person` should have a `name`, an `age`, and a `gender`.
- In addition to getters and setters for the above methods, the `Person` class must have an `__init__()` method and a `__str__()` method.
- The `__init__()` and `__str__()` methods must be defined such that the following can be tested inside of an application.

```
p1 = Person("Michael", 45, "M")
print(p1)
```

Exercise 2

Create a class called `Family`.

- The `Family` does not extend `Person` but rather must be composed of two `Person` objects representing the parents and a `list` of `Person` objects representing the children.
- Therefore, the `__init__()` method must take two required parameters (the parents), followed by a variable number of arguments (the children).
- The following files are available in the `AD141-apps` repository, within the `classes/starter` directory, and you can use them in the exercise:

family_test.py

```
#!/usr/bin/env python3


def main():
    mother = Person("Mom", 45, "F")
    father = Person("Dad", 45, "M")
    kid1 = Person("Johnie", 2, "M")
    kid2 = Person("Janie", 3, "F")
    myFamily = Family(mother, father, kid1, kid2)
    kid3 = Person("Paulie", 1, "M")
    myFamily.add(kid3)
    print(myFamily)


if __name__ == "__main__":
    main()
```

- Note the `add` method in the `Family` class.

Exercise 3

Implement the necessary special methods so that the <, ==, and > operators can be used with `Family` objects.

- The criteria for the methods should be the number of children.

- The following code could be used to test the methods:

```
myFamily = Family(mom, dad, kid1, kid2)
smiths = Family(mom, dad, kid1)
if (myFamily > smiths):
    print("we have more kids than smiths")
if (myFamily == smiths):
    print("families have same # of kids")
if (myFamily < smiths):
    print("we have fewer kids than smiths")
```

Exercise 4

Implement the following class hierarchy.

- Define a `Worker` class with a `name`, a `salary`, and number of `years` worked.
  - Provide a method called `pension` that returns an amount equal to the years worked times 10% of the salary.
  - Implement a `name()` method in the `Worker` class and have this be a default method for all derived classes.
- Derive `Manager` from `Worker`.
  - A manager's pension is defined by the number of years worked times 20% of the salary.
- Derive `Executive` from `Manager`.
  - An executive's pension is defined by the number of years worked times 30% of the salary.

References

The list of special methods that can be defined inside a class can be found in the documentation here:

`http://docs.python.org/3/reference/datamodel.html#special-method-names`

Summary

- You can build software solutions that are closer to the problem domain by using object-oriented programming.
- By using class variables, you can share data among the instances of a class.
- You can enhance the reusability of your code by using inheritance and polymorphism.

Chapter 7

Exceptions

Goal

Create, handle, and throw exceptions to control errors.

Objectives

- Understand and use the Python exception model.
- Use `try` and `except` as the basic exception handling clauses in Python.
- Understand and use various exceptions in the exception hierarchy.
- Raise exceptions within your code as indicators of errors happening when executing the code.
- Create and use user-defined exceptions within your code.
- Understand the `assert` keyword and its benefits when writing your code.

Sections

- Exceptions

Exceptions

Objectives

- Understand and use the Python exception model.
- Use `try` and `except` as the basic exception handling clauses in Python.
- Understand and use various exceptions in the exception hierarchy.
- Raise exceptions within your code as indicators of errors happening when executing the code.
- Create and use user-defined exceptions within your code.
- Understand the `assert` keyword and its benefits when writing your code.

Errors and Exceptions

Python has two main kinds of errors: syntax errors and exceptions.

- Syntax errors are caught by the parser as the script is being interpreted.
- Python displays the file name and line number, so you know where to look to find the error.
- Even if the program is free of syntax errors, things can still go wrong during the execution of the program.
- Errors detected during execution are called exceptions.
- When your code raises an exception, if the exception is not handled, then the program terminates.
- You can handle exceptions to recover from errors and allow the program to continue its execution.

For example, consider the following program, which expects a series of integers, and outputs the sum of those integers.

totals.py

```
#!/usr/bin/env python3
def main():
    total = 0
    msg = "Please enter a number, or 'end' to quit: "
    while True:
        value = input(msg)
        if value == "end":
            break
        total += int(value)

    print("Total is", total)
```

```
if __name__ == "__main__":
    main()
```

If the user enters non-integer data (other than the `end` string), then the program raises an exception and the program terminates, as shown in the following execution output:

```
$ python3 totals.py
Please enter a number, or 'end' to quit: 3
Please enter a number, or 'end' to quit: 5
Please enter a number, or 'end' to quit: 7
Please enter a number, or 'end' to quit: one
Traceback (most recent call last):
  File "totals.py", line 15, in <module>
    main()
  File "totals.py", line 9, in main
    total += int(value)
ValueError: invalid literal for int() with base 10: 'one'
$
```

- When the user enters a value of `one` as the input, the call to `int()` raises a `ValueError` exception.
- The default Python runtime response to an exception is to print a stack trace (or stack traceback) and terminate the program.
- The stack trace includes information about the file, line number, and method or function the exception occurred in.
- The last line of the error message indicates what happened.
- The actual details are based on the exception type.
- The following output shows what happens when the end-of-transmission character (`Ctrl+D`) is typed at the input prompt.

```
$ python3 totals.py
Please enter a number, or 'end' to quit: Traceback (most recent call
last)
   File "totals.py", line 15, in <module>
     main()
   File "totals.py", line 6, in main
    value = input(msg)
EOFError
$
```

- The resulting exception raised in the preceding program is an `EOFError` exception.
- If developers wish to respond in their own way, then exception handling must be added to the program.
- To handle error scenarios, and prevent your program from crashing, you must use exception handling.

The Exception Model

In Python, the exception model consists of the following.

- A `try` statement is used to surround code that may generate one or more exceptions.
- `try` statements can specify the following:
  - One or more `except` clauses that serve as exception handlers.
  - An `else` clause that only runs if no exception occurs.
  - A `finally` clause that runs whether an exception is raised or not.
- A `try` statement cannot stand by itself.
  - It must be followed by an `except` clause or a `finally` clause.
  - If followed by an `except` clause, it may then optionally define additional `except` clauses, an `else` clause, and/or a `finally` clause.

The following example is a rewrite of the previous application. This version incorporates Python's exception handling model.

totals_handled.py

```
#!/usr/bin/env python3
def main():
    total = 0
    while True:
        value = input("Please enter a number: ")
        if value == "end":
            break
        try:
            total += int(value)
        except ValueError:
            print("Invalid Number - Please try again")

    print("Total is", total)


if __name__ == "__main__":
    main()
```

- Now, when an illegal value is entered, the program handles the exception, after which the program is able to continue.

Exception Handling

The combination of `try` and `except` clauses can occur in various parts of your program.

Exceptions are instances of the `Exception` class, so you can treat them as any other object.

- To get the reference of an exception object in an `except` clause, use the `as` keyword, as the following example demonstrates:

handled_separately.py

```
#!/usr/bin/env python3
def main():
    total = 0
    while True:
        try:
            value = input("Please enter a number: ")
            if value == "end":
                break
        except EOFError:
            print('Unexpected End of Stream')
            continue
        try:
            total += int(value)
        except ValueError as ve:
            print("Exception: ", ve)
        finally:
            print("Running subtotal is:", total)

    print("Total is", total)


if __name__ == "__main__":
    main()
```

In the preceding example, the `except ValueError as ve` clause gets the reference of the `ValueError` exception in the `ve` variable, to use it in the `print()` statement.

Also, Python always executes the `finally` clause, whether the call to `int()` throws an exception or not.

Exception Hierarchy

- The hierarchy of Python built-in exceptions is as follows.

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
     +-- StopIteration, StopAsyncIteration, AssertionError, AttributeError
     +-- BufferError, EOFError, ImportError, MemoryError, ReferenceError
     +-- SystemError, TypeError
     +-- ArithmeticError
     |    +-- FloatingPointError, OverflowError, ZeroDivisionError
     +-- LookupError
     |    +-- IndexError, KeyError
     +-- NameError
     |    +-- UnboundLocalError
     +-- OSError
     |    +-- BlockingIOError, ChildProcessError, FileExistsError
     |    +-- FileNotFoundError, InterruptedError, IsADirectoryError
```

```
      |    +-- NotADirectoryError, PermissionError, ProcessLookupError
      |    +-- TimeoutError
      |    +-- ConnectionError
      |    | +-- BrokenPipeError, ConnectionAbortedError
      |    | +-- ConnectionRefusedError, ConnectionResetError
      +-- RuntimeError
      |   +-- NotImplementedError, RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError, UnicodeEncodeError, UnicodeTranslateError
      +-- Warning
           +-- DeprecationWarning, PendingDeprecationWarning, RuntimeWarning
           +-- SyntaxWarning, UserWarning, FutureWarning, ImportWarning
           +-- UnicodeWarning, BytesWarning, ResourceWarning
```

As seen from the previous list of exceptions, there are many different exception types in Python.

- You can handle multiple exception types in a single `except` block by handling the parent class type.

multi.py

```
#!/usr/bin/env python3
def main():
    names = ['Mike', 'John',  'Jane',  'Alice']
    themap = {'Mike': 15, 'Chris': 10, 'Dave': 25}

    while True:
        try:
            value = input("Enter an integer: ")
            if value == "end":
                break
            value = int(value)
            print("Name is: " + names[value])
            name = input("Enter a name: ")
            print(name, " => ", themap[name])
        except ValueError:
            print("Value Error: non numeric data")
        except (KeyError, IndexError) as err:
            # Above could be written as:
            #      except LookupError as err:
            print("Illegal value:", err)
        except Exception:
            print("Unknown Exception: ")


if __name__ == "__main__":
    main()
```

Raising Exceptions

Python provides the `raise` statement, allowing the programmer to force a specified exception to occur.

- Several basic examples of this are demonstrated below in an interactive shell.
- Most of the examples use the `raise` keyword followed by an instance of an exception object.
- The last example simply uses the `raise` keyword by itself to reraise the exception after handling it.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> raise Exception()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception
>>>
>>> raise Exception("There was a problem")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
Exception: There was a problem
>>>
>>> raise ValueError("Bad value 'one'")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Bad value 'one'
>>>
>>> try:
...     int("one")
... except ValueError:
...     print("Not a Number")
...     raise
...
Not a Number
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: invalid literal for int() with base 10: 'one'
>>> exit()
$
```

- The following example is a rewrite of the custom `Fraction` class defined previously.
- It incorporates raising a built-in exception for a denominator of zero.

fraction.py

```
#!/usr/bin/env python3
class Fraction:
    def __init__(self, numerator=0, denominator=1):
        self.numerator = numerator
        self.denominator = denominator
```

```
    @property
    def denominator(self):
        return self._denominator

    @denominator.setter
    def denominator(self, denominator):
        if denominator == 0:
            raise ZeroDivisionError()
        self._denominator = denominator

    def __str__(self):
        return "{}/{}".format(self.numerator, self.denominator)
```

fraction_test.py

```
#!/usr/bin/env python3
from fraction import Fraction


def main():
    try:
        while True:
            numer = int(input("Please enter a numerator"))
            denom = int(input("Please enter a denominator"))
            fraction = Fraction(numer, denom)
            print(fraction)
    except ZeroDivisionError:
        print("Zero Division Error")


if __name__ == "__main__":
    main()
```

- When an appropriate built-in exception does not exist to sufficiently convey the nature of a problem, you can define your own exceptions by creating a new exception class.

User-Defined Exceptions

Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

- The base class for built-in exceptions is the `BaseException` class.
- User-defined classes should not inherit from `BaseException`, but from the `Exception` class.
- User-defined exceptions can do anything that any other class can do, but are usually kept simple.

The following example defines several user-defined exceptions to represent the various forms of a password that are not acceptable.

- The first exception is designed to be the base class for all exceptions that pertain to passwords.
  - All other exceptions will extend the defined base class.

- The exceptions defined within the module can then be used by any application that deals with checking to see if a password meets given criteria to be considered acceptable.
  - The exceptions classes themselves do not actually do any of the checking.
  - They are simply defined to represent the various types of potential problem passwords.

password_errors.py

```
#!/usr/bin/env python3
class PasswordError(Exception):
    """Base class for exceptions in this module."""
    pass


class TrivialPasswordError(PasswordError):
    """Passwords that are too Trivial like: 'password'"""
    def __init__(self, msg):
        super().__init__("Trivial Password:" + msg)


class PasswordLengthError(PasswordError):
    """Passwords that do not meet certain length criteria"""
    def __init__(self, msg, length):
        super().__init__(msg)
        self.length = length

    def get_length(self):
        return self.length


class RepetetiveError(PasswordError):
    """Passwords that have repetitive characters"""
    def __init__(self, msg):
        super().__init__(msg)
```

With the above hierarchy of `PasswordError` exceptions available, an application can now be written to raise any of them as needed when validating the syntax of a password.

- The following example presents a utilities module with several functions that check the validity of a password.
- This is done so that many different applications can utilize the behavior defined within the module.
- This module defines the various criteria of what constitutes a valid password.
- Each function defined in the module simply raises a particular subclass of `PasswordError` if an error is detected.

password_utilities.py

```
#!/usr/bin/env python3
import password_errors
```

```
def check_trivial(password):
    bad = ["password", "p@ssword", "passw0rd", "p@ssw0rd"]
    if password.lower() in bad:
        raise password_errors.TrivialPasswordError(password)


def check_length(password):
    min_length = 10
    length = len(password)
    if length < min_length:
        raise password_errors.PasswordLengthError("Too short", length)


def check_duplicates(password):
    removedupes = set(password)
    if len(removedupes) < len(password):
        raise password_errors.RepetetiveError("Repetetive Characters Exist")
```

The following example shows how you can you use the password checks and the exceptions in an application.

password_tests.py

```
#!/usr/bin/env python3
from password_errors import PasswordError
from password_utilities import check_trivial, check_length, check_duplicates


def check_password(password):
    check_trivial(password)
    check_length(password)
    check_duplicates(password)


def main():
    while True:
        try:
            line = input("Please enter a password")
            check_password(line)
            print("That would be a valid password")
        except PasswordError as pe:
            print("Password Error: ", pe)


if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print()
        print("Terminating Program")
    except Exception as e:
        print("Unknown Issue:", e)
```

The assert statement

An assertion is a way to check that the internal state of a program is as the programmer expected.

- Assertions are expressions that evaluate to either `True` or `False`.

`assert` statements allow a way to insert debugging assertions.

- If an `assert` statement evaluates to `False`, then an `AssertionError` exception is raised.
- If the `assert` statement evaluates to `True`, control flow passes to the next statement.

The general syntax for an *assert statement* is as follows.

```
assert expression1["," message]
```

- The required `expression1` is the part of the statement that should evaluate to `True` or `False`.
- The optional `, message` is a message to include in the `AssertionError` that might get generated.

assert_demo.py

```
#!/usr/bin/env python3
def findmax(a, b):
    max = 0
    if a < b:
        max = b
    elif a > b:
        max = a

    fmt = "Max is not {} or {}"
    assert max == a or max == b, fmt.format(a, b)
    return max


def main():
    try:
        print(findmax(2, 9), findmax(7, 4))
        print(findmax(3, 3))
    except AssertionError as ae:
        print("Assertion Failed:", ae)


if __name__ == "__main__":
    main()
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `exceptions/solutions` directory.

Exercise 1

Create a list in your program that has 10 numbers.

- Then, in a loop, ask the user for a number.
- Use this number as an index into your list and print the value located at that index.
- End the program when the user enters `end`.
- Handle the case of an illegal number.
- Handle the case of an illegal subscript.

Exercise 2

Test Exercise 1 again by using a few negative numbers as the index.

- Eliminate negative numbers as legitimate subscripts by raising the `IndexError` exception when a negative number is given.

Exercise 3

Write a program that uses a loop to prompt the user and get an integer value.

- The program should print the sum of all the integers entered.
- If the user enters a blank line or any other line that cannot be converted to an integer, the program should handle this `ValueError`.
- If the user uses `Ctrl+C` to terminate the program, then it should be trapped with a `KeyboardInterrupt`, and a suitable message should be printed.
- When the user enters the end of file character (`Ctrl+D` on Linux or `Ctrl+Z` on Windows), the program should trap this with the `EOFError`, break out of the loop, and print the sum of all the integers.

Summary

- You can handle exceptions to avoid interrupting the program control flow.
- You can create your own exceptions to express a different kind of error.
- You can throw exceptions to signal errors that must be handled at a different level.

Chapter 8

Input and Output

Goal

Read and write sequences of bytes in input and output data streams.

Objectives

- Use Python's additional I/O capabilities beyond the `input` and `print` functions.
- Create and use data streams to read and write to files.
- Read and write to text files.
- Use `bytes` and `bytearray` data types to read and write binary files.
- Use the `seek` and `tell` methods to randomly access the stream contents.
- Use the `os` and `os.path` modules to work with files and directories.

Sections

- Input and Output

Input and Output

Objectives

- Use Python's additional I/O capabilities beyond the `input` and `print` functions.
- Create and use data streams to read and write to files.
- Read and write to text files.
- Use `bytes` and `bytearray` data types to read and write binary files.
- Use the `seek` and `tell` methods to randomly access the stream contents.
- Use the `os` and `os.path` modules to work with files and directories.

Introduction

Up to this point in the course, the discussion of input and output has been limited to the `input()` and `print()` functions.

- This chapter introduces additional ways in which Python programs can read and write to and from various sources, other than the standard input and output files.

An input data stream is an object that provides data to the application as a sequence of bytes.

- `sys.stdin` is the data stream that represents the operating system's standard input device - usually the keyboard.

An output data stream is an object used by an application for writing data out as a sequence of bytes.

- `sys.stdout` is the data stream that represents the standard output device - usually the system console.
- `sys.stderr` is the data stream that represents the standard error device - usually the system console.

Although `stdin`, `stdout`, and `stderr` data streams are opened by Python automatically, the `sys` module must be imported to access them directly.

sys_io.py

```
#!/usr/bin/env python3
import sys


def main():
    sys.stdout.write("Please enter some text:\n")
    x = sys.stdin.readline()
    # Use of literal fstring instead of format method
    sys.stdout.write(f"Standard Output\n{x}")
```

```
    sys.stderr.write(f"Error Output\n{x}")


if __name__ == "__main__":
    main()
```

Creating Your Own Data Streams

The built-in `open()` function opens a file and returns a data stream.

When opening a data stream, you must declare it as an input data stream, or as an output data stream.

If the `open()` function fails, then a subclass of `OSError` is raised.

The `open()` function accepts multiple arguments:

- It has a required string as its first argument, representing the name of the file.
- There is also an optional argument named `mode`, that among other things is used to declare the type of data stream it should create.

References

A complete list of named arguments and their meanings can be found in the documentation for the `open()` function here:

`https://docs.python.org/3/library/functions.html#open`

The following table lists the different values that can be used to specify the mode in which the file is opened.

File Opening Modes

Mode	File Opened For:
`r`	Reading (`default`). It fails if the file does not exist.
`w`	Writing. It truncates the file if it already exists.
`x`	Exclusive creation. It fails if the file already exists (added in Python 3.3).
`a`	Appending to end of file. It creates the file if it does not exist.
`b`	Binary mode.
`t`	Text mode (`default`).
`+`	Updating (reading and writing).

- As an example, `mode="w+b"` would open a file for binary read-write access.
- Once a stream is opened, various methods can then be used to read and write files.

Writing to a Text File

Here is a simple program that writes data to a file.

- When the program is run, the `print` statements output information about the stream returned by the call to `open()`.
- The calls to `write()` will output to the file rather than the display.
- Finally, the `close()` function closes the stream.
  - It is always recommended to close a stream you opened after you are finished processing it.

write1.py

```
#!/usr/bin/env python3
def main():
    f = open('output', 'w')
    print("Type:", type(f).__name__, "\tModule:", type(f).__module__)
    f.write('This is a test.\n')
    f.write('This is another test.\n')
    f.close()


if __name__ == "__main__":
    main()
```

The file name could be given as a command line argument or as user input.

- Note the use of a shorthand `if` statement to determine the file name.

write2.py

```
#!/usr/bin/env python3
import sys


def main():
    filename = sys.argv[1] if len(sys.argv) > 1 else input("Enter file name: ")
    f = open(filename, 'w')
    f.write('This is a test.\n')
    f.close()


if __name__ == "__main__":
    main()
```

The `write()` function can only write strings. Any data that is not text will have to be converted first.

write3.py

```
#!/usr/bin/env python3
import datetime


def main():
    f = open('output', 'w')
    data = [1, 2, 3]
    today = datetime.datetime.today()
    # f.write(today)  ~ Will not work since not a string
    f.write(str(data) + "\n")
    f.write(str(today) + "\n")
    f.close()


if __name__ == "__main__":
    main()
```

The following program appends data to a file and uses the `writelines()` method, which writes all the elements of a `list` of strings passed as a parameter.

writelines.py

```
#!/usr/bin/env python3
def get_data():
    the_list = []
    while True:
        data = input("Enter data ('q' to exit): ")
        if data == "q":
            break
        the_list.append(data)
    return the_list


def main():
    data = get_data()
    f = open("output", "a")
    f.writelines(data)
    f.close()


if __name__ == "__main__":
    main()
```

The `print()` function can also be used to write data to a file.

- The named parameter, `file`, can be used to specify an output file.

print1.py

```
#!/usr/bin/env python3
from datetime import datetime


def main():
    f = open("output", "w")
    cnt = 1
    while True:
        data = input("Enter data ('q' to exit): ")
        if data == "q":
            break
        txt = "{:04}".format(cnt)
        print(txt, datetime.today(), data, file=f)
        cnt += 1
    f.close()


if __name__ == "__main__":
    main()
```

When dealing with stream objects, it is good practice to use the `with` keyword.

- This has the advantage that the stream is properly closed after finishing its processing.

print2.py

```
#!/usr/bin/env python3
def main():
    with open("output", "w") as a_file:
        while True:
            data = input("Enter data ('q' to exit): ")
            if data == "q":
                break
            print(data, file=a_file)

    print("File Is Now Closed? ", a_file.closed)


if __name__ == "__main__":
    main()
```

Reading From a Text File

The following functions can be used to read data once a stream has been opened for reading.

- `read()`
  - For reading an entire stream into a string.
  - The number of characters to be read can be controlled by specifying the number of bytes to read at a time as a parameter.

- `readline()`
  - For reading a single line into a string, retaining newline character(s).
  - Returns an empty string when there is no more data to read.
- `readlines()`
  - For reading an entire stream into a `list`, where each element of the `list` contains a line from the stream.

The following program uses the `readline` method to read a line at a time from a file.

- It counts the number of lines and characters in a file.

read1.py

```
#!/usr/bin/env python3
def main():
    char_count = line_count = 0
    with open(input("Enter a file name: "), "r") as a_file:
        while True:
            txt = a_file.readline()
            if not txt:
                break
            char_count += len(txt)
            line_count += 1

    print("Characters:", char_count, " Lines:", line_count)


if __name__ == "__main__":
    main()
```

A stream object in Python is iterable, so it can be read from in a `for` loop.

read2.py

```
#!/usr/bin/env python3
def main():
    with open(input("Enter a file name: "), "r") as the_file:
        for a_line in the_file:
            print(a_line, end="")


if __name__ == "__main__":
    main()
```

The `readlines()` method reads the entire contents of a stream into a `list` of strings.

- Similar to the `readline()` method, all newline characters are retained.
- Each line from the stream is an element in the resulting `list`.

read3.py

```
#!/usr/bin/env python3
def main():
    with open(input("Enter a file name: "), "r") as the_file:
        the_lines = the_file.readlines()
    for a_line in the_lines:
        print(a_line, end="")


if __name__ == "__main__":
    main()
```

The `read()` method reads the data and returns it as a string.

- Passing no arguments will cause it to read the whole stream.
- Passing in a number as an argument indicates the quantity of data to be read.

The following example incorporates exception handling in addition to demonstrating the `read()` method.

- It also utilizes the built-in `string` module to easily obtain the letters of the alphabet.

read4.py

```
#!/usr/bin/env python3
import string


def main():
    try:
        with open("alphabet", "w") as the_file:
            the_file.write(string.ascii_letters)
        print("The following was written to the file:")
        print(string.ascii_letters, "\n")

        with open("alphabet", "r") as the_file:
            while True:
                the_text = the_file.read(10)
                if not the_text:
                    break
                print(the_text)
    except OSError as err:
        print("IO Error:", err)


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 read3.py
The following was written to the file:
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ

abcdefghij
klmnopqrst
uvwxyzABCD
EFGHIJKLMN
OPQRSTUVWX
YZ
$
```

- The `read()` method from the preceding example reads `10` characters at a time.
- The second to last call to `read()` returned the two remaining characters because the total number of characters was not a multiple of `10`.
- The last call to `read()` returned an empty string, which resulted in breaking out of the while loop.

bytes and bytearray Objects

The examples in this chapter, up to this point, have focused on reading and writing text files as opposed to *binary files*.

- The `open()` function has returned a `TextIOWrapper` object which expects and produces `str` objects to read and write.
- Writing and reading to and from binary files deal with `bytes` and `bytearray` objects instead of strings.
- No encoding, decoding, or newline translation is performed.

A `bytearray` object is a mutable sequence of integers in the range `0 >= x < 256`.

- It has most of the usual methods of mutable sequences.
- It also has most of the methods that a `bytes` object has.

A `bytes` object is an immutable sequence of integers in the range `0 >= x < 256`.

- A `bytes` object is an immutable version of `bytearray`.
- The syntax for a literal `bytes` object is similar to that of string literals, except that a `b` prefix is added:

```
b'this is a bytes object'
b"this is also a bytes object"
```

Converting between strings and bytes can be done by using the `bytes decode()` and the `str encode()` methods.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>> x = b"Goodbye"
>>> y = "Hello"
>>> print(type(x), type(y))
<class 'bytes'> <class 'str'>
>>> x = x.decode()
>>> y = y.encode()
>>> print(type(x), type(y))
<class 'str'> <class 'bytes'>
```

Reading and Writing Binary Files

The following example reads and writes binary text by using `bytes` and `bytearray` objects.

binary_io.py

```
#!/usr/bin/env python3
def main():
    text = "This is a string of data to be written\n"
    some_bytes = b"This should also be written to the file\n"
    a_byte_aray = bytearray("Hello", "utf-8")
    more_bytes = bytes("\nConverting this to bytes", "ascii")
    try:
        # Write to the file
        with open("binarydata", "wb") as the_file:
            print("Type of stream:", type(the_file).__name__)
            print("Module:", type(the_file).__module__)
            the_file.write(text.encode())
            the_file.write(some_bytes)
            the_file.write(a_byte_aray)
            the_file.write(more_bytes)

        # Read from the file
        with open("binarydata", "rb") as the_file:
            buffer = b''
            while True:
                the_text = the_file.read(10)
                if not the_text:
                    break
                buffer += the_text
        print(buffer)
        print("*" * 50)
        print(buffer.decode())
    except OSError as err:
        print("OS Error:", err)


if __name__ == "__main__":
    main()
```

Random Access

The `seek()` method on a file object provides arbitrary random access (seeking forwards or backwards to any location) within the file.

- The syntax for the method is `seek(offset[, whence])`
- The method changes the stream position to the given byte `offset`.
- The `offset` is interpreted relative to the position indicated by the optional parameter `whence`.
- The `os` module defines three int constants (`SEEK_SET`, `SET_CUR`, and `SEEK_END`) for the values of `whence`.

The table below shows the valid values for the `whence` argument.

- It lists the constant name, the constant value as an `int`, and the constant meaning.

whence Argument Values

Constant	Value	Meaning
`os.SEEK_SET`	`0`	Start of the stream (default). The *offset* should be zero or positive.
`os.SEEK_CUR`	`1`	Current stream position. The *offset* may be negative.
`os.SEEK_END`	`2`	End of the stream. The *offset* is usually negative

- Seeking relative to the current position and end position requires an offset of `0` (`os.SEEK_SET`) when working in text mode.

The `tell()` method returns the current position as the byte offset from the beginning of the file.

The following example demonstrates the use of the `seek()` and `tell()` methods while reading and/or writing files.

- The file read in the example contains the 26 letters of the alphabet.

seek1.py

```
#!/usr/bin/env python3
import os


def main():
    fmt = "CursorStart:{:<3}  Offset:{:<3}  Read:{:<3} " +\
          "CursorEnd:{:<3}  Data:{}"
    # Reading from file in binary mode
    with open("seekdata.txt", "rb") as f:
        offset, whence, chunk = (-20, os.SEEK_END, 5)
        f.seek(0, whence)
        start = f.tell()
        f.seek(offset, whence)
```

```
        data = f.read(chunk)
        print(fmt.format(start, offset, chunk, f.tell(), data))

        offset, whence, chunk = (3, os.SEEK_CUR, 7)
        start = f.tell()
        f.seek(offset, whence)
        data = f.read(chunk)
        print(fmt.format(start, offset, chunk, f.tell(), data))


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 seek1.py
CursorStart:26 Offset:-20 Read:5 CursorEnd:11 Data:b'GHIJK'
CursorStart:11 Offset:3 Read:7 CursorEnd:21 Data:b'OPQRSTU'
$
```

Working With Files and Directories

When reading and writing files, the `os` and `os.path` modules provide a variety of functions for working with files and directories.

os Module

Name	Behavior
`os.chdir()`	Changes the current working directory.
`os.getcwd()`	Returns a string representing the current working directory.
`os.listdir()`	Returns a list containing the names of the entries in a directory.
`os.mkdir()`	Creates a directory for a given path.
`os.makedirs()`	Recursive directory creation, creates all intermediate-level directories required to contain the leaf directory.
`os.remove()`	Deletes a file path. If the path is a directory, then an `OSError` exception is raised.
`os.rmdir()`	Deletes the directory path. It only works when the directory is empty.
`os.rename()`	Renames a file or directory.
`os.chown()`	Changes the owner and group ID of a given path.

Name	Behavior
`os.chmod()`	Changes the mode of a given path.
`os.stat()`	Gets the status of a file as an `os.stat_result` object.
`os.getenv()`	Returns the value of an environment variable.
`os.path.isabs()`	Returns `True` if the path is an absolute pathname.
`os.path.isfile()`	Returns `True` if the path is an existing regular file.
`os.path.isdir()`	Returns `True` if the path is an existing directory.
`os.path.dirname()`	Returns the directory name of a path name.
`os.path.exists()`	Returns `True` if the path refers to an existing path or an open file descriptor.
`os.path.getsize()`	Returns the size, in bytes, of a given path.
`os.path.join()`	Joins one or more path components by using the property directory separator.

The `os.stat_result` object has the following properties.

os.stat Object Properties

Index	Attribute	Meaning
`0`	`st_mode`	File mode (type and permissions).
`1`	`st_ino`	The inode number.
`2`	`st_dev`	Device number of the file system.
`3`	`st_nlink`	Number of hard links to the file.
`4`	`st_uid`	Numeric user ID of file's owner.
`5`	`st_gid`	Numerical group ID of file's owner.
`6`	`st_size`	Size of the file in bytes. Same value as the one returned by `os.path.getsize()`.

Index	Attribute	Meaning
7	`st_atime`	Last access time (seconds since epoch). Same value as the one returned by `os.path.getatime()`.
8	`st_mtime`	Last modify time (seconds since epoch). Same value as the one returned by `os.path.getmtime()`.
9	`st_ctime`	Linux: Last inode change time (seconds since epoch). Windows: Creation time (seconds since epoch). Same value as the one returned by `os.path.getctime()`.

The following application demonstrates many of the functions available in the `os` and `os.path` modules.

filestats.py

```
#!/usr/bin/env python3
import sys
import os
import time


def fileinfo(files):
    maxlength = str(len(max(files, key=len)))
    # Use of ^ for centering and the use of a nested {}  ~   {:>12{}}
    fmt = "{:" + maxlength + "} {:^9} {:^9} {:^9} {:>12{}}"
    pieces = ("Name:", "Exists?", "File?", "Dir?", "Size(bytes)", "")
    # Use of *pieces to splat pieces into an argument list
    print(fmt.format(*pieces))
    for afile in files:
        print(fmt.format(afile, str(os.path.exists(afile)),
                         str(os.path.isfile(afile)), str(os.path.isdir(afile)),
                         os.path.getsize(afile), ","))


def allstats(afile):
    tag = ["mode", "inode#", "device#", "#links", "user", "group", "bytes",
           "last access", "last modified", "change/creation time"]
    print("File Stats for:", afile)
    stats = os.stat(afile)
    fmt = "{:>22} : {}"
    for i, stat in enumerate(stats):
        print(fmt.format(tag[i], stat))
```

```
def datestats(afile):
    print("More Stats for:", afile)
    stats = os.stat(afile)
    print("Last Access:", time.ctime(stats.st_atime))
    print("Last Modified:", time.ctime(stats.st_mtime))
    print("Last Change:", time.ctime(stats.st_ctime))


def main():
    cwd = os.getcwd()
    print("Current Directory:", cwd)
    newdir = "afolder/asubfolder"
    os.makedirs(newdir, exist_ok=True)

    # Get list of files and get info about first 5
    filelist = sorted(os.listdir(cwd))[:5]
    fileinfo(filelist)

    # Get stats on a file
    allstats(sys.argv[0])
    datestats(sys.argv[0])


if __name__ == "__main__":
    main()
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `io/solutions` directory.

Exercise 1

Write a program that counts the number of lines, words, and characters in each file named on the command line.

Exercise 2

Revise Exercise 1 so that it accepts as an optional first command line argument a `-t` option.

- The program must then only print the total number of lines, words, and characters in all the files combined.

Exercise 3

Write a program that asks the user for the names of an input and an output file.

- Open both of these files and then have the program read from the input file (by using `readline()`) and write to the output file (by using `write()`).
- In effect, this is a copy program, whose interface to the program might look like:

```
Enter the name of the input file: myinput
Enter the name of the output file: myoutput
```

Exercise 4

Rewrite Exercise 3 such that the file names are obtained from the command line if two arguments are supplied.

- If the number of arguments is not two, then it should fall back on prompting the user for the filenames.
- The interface might look like:

```
python3 your_program_name.py inputfile outputfile
```

Exercise 5

Add exception handling to the previous exercise so that if a file open fails, an `OSError` exception is handled, and the program is halted.

Exercise 6

Write a program that displays the file name, size, and modification date for all those files in a directory that are greater than a given size.

- The directory name and the size criteria are given as command line arguments.

Exercise 7

Create two data files, each with a set of names, one per line.

- Now, write a program that reads both files and lists only those names that are in both files.
- The two file names should be supplied on the command line.

Exercise 8

Now, create a few more files with one name per line.

- The program in this exercise should read all these files and print the number of times each line occurs over all the files.
- The file names should be supplied on the command line.
- The following files are available in the `AD141-apps` repository, within the `io/starter` directory, and you can use them in the exercise:

```
names_a.txt
names_b.txt
names_c.txt
names_d.txt
```

- The output from the program might be as follows:

```
Alice       4
Bart        2
Beverly     1
Bill        4
Chris       2
```

```
Dave        1
Frank       3
Jane        3
John        2
Mary        1
Mike        4
Peter       3
Susan       2
```

References

A complete list of named arguments and their meanings can be found in the documentation for the `open()` function here:

`https://docs.python.org/3/library/functions.html#open`

Summary

- You can operate with data from various sources, separate from the standard input and output files.
- You can use input data streams to read data, and output data streams to persist data.

Chapter 9

Data Structures

Goal

Use compact syntax and specialized statements to create, process, and sort data structures.

Objectives

- Use list comprehensions as an alternative and concise way of creating lists.
- Use dictionary comprehensions as an alternative and concise way of creating dictionaries.
- Understand and use generators to retrieve large amounts of data without the overhead of storing the data.
- Use generator expressions as an alternative and concise way of creating generators.
- Use the zip datatype to process parallel collections.

Sections

- Data Structures

Data Structures

Objectives

- Use list comprehensions as an alternative and concise way of creating lists.
- Use dictionary comprehensions as an alternative and concise way of creating dictionaries.
- Understand and use generators to retrieve large amounts of data without the overhead of storing the data.
- Use generator expressions as an alternative and concise way of creating generators.
- Use the zip datatype to process parallel collections.

List Comprehensions

- *List comprehensions* provide a concise way of creating a `list`.
- Each list comprehension consists of square brackets `[]` containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses.
- The result is a `list` obtained from evaluating the expression in the context of the `for` and `if` clauses that follow it.
- When the expression is placed inside parentheses `()`, each member of the resulting `list` is a `tuple`.
- The following example uses the interactive Python shell to demonstrate creating several list comprehensions.

list_comprehensions.py

```
#!/usr/bin/env python3
def main():
    sample = [hex(x) for x in range(0, 16)]
    print(type(sample), sample, sep="\t")
    print([(x, x * x) for x in range(2, 7)])
    print([x * x for x in range(1, 16) if x % 2 == 0])


if __name__ == '__main__':
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 list_comprehensions.py
<class 'list'>  ['0x0', '0x1', '0x2', '0x3', '0x4', '0x5', '0x6', '0x7', '0x8',
 '0x9', '0xa', '0xb', '0xc', '0xd', '0xe', '0xf']
[(2, 4), (3, 9), (4, 16), (5, 25), (6, 36)]
[4, 16, 36, 64, 100, 144, 196]
$
```

- The following snippet includes more examples of list comprehensions:

triples.py

```
#!/usr/bin/env python3
def main():
    alist = [2, 4, 6]
    print([3 * x for x in alist])
    print([3 * x for x in [1, 2, 3, 4, 5]])
    print([3 * x for x in range(1, 5)])
    print([[x, 3 * x] for x in range(1, 5)])


if __name__ == "__main__":
    main()
```

The following example shows a function being applied to each element of a `list`.

from_functions.py

```
#!/usr/bin/env python3
def main():
    names = ["Ashley", "Emma", "Jayden", "Ethan"]
    print([len(name) for name in names])
    print([[name, len(name)] for name in names])
    grades = [[88, 77, 99], [95, 98, 97], [79, 100, 95]]
    highest = [max(grade) for grade in grades]
    print(highest)


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 from_functions.py
[6, 4, 6, 5]
[['Ashley', 6], ['Emma', 4], ['Jayden', 6], ['Ethan', 5]]
[99, 98, 100]
$
```

A list comprehension can have an `if` clause to act as a filter.

palindromes.py

```
#!/usr/bin/env python3
def main():
    words = ["hello", "racecar", "eye", "bike", "stats", "civic"]
    palindromes = [x for x in words if x[::-1] == x]
    print(palindromes)


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 palindromes.py
['racecar', 'eye', 'stats', 'civic']
$
```

Dictionary Comprehensions

A *dictionary comprehension* is similar to a list comprehension, but it constructs a `dict` object instead of a `list`.

The syntax of a dictionary comprehension has two main differences:

- It is enclosed in curly braces `{}` instead of square brackets.
- It contains two expressions, separated by a colon `:`.
  - The expression before the colon is the dictionary key.
  - The expression after the colon is the dictionary value.

dictionary_comprehension.py

```
#!/usr/bin/env python3
def main():
    names = ["Ashley", "Emma", "Jayden", "Ethan"]
    print({name: len(name) for name in names})


if __name__ == "__main__":
    main()   # Output: {'Jayden': 6, 'Ethan': 5, 'Ashley': 6, 'Emma': 4}
```

The following example maps the names of a file to its sizes for each file in the current directory.

fileinfo.py

```
#!/usr/bin/env python3
import os


def main():
```

```
    files = os.listdir(".")
    fileinfo = {f: os.path.getsize(f) for f in files}
    for name, size in fileinfo.items():
        print("{0:30} : {1:>6,} bytes".format(name, size))


if __name__ == "__main__":
    main()
```

Dictionaries with Compound Values

The following example is more complex and uses both list and dictionary comprehensions.

- The example creates a dictionary of customers where the value is a list of all the customer information.
- The example uses the data from the `customers.txt` file.
- A portion of the comma separated file is shown below.

customers.txt

```
Alice,Young,1 Main St,Carrington,ND,58421
Amanda,Wright,103 Center St,Mannington,WV,26582
Amy,Martinez,1108 Sheller Ave,Paden City,WV,26159
Andrew,Miller,113 Mountain Village Rd Apt B,New Rockford,ND,58356
Angela,Mitchell,119 W Court St,Veedersburg,IN,47987
Ann,Moore,12 34th St,Berlin,MD,21811
Anna,Morris,1209 7th St Ne,Chambersburg,PA,17201
Anthony,Nelson,1251 Hagan Dr,Evanston,WY,82930
Barbara,Parker,127 Franklin St,Goldendale,WA,98620
Betty,Perez,127 S Wood St,Ocean City,MD,21842
Brenda,Phillips,1300 N Yorktown Dr,Devils Lake,ND,58301
Brian,Reed,134 Adams St,Ashdown,AZ,71822
Carl,Roberts,1411 Us Highway 59 S,Lyman,WY,82937
Carol,Robinson,1600 N Hervey,Bedford,PA,15522
Edward,Harris,406 S Jackson Ave,Fowler,IN,47944
Christine,Scott,1807 W Pike St Ste B,Hope,AZ,71801
Christopher,Smith,1918 Mercersburg Rd,Juneau,AK,99801
Cynthia,Stewart,212 Laurel St,Prescott,AZ,71857
Daniel,Taylor,225 Lincoln Way W,Kemmerer,WY,83101
David,Thomas,815 Hill Ave,Clarksburg,WV,26301
Deborah,Thompson,800 Booth St,Greencastle,PA,17225
Jennifer,Turner,746 Saint Andrews Blvd,Texarkana,AZ,71854
Jeffrey,Walker,715 N 42nd St,Mc Connellsburg,PA,17233
Jason,White,6100 E Rio Grande Ave,Newport,IN,47966
Janet,Williams,607 Waynetown Rd,Mercersburg,PA,17236
James,Wilson,537 10th St,Grand Forks,ND,58201
Henry,Martin,524 N Lincoln,Toppenish,WA,98948
Douglas,Hall,400 Se Lincoln Rd,Bloomington,IN,47404
Dorothy,Green,3880 E 3rd St,Yakima,WA,98908
Donna,Gonzalez,3375 Koapaka St Suite 250,Clinton,IN,47842
Donald,Garcia,330 Boise St,Defuniak Springs,FL,32435
Diane,Evans,3142 Tyler Hwy,Charleston,SC,29407
```

The example is broken down into the following five function calls to more clearly delineate each step of the process.

- The `get_customers()` function reads from the text file and returns it as a nested list.
- The `get_info()` function returns information about the nested list, basically for informational purposes.
- The `get_dictionary()` function is where the nested list is converted into a dictionary, using a dictionary comprehension, and returned.
- The `print_customers()` function prints out all the customer names (the keys) of the resulting dictionary.
- The `user_interaction()` function gets a customer name from the user and prints the value associated with that key.

The module that defines all the preceding functions is as follows:

customer_functions.py

```
#!/usr/bin/env python3
def get_customers():
    with open("customers.txt", "r") as thefile:
        customer_list = thefile.readlines()
    # use a list comprehension to convert to a
    # nested list of lists (each a list of strings)
    return [customer.rstrip().split(",") for customer in customer_list]


def get_info(customers):
    print("Nested Structure:", type(customers),
          type(customers[0]), type(customers[0][0]))
    # partial contents of customers
    print(customers[0], customers[1], sep="\n", end="\n\n")


def get_dictionary(customers):
    # Convert to dictionary using a dictionary comprehension
    return {"{} {}".format(cust[0], cust[1]): cust for cust in customers}


def print_customernames(customer_names):
    print("Customer names:")
    for i, name in enumerate(customer_names):
        print("{0:20}".format(name), end="|")
        if i % 4 == 3:
            print()
    print("\n")


def user_interaction(customers):
    tags = ["FirstName", "LastName", "Street", "City", "State", "ZipCode"]
    fmt = "{:16}{:16}{:20}{:16}{:6}{:5}"
    while True:
```

```
        name = input("Enter a name (or 'quit' to quit):")
        if name == "quit":
            break
        data = customers.get(name)
        if data:
            print(fmt.format(*tags))
            print(fmt.format(*data))
```

An application that calls all the preceding functions is as follows:

customers.py

```
#!/usr/bin/env python3
import customer_functions


def main():
    customer_list = customer_functions.get_customers()
    customer_functions.get_info(customer_list)
    customer_map = customer_functions.get_dictionary(customer_list)
    customer_functions.print_customernames(customer_map.keys())
    customer_functions.user_interaction(customer_map)


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 customers.py
Nested Structure: <class 'list'> <class 'list'> <class 'str'>
['Alice', 'Young', '1 Main St', 'Carrington', 'ND', '58421']
['Amanda', 'Wright', '103 Center St', 'Mannington', 'WV', '26582']

Customer names:
Alice Young         |Amanda Wright       |Amy Martinez        |Andrew Miller
 |
Angela Mitchell     |Ann Moore           |Anna Morris         |Anthony Nelson
 |
Barbara Parker      |Betty Perez         |Brenda Phillips     |Brian Reed
 |
Carl Roberts        |Carol Robinson      |Edward Harris       |Christine Scott
 |
Christopher Smith   |Cynthia Stewart     |Daniel Taylor       |David Thomas
 |
Deborah Thompson    |Jennifer Turner     |Jeffrey Walker      |Jason White
 |
Janet Williams      |James Wilson        |Henry Martin        |Douglas Hall
 |
Dorothy Green       |Donna Gonzalez      |Donald Garcia       |Diane Evans
 |
Dennis Edwards      |Debra Davis         |Jerry Cook          |Jessica Collins
 |
```

```
John Clark          |Alicia Reed         |

Enter a name (or 'quit' to quit):Alice Young
FirstName       LastName        Street              City            State ZipCode
Alice           Young           1 Main St           Carrington      ND    58421
Enter a name (or 'quit' to quit):Jessica Collins
FirstName       LastName        Street              City            State ZipCode
Jessica         Collins         300 E 1st N         Burlington      CO    80807
Enter a name (or 'quit' to quit):quit
$
```

Generators

While `list` and dictionary comprehensions are concise ways of creating their respective data types, they require that the entire structure be created in memory before being available for use.

- If the amount of data is huge, then it requires a lot of memory.
- If the amount of data is infinite, then their use becomes impossible.

Generators are often used in situations where a lot of data needs to be generated without the overhead of storage of the entire dataset.

- Generators are a special class of functions that simplify the task of writing iterators.
- Any function containing a `yield` keyword is a generator function.
  - On reaching a `yield`, the generator's state of execution is suspended and local variables are preserved.
  - On the next call to the generator's `next()` method, the function will resume executing.

Here is a simple example of a generator function.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def generate_odds(upper_limit):
...     for odd in range(1, upper_limit, 2):
...         yield odd
...
>>> odds = generate_odds(5)
>>> print(type(odds))
<class 'generator'>
>>> next(odds)
1
>>> next(odds)
3
>>> next(odds)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> quit()
$
```

As seen in the output on the previous example; if the `next()` function is called too many times for a given generator object, a `StopIteration` exception is raised.

Since a generator acts as an iterator, it can be used with a `for` loop.

- The benefit of this is that the `for` loop automatically calls the `next()` function and silently handles the `StopIteration` exception.
  - This can be seen in the following example.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> def generate_odds(upper_limit):
...     for odd in range(1, upper_limit, 2):
...         yield odd
...
>>> for val in generate_odds(5):
...     print(val, end=" ")
...
1 3
>>> exit()
$
```

The following example defines a generator to iterate through a given number of days relative to the current date.

generate_dates.py

```
#!/usr/bin/env python3
import datetime


def following_days(howmany):
    now = datetime.date.today()
    for i in range(1, howmany + 1):
        yield now + datetime.timedelta(days=i)


def main():
    print("Code was run on:", datetime.date.today())
    print("Next 7 days are:")
    for adate in following_days(7):
        print(adate)


if __name__ == "__main__":
    main()
```

Generator Expressions

Similar to list and dictionary comprehensions, a *generator expression* is a concise way of creating a generator.

- While *list comprehensions* use square brackets and dictionary comprehensions use curly braces in their syntax, generator expressions use parentheses `()`.

The following example rewrites the odd number generator defined previously as a *generator expression*.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> for x in (odd for odd in range(1, 1000, 2)):
...     print(x, end = "|")
...
1|3|5|7|9|11|13|15|17|19|21|23|25|27|29|31|33|35|37|39|41|43|45|47|49|51|53|55|57
 |59|61|63|65|67|69|71|73|75|77|79|81|83|85|87|89|91|93|95|97|99|101|103|105|107|
109|111|113|115|117|119|121|123|125|127|129|131|133|135|137|139|141|143|145|147|
149|151|153|155|157|159|161|163|165|167|169|171|173|175|177|179|181|183|185|187|
189|191|193|195|197|199|201|203|205|207|209|211|213|215|217|219|221|223|225|227|
229|231|233|235|237|239|241|243|245|247|249|251|253|255|257|259|261|263|265|267|
269|271|273|275|277|279|281|283|285|287|289|291|293|295|297|299|301|303|305|307|
309|311|313|315|317|319|321|323|325|327|329|331|333|335|337|339|341|343|345|347|
349|351|353|355|357|359|361|363|365|367|369|371|373|375|377|379|381|383|385|387|
389|391|393|395|397|399|401|403|405|407|409|411|413|415|417|419|421|423|425|427|
429|431|433|435|437|439|441|443|445|447|449|451|453|455|457|459|461|463|465|467|
469|471|473|475|477|479|481|483|485|487|489|491|493|495|497|499|501|503|505|507|
509|511|513|515|517|519|521|523|525|527|529|531|533|535|537|539|541|543|545|547|
549|551|553|555|557|559|561|563|565|567|569|571|573|575|577|579|581|583|585|587|
589|591|593|595|597|599|601|603|605|607|609|611|613|615|617|619|621|623|625|627|
629|631|633|635|637|639|641|643|645|647|649|651|653|655|657|659|661|663|665|667|
669|671|673|675|677|679|681|683|685|687|689|691|693|695|697|699|701|703|705|707|
709|711|713|715|717|719|721|723|725|727|729|731|733|735|737|739|741|743|745|747|
749|751|753|755|757|759|761|763|765|767|769|771|773|775|777|779|781|783|785|787|
789|791|793|795|797|799|801|803|805|807|809|811|813|815|817|819|821|823|825|827|
829|831|833|835|837|839|841|843|845|847|849|851|853|855|857|859|861|863|865|867|
869|871|873|875|877|879|881|883|885|887|889|891|893|895|897|899|901|903|905|907|
909|911|913|915|917|919|921|923|925|927|929|931|933|935|937|939|941|943|945|947|
949|951|953|955|957|959|961|963|965|967|969|971|973|975|977|979|981|983|985|987|
989|991|993|995|997|999|>>>
>>> quit()
$
```

Processing Parallel Collections

When working with data structures such as `lists` and `tuples`, it is common to have multiple structures whose data runs in parallel.

- The built-in `zip()` constructor can be used to easily create a `zip` object that can iterate through each of the collections in parallel.

in_parallel.py

```
#!/usr/bin/env python3
long_names = ["January", "February", "March", "April",
              "May", "June", "July", "August", "September",
              "October", "November", "December"]
abbr_names = ["Jan", "Feb", "Mar", "Apr", "May", "Jun",
              "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]
numdays = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]

fmt = "{:^10} {:^10} {}"
print(fmt.format("#Days:", "Abbr Name:", "Long Name:"))
for days, abbr, lng in zip(numdays, abbr_names, long_names):
    print(fmt.format(days, abbr, lng))
```

- If the iterable objects passed to the `zip()` constructor are not all the same size, the shortest argument dictates the number of times the `zip` object can be iterated over.

Specialized Sorts

Several examples throughout the course have shown the use of the `sort()` method of a `list` and the top level `sorted()` function.

- Sorting a sequence such as a `list` or a `tuple` first compares the first two items, and if they differ this determines the outcome of the comparison.
  - If they are equal, the next two items are compared, and so on, until either sequence is exhausted.
- This n-ary level of sorting is demonstrated in the tertiary sort of the two-dimensional list shown below.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> thelist = [[1,1,7], [1,1,1], [1,2,3], [1,2,1], [2,1,1], [1,2,2]]
>>> thelist.sort()
>>> print(thelist)
[[1, 1, 1], [1, 1, 7], [1, 2, 1], [1, 2, 2], [1, 2, 3], [2, 1,
1]]
>>> quit()
$
```

The ability of a sequence to be sorted to any level provides additional capabilities when calling the `sort()` method and `sorted()` function with the `key` parameter.

- The following example demonstrates a tertiary sort of customer's information.
  - The primary sort will be by state.
  - If the states match, the secondary sort will be by last name.
  - If the last names match, the tertiary sort will be by first name.

- This is accomplished by defining a lambda function that takes a list of strings pertaining to the customer and returns a tuple of the state, last name and first name.

- The application reuses the `get_customers()` function from the `customer_functions` module defined earlier in the chapter.

tertiary_sort.py

```
#!/usr/bin/env python3
from customer_functions import get_customers


def main():
    customer_list = get_customers()
    customer_list.sort(key=lambda x: (x[4], x[1], x[0]))
    for customer in customer_list:
        print(customer)


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 tertiary_sort.py
['Christopher', 'Smith', '1918 Mercersburg Rd', 'Juneau', 'AK', '99801']
['Alicia', 'Reed', '34 Southern Blvd', 'Tucson', 'AZ', '71801']
['Brian', 'Reed', '134 Adams St', 'Ashdown', 'AZ', '71822']
['Christine', 'Scott', '1807 W Pike St Ste B', 'Hope', 'AZ', '71801']
['Cynthia', 'Stewart', '212 Laurel St', 'Prescott', 'AZ', '71857']
['Jennifer', 'Turner', '746 Saint Andrews Blvd', 'Texarkana', 'AZ', '71854']
['Jessica', 'Collins', '300 E 1st N', 'Burlington', 'CO', '80807']
['Donald', 'Garcia', '330 Boise St', 'Defuniak Springs', 'FL', '32435']
['Dennis', 'Edwards', '306 Hwy 59 N.', 'Mecca', 'IN', '47860']
['Donna', 'Gonzalez', '3375 Koapaka St Suite 250', 'Clinton', 'IN', '47842']
['Douglas', 'Hall', '400 Se Lincoln Rd', 'Bloomington', 'IN', '47404']
['Edward', 'Harris', '406 S Jackson Ave', 'Fowler', 'IN', '47944']
['Angela', 'Mitchell', '119 W Court St', 'Veedersburg', 'IN', '47987']
...
$
```

- The second and third lines of the preceding output show how the customers were sorted by their first names because the state and last name were the same for both entries.

A more object-oriented approach to the previous examples dealing with the `customers.txt` file, would be to start by defining classes to represent the data as an `Address` class and a `Customer` class to represent all the data as objects.

- These two data types are defined in the following module.

customer_and_address.py

```
#!/usr/bin/env python3
class Address:
    def __init__(self, street, city, state, zip):
        self.street = street
        self.city = city
        self.state = state
        self.zip = zip

    def __str__(self):
        return "".join([self.street, "\n", self.city, ", ", self.state, " ",
                        self.zip])


class Customer:
    def __init__(self, first_name, last_name, address):
        self.first_name = first_name
        self.last_name = last_name
        self.address = address

    def __str__(self):
        return "".join([self.first_name, " ", self.last_name, "\n",
                        str(self.address)])
```

The following application reads the `customers.txt` file into a list of `Customer` objects by using the two data types defined previously.

- The `list` is then sorted similarly to the previous tertiary sort.

tertiary_sort2.py

```
#!/usr/bin/env python3
from customer_and_address import Address, Customer


def get_customers():
    customer_list = []
    with open("customers.txt", "r") as thefile:
        for customer_txt in thefile:
            c = customer_txt.rstrip().split(",")
            address = Address(c[2], c[3], c[4], c[5])
            customer = Customer(c[0], c[1], address)
            customer_list.append(customer)
    return customer_list


def sortby(customer):
    return (customer.address.state, customer.last_name, customer.first_name)


def main():
    customer_list = get_customers()
```

```
    customer_list.sort(key=sortby)
    for customer in customer_list:
        print(customer, end="\n\n")


if __name__ == "__main__":
    main()
```

The execution output of the preceding program might be as follows:

```
$ python3 tertiary_sort2.py
Christopher Smith
1918 Mercersburg Rd
Juneau, AK 99801

Alicia Reed
34 Southern Blvd
Tucson, AZ 71801

Brian Reed
134 Adams St
Ashdown, AZ 71822

Christine Scott
1807 W Pike St Ste B
Hope, AZ 71801
$
```

Python provides the `operator` module as part of the standard library that contains convenience functions to make it easier and faster to do both basic and specialized sorts.

- The `operator` module has an `itemgetter()` function and an `attrgetter()` function that can be used as the value of the key parameter to both the `sort()` method of a `list` and the `sorted()` function.
  - The following example uses the `operator.itemgetter()` and `operator.attrgetter()` functions to sort customers.

items_and_attributes.py

```
#!/usr/bin/env python3
from operator import itemgetter, attrgetter
from customer_functions import get_customers as getlist01
from tertiary_sort2 import get_customers as getlist02


def main():
    nestedlist = getlist01()
    # get items 4, 1 and 0 as the sort keys from the list
    nestedlist.sort(key=itemgetter(4, 1, 0))
    for alist in nestedlist[:3]:
        print(alist)
```

```
    print()
    customerlist = getlist02()
    # get the address.state, last_name, and first_name
    # attributes from each Customer object being sorted
    customerlist.sort(key=attrgetter('address.state', 'last_name',
                                     'first_name'))
    for customer in customerlist[:3]:
        print(customer, end=2*"\n")


if __name__ == "__main__":
    main()
```

The execution output of the preceding example might be as follows:

```
$ python3 items_and_attributes.py
['Christopher', 'Smith', '1918 Mercersburg Rd', 'Juneau', 'AK','99801']
['Alicia', 'Reed', '34 Southern Blvd', 'Tucson', 'AZ', '71801']
['Brian', 'Reed', '134 Adams St', 'Ashdown', 'AZ', '71822']

Christopher Smith
1918 Mercersburg Rd
Juneau, AK 99801

Alicia Reed
34 Southern Blvd
Tucson, AZ 71801

Brian Reed
134 Adams St
Ashdown, AZ 71822
$
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `data_structures/solutions` directory.

Exercise 1

Write list comprehensions to produce the following lists:

- A `list` of elements `0,1,2,3,4,...,99`
- A `list` from the preceding comprehension of those values that are evenly divisible by 5.

Exercise 2

Write a list comprehension to create a `list` of `tuples`, of x and the factorial of x, for the numbers from *5* to *10* inclusive.

- The `math` module has a `factorial()` function that can be used.

Exercise 3

Write a dictionary comprehension that generates a dictionary of numbers and their factorials in the range (1,10).

- Using that dictionary, multiply 6 factorial times 5 factorial.

Exercise 4

Suppose there is a file with three values per line.

- The values are white space separated as follows.
  - `OwnerName ComputerType ComputerValue`
- Read the lines and make a dictionary of dictionaries so the keys are the owner and the values are a dictionary consisting of the computer type as the key and the computer value as the value.
- Finally, print the dictionary.
- The dataset might look as follows:

```
Joe Desktop 500
Joe Laptop 200
Joe Desktop 400
Mary Desktop 200
Mary Laptop 800
Beth Laptop 500
Beth Tablet 250
Joe Tablet 250
```

- The output might look as follows:

```
{
'Mary': {'Desktop': 200, 'Laptop': 800},
'Beth': {'Tablet': 250, 'Laptop': 500},
'Joe': {'Desktop': 900, 'Tablet': 250, 'Laptop': 200}
}
```

Summary

- You can use list and dictionary comprehensions as a concise way of creating your lists and dictionaries.
- To avoid the overhead of storing data sets in memory prior to the data structure creation, you can use generators.
- You can use specialized statements to process collections in parallel or to sort the data.

Chapter 10

Regular Expressions

Goal

Use pattern-matching to find, modify, or split strings.

Objectives

- Use the `re` module to perform regular expression pattern matching.
- Understand and use character classes and quantifiers when building a regular expression.
- Use groups to apply quantifiers to a group within an expression.
- Use groups to capture and manipulate the text that a regular expression matches.
- Understand and use the various functions within the `re` module to operate on regular expressions.

Sections

- Regular Expressions

Regular Expressions

Objectives

- Use the `re` module to perform regular expression pattern matching.
- Understand and use character classes and quantifiers when building a regular expression.
- Use groups to apply quantifiers to a group within an expression.
- Use groups to capture and manipulate the text that a regular expression matches.
- Understand and use the various functions within the `re` module to operate on regular expressions.

Introduction

Regular expressions are a highly specialized language used for pattern matching.

- By using this language, you can specify the rules for the set of possible strings that you want to match, modify, or split.

The **re** module provides regular expression matching operations within the Python language.

- The functions in the `re` module let you check if a particular string matches a given regular expression.

The following example demonstrates the use of the `re.search()` function.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> re.search("z", "abcdefg")  # No Match since "z" is not found in "abcdefg"
>>> re.search("c", "abcdefg")  # Match Found
<re.Match object; span=(2, 3), match='c'>
>>> exit()
$
```

- The `re.search()` function takes a regex pattern as its first argument, and the string to search as the second argument.
- It scans through the string, from left to right, looking for the first location where the regular expression pattern produces a match.
  - If a match is found, then it returns a corresponding match object.
  - If no match is found, then it returns `None`.
- The following application can be used to provide both a regular expression and a string to match against it.

regex_testing.py

```
#!/usr/bin/env python3
import re


def getinput(regex):
    prompt = "Enter a RegEx or (<enter> to reuse previous):"
    prevregex = regex
    regex = input(prompt)
    if not regex:
        regex = prevregex
    elif regex == "quit":
        return tuple()
    line = input("Enter a string to search: ")
    if line == "quit":
        return tuple()
    return (regex, line)


def main():
    previous_regex = ""
    print("Enter 'quit' at any time to quit the program")
    while True:
        the_tuple = getinput(previous_regex)
        if the_tuple:
            regex, text = the_tuple
            x = re.search(regex, text)
            if x:
                print(x, "\n")
            else:
                print("No Match found\n")
            previous_regex = regex
        else:
            break


if __name__ == "__main__":
    main()
```

Simple Character Matches

Regular expressions can contain both special and ordinary characters.

- Most ordinary characters, like A, a, or 8, are the simplest regular expressions; they simply match themselves, and they are also known as literals.
- You can concatenate literals, to compose patterns.

The output of `regex_testing.py` is shown below.

- It searches for strings that contain the regular expression: `the`.

```
$ python3 regex_testing.py
Enter 'quit' at any time to quit the program
Enter a RegEx or (<enter> to reuse previous):the
Enter a string to search: the
<re.Match object; span=(0, 3), match='the'>

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: themselves
<re.Match object; span=(0, 3), match='the'>

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: breathe
<re.Match object; span=(4, 7), match='the'>

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: This line contains the words the and breathe
<re.Match object; span=(19, 22), match='the'>

Enter a RegEx or (<enter> to reuse previous):quit
$
```

- The `span` is a `tuple` of both the start and end positions of the match.

Special Characters

Special characters, also called *metacharacters*, are characters that either stand for classes of ordinary characters, or affect how the regular expressions around them are interpreted.

Regular Expression Metacharacters

\	.	^	$	?	+	*	{	}	[	]	(	)	|

- If any of the preceding characters are to be searched for, then they need to be preceded with a `\` to escape their special meaning.

- For example, if you want to search for strings that contain the consecutive characters `+*`, then you can define the following regular expression: `\+\*`.

```
$ python3 regex_testing.py
Enter 'quit' at any time to quit the program
Enter a RegEx or (<enter> to reuse previous):\+\*
Enter a string to search: abc+
No Match found

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: abc+abc*
No Match found

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: abc+*abc
<re.Match object; span=(3, 5), match='+*'>
```

```
Enter a RegEx or (<enter> to reuse previous):quit
$
```

Character Classes

A character class is a set of characters enclosed within square brackets `[]`.

- It specifies the characters that will successfully match a single character from a given input string.
- Take for example the simple character class `[drm]ice`.
  - The above will match strings that contain `dice`, `mice`, or `rice`.
  - It would not match a string containing `vice`.

A ^ (caret) as the first character inside a character class negates the class, causing it to match all characters except those within the square brackets.

- Take for example the character class `[^drm]ice`.
  - The above will no longer match strings that contain `dice`, `mice`, or `rice`.
  - It will now match a string containing `vice`, and `helloicegoodbye`.

Ranges of characters can be specified within a character class through the inclusion of a `-` (minus sign) between two characters.

- Take for example the character class `[0123456789]`.
  - It can be simplified using a range and rewritten as `[0-9]`.
- Any digit or letter can be represented as a range of `[0-9a-zA-Z]`.

Some special sequences beginning with \ represent predefined character classes.

Character Classes for Regular Expressions

Character Class	Meaning
`.`	Any character.
`\d`	A Unicode digit. More than the standard digits of `[0-9]` or `[0123456789]`.
`\D`	A non Unicode digit. More than the standard of `[^0-9]` or `[^0123456789]`.
`\s`	A white space character. Equivalent to `[ \t\n\r\f\v]` and many other white space characters in Unicode.
`\S`	A non-white-space character. Equivalent to `[^ \t\n\r\f\v]` or `[^\s]`.
`\w`	A word character. Equivalent to `[a-zA-Z0-9_]`.

Character Class	Meaning
\W	A non-word character. Equivalent to [^a-zA-Z0-9_] or [^\w].

- The following example demonstrates searching for strings that contain two consecutive digits, followed by a white space character, followed by two consecutive digits.

```
$ python3 regex_testing.py

Enter 'quit' at any time to quit the program
Enter a RegEx or (<enter> to reuse previous):\d\d\s\d\d
Enter a string to search: 01234 56789
<re.Match object; span=(3, 8), match='34 56'>

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: ## ##
<re.Match object; span=(0, 5), match='## ##'>

Enter a RegEx or (<enter> to reuse previous):quit
$
```

- The preceding example includes the Unicode character # (\U0001D7E1 ~ MATHEMATICAL DOUBLE-STRUCK DIGIT NINE).

Quantifiers

Quantifiers specify how often a regular expression must match.

Regular Expression Quantifiers

Quantifier	Occurrences
*	Zero or more
+	One or more
?	Zero or one
{m,n}	Minimum m and maximum n
{m,}	Minimum m and maximum unbounded
{,n}	Minimum unbounded and maximum n
{m}	Exactly m

- The following regular expression matches any string with three digits, followed by one or more white spaces, followed by zero or more characters: \d{3}\s+.*

Greedy and Non-Greedy Quantifiers

The quantifiers as listed on the previous section are greedy. That is, when there is a choice, the longest match will be chosen.

- For example, given the following pattern `_.*_`, a match will occur if the target string is preceded and followed by an underscore character.
  - If the string to be searched using the preceding regular expression is `This_is_the_way_to_do_it`, then the following portions of it fit the regular expression pattern as the string is scanned left to right.

```
"_is_"
"_is_the_"
"_is_the_way_"
"_is_the_way_to_"
"_is_the_way_to_do_"
```

- Because a quantifier is greedy, the longest match will be chosen.

Placing a ? after a quantifier makes that quantifier a non-greedy quantifier, and it will match the shortest option.

The following example demonstrates the use of greedy and non-greedy quantifiers.

```
$ python3 regex_testing.py
Enter 'quit' at any time to quit the program
Enter a RegEx or (<enter> to reuse previous):_.*_
Enter a string to search: This_is_the_way_to_do_it
<re.Match object; span=(4, 22), match='_is_the_way_to_do_'>

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: This __ is also a match
<re.Match object; span=(5, 7), match='__'>

Enter a RegEx or (<enter> to reuse previous):_.*?_
Enter a string to search: This_is_the_way_to_do_it
<re.Match object; span=(4, 8), match='_is_'>

Enter a RegEx or (<enter> to reuse previous):quit
$
```

Alternatives

The | character can be used as an `or` operator to specify alternatives.

- For example, if you wished to search for "Anne", "Chris", or "Robert", you could code as follows.

```
$ python3 regex_testing.py
Enter 'quit' at any time to quit the program
Enter a RegEx or (<enter> to reuse previous):Anne|Chris|Robert
Enter a string to search: His name is Robert
<re.Match object; span=(12, 18), match='Robert'>
```

```
Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: Anne is her name
<re.Match object; span=(0, 4), match='Anne'>

Enter a RegEx or (<enter> to reuse previous):quit
$
```

Matching at Beginning and/or End

The ^ character can be used at the beginning of a regular expression to search for strings that start with the regular expression.

The $ character can be used as the last character of a regular expression to search for string that end with the regular expression.

Using both the ^ at the beginning and the $ at the end of a regular expression searches for strings that completely match the expression from start and end.

Grouping

Groups are marked by the `(` and `)` metacharacters, and they group together the expressions contained inside them.

- This permits quantifiers to be specified on the group.
- For example, `(ab)*` will match zero or more repetitions of `ab`.
  - The following regular expression uses a group to specify the optional part of a zip code.

```
\d{5}(-\d{4})?
```

- The grouping permits the ? quantifier to apply to the entire group of characters of the dash and 4 digits `(-\d{4})`.

Groups can be nested and also capture the starting and ending index of the text that they match.

- Each group matched can be retrieved by passing an argument to the `group()`, `start()`, `end()`, and `span()` methods of the resulting match object.
  - Groups are numbered starting with `0`.
  - Group `0` is always present and represents the whole regular expression.
  - Subgroups are numbered from left to right, from 1 upward.

The `groups()` method can be used to return a `tuple` containing the strings for all the subgroups.

- The returned `tuple` is from 1 up to however many there are.

The example on the following page is a rewrite of the previous application.

- It has been modified to print more information about any and all groups defined within the regular expression.

group_testing.py

```
#!/usr/bin/env python3
import re
from regex_testing import getinput


def print_details(m):
    headers = ("#", "Start", "End", "Span", "Text")
    fmt = "{} {:^7}{:^7}{:^10} {}"
    print(fmt.format(*headers))
    # Group 0
    print(fmt.format(0, m.start(0), m.end(0), str(m.span(0)), m.group(0)))
    # Group 1 to the Number of groups
    # Note use of value of 1 as starting enumerate value
    for idx, a_group in enumerate(m.groups(), 1):
        print(fmt.format(idx, m.start(idx), m.end(idx),
                         str(m.span(idx)), a_group))


def main():
    previous_regex = ""
    print("Enter 'quit' at any time to quit the program")
    while True:
        the_tuple = getinput(previous_regex)
        if the_tuple:
            regex, text = the_tuple
            x = re.search(regex, text)
            if x:
                print_details(x)
                print()
            else:
                print("No Match found\n")
            previous_regex = regex
        else:
            break


if __name__ == "__main__":
    main()
```

Several examples of using groups are shown in the following output.

```
$ python3 group_testing.py
Enter 'quit' at any time to quit the program
Enter a RegEx or (<enter> to reuse previous):((.*) (.*)) (.*)
Enter a string to search: *This is a string of text*
#  Start   End     Span    Text
0    0     24    (0, 24)   This is a string of text
1    0     19    (0, 19)   This is a string of
2    0     16    (0, 16)   This is a string
3   17     19    (17, 19)  of
4   20     24    (20, 24)  text
```

```
Enter a RegEx or (<enter> to reuse previous):www\.(.+)\.(com|edu|org)
Enter a string to search: www.somewhere.com
#  Start   End     Span    Text
0    0     17    (0, 17)   www.somewhere.com
1    4     13    (4, 13)   somewhere
2   14     17    (14, 17)  com

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: www.someothersite.org
#  Start   End     Span    Text
0    1     22    (1, 22)   www.someothersite.org
1    5     18    (5, 18)   someothersite
2   19     22    (19, 22)  org

Enter a RegEx or (<enter> to reuse previous):
Enter a string to search: The website, www.redhat.com, contains all our
 open-source products
#  Start   End     Span    Text
0   13     27    (13, 27)  www.redhat.com
1   17     23    (17, 23)  redhat
2   24     27    (24, 27)  com

Enter a RegEx or (<enter> to reuse previous):quit
$
```

Additional Functions

The `re` module contains several other functions operate on regular expressions.

Function	Action
`re.match()`	Determines if the match is found at the beginning of the string.
`re.fullmatch()`	Requires the complete string to match the regular expression pattern.
`re.findall()`	Finds all substrings that generate a match (as opposed to only the first), and returns them as a list.
`re.split()`	Splits a string based on a regular expression and returns a list of strings as the result.
`re.sub()`	Makes substitutions within a given string based on a regular expression.
`re.compile()`	Compiles a regular expression string to provide efficiency on multiple usage.

The following example demonstrates the use of the `re.findall()` and `re.sub()` functions.

findall_sub.py

```
#!/usr/bin/env python3
import re


def main():
    text = "The numbers 123 and 57 are odd while 948 and 2800 are even"
    numbers = re.findall(r"\d+", text)
    print(numbers)

    result = re.sub(r"(\d+)", "#\\1", text)
    print(result)

    result = re.sub(r"(\d+)", r"#\1", text)
    print(result)


if __name__ == "__main__":
    main()
```

- The `#\\1` used as the 2nd argument to `re.sub()` acts as a back-reference to the group `#1`, matched in the first argument.
  - The double backslash is required since Python treats `\1` as a special character.
  - The `r"#\1"` defined as a raw string, prevents the need to escape the `\` character.

The output of the above program is shown below.

```
$ python3 findall_sub.py
['123', '57', '948', '2800']
The numbers #123 and #57 are odd while #948 and #2800 are even
The numbers #123 and #57 are odd while #948 and #2800 are even
$
```

Flags

Many of the functions in the `re` module accept an optional flags argument, used to enable various special features and syntax variations.

- Flags are available in the `re` module under two names, a long and a short name.
- Multiple flags can be specified by bitwise OR-ing them.
  - For example, `re.I | re.M` sets both the `I` and `M` flags.

Flags available in the `re` Module

Flag Name	Short Name	Meaning
`re.ASCII`	`re.A`	Makes several escapes such as `\w`, `\b`,`\s`, and `\d` match only on ASCII characters with the respective property.

Flag Name	Short Name	Meaning
`re.DOTALL`	`re.S`	Make `.` match any character, including newlines.
`re.IGNORECASE`	`re.I`	Do case-insensitive matches.
`re.LOCALE`	`re.L`	Do a locale-aware match.
`re.MULTILINE`	`re.M`	Multi-line matching, affecting ^ and $.
`re.VERBOSE`	`re.X`	Enable verbose REs, which can be organized more cleanly and understandably.

The following example demonstrates the use of the `re.IGNORECASE` flag.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> x = re.findall("hello", "Hello hello HeLLo heLLO", flags=re.IGNORECASE)
>>> print(x)
['Hello', 'hello', 'HeLLo', 'heLLO']
>>> quit()
$
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `regular_expressions/solutions` directory.

Exercise 1

Write a program that reads a line at a time and determines whether the input consists solely of an integer number that is positive or negative.

- Specify whether it is positive or negative.

Exercise 2

Repeat the previous exercise, but this time use a floating point number.

- A floating point number should have at least one digit to the left and to the right of the decimal point.
- Specify whether the number is positive or negative.

Exercise 3

Write a program that reads lines from the user one at a time to see if they are formatted according to the following criteria.

- Correctly formatted lines should consist of a four character identifier, any number of spaces or tabs, and a description.

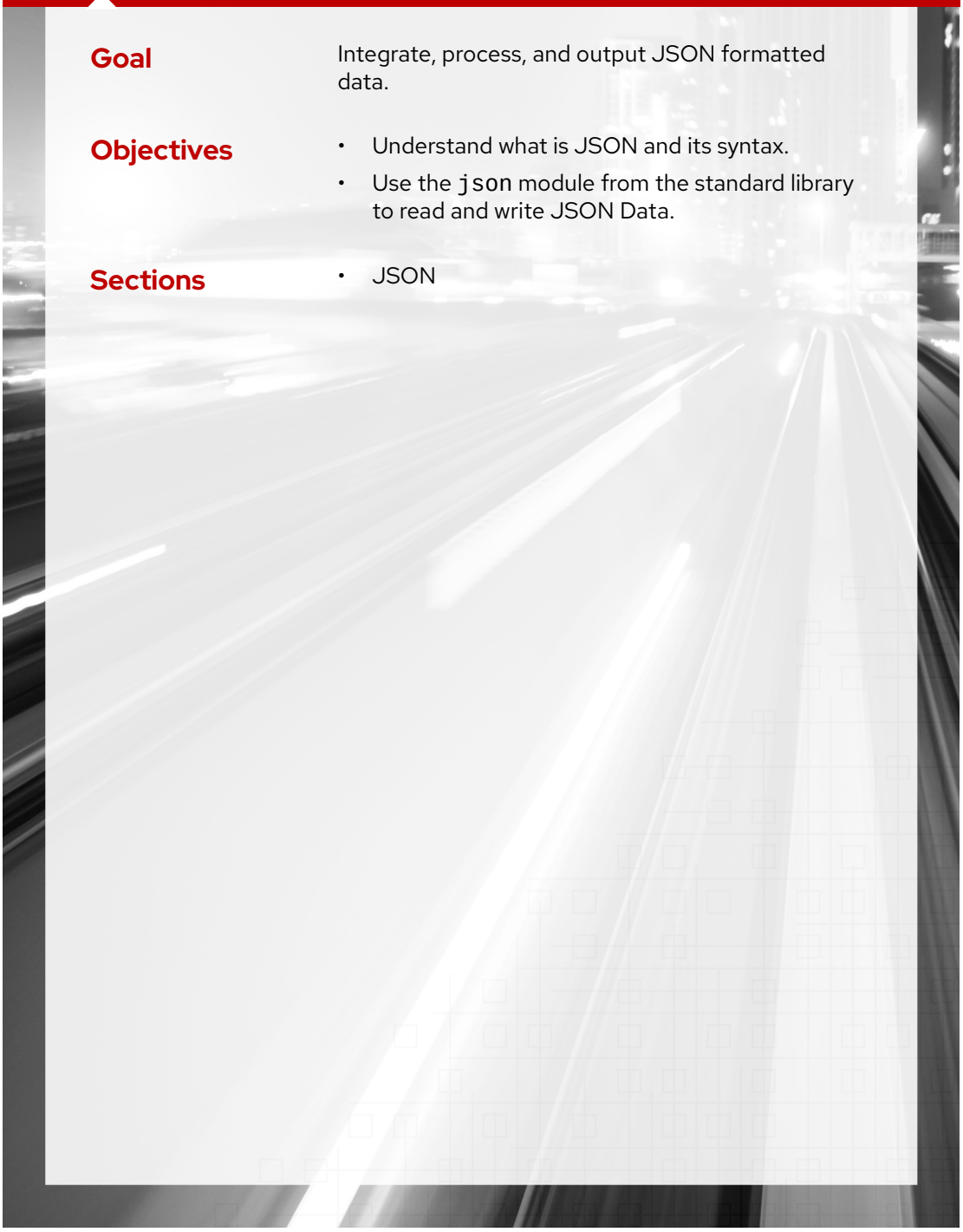
- The four character identifier should consist of two digits followed by two uppercase characters.
- For each correctly formatted line, print the two digits, the two characters, and the descriptions.
  - Print all of these pieces of information on separate lines.

Summary

- You can use regular expressions to match, modify, or split strings based on a given pattern.
- By including the `re` module in your applications, you can operate on regular expressions.
- You can compose regular expressions by using special and ordinary characters.

Chapter 11

JSON



Goal

Integrate, process, and output JSON formatted data.

Objectives

- Understand what is JSON and its syntax.
- Use the `json` module from the standard library to read and write JSON Data.

Sections

- JSON

JSON

Objectives

- Understand what is JSON and its syntax.
- Use the `json` module from the standard library to read and write JSON Data.

What is JSON?

- JSON stands for JavaScript Object Notation.
- It is plain text that stores data in key-value pairs.
- Self-describing text is easy to understand.
- The text must conform to a set of rules.
- It is a lightweight data interchange format.
- JSON is less verbose than XML.
- It is language independent.
- It is commonly used to share data between applications.
- RESTful APIs typically use JSON.

JSON Syntax Requirements

There are a few, simple rules to follow in JSON syntax:

- Data is in key-value pairs separated with a colon.
- Key-value pairs are comma separated.
- Data is wrapped in curly braces.
- Data can be nested.
- Various data types can be serialized to JSON.
- Arrays, hashes, numbers, strings, `true`, `false`, and `null` values can all be serialized.

A simple example:

```
{
  "make":"GMC",
  "model":"Canyon",
  "mileage":37654
}
```

Sample JSON

colors.json

```
{
  "colors": [
    {
      "name": "red",
      "value": "#f00"
    },
    {
      "name": "green",
      "value": "#0f0"
    },
    {
      "name": "blue",
      "value": "#00f"
    },
    {
      "name": "cyan",
      "value": "#0ff"
    },
    {
      "name": "magenta",
      "value": "#f0f"
    },
    {
      "name": "yellow",
      "value": "#ff0"
    },
    {
      "name": "black",
      "value": "#000"
    }
  ]

}
```

The json Module

The Python standard library includes a module named `json` that provides an API for working with JSON data.

Use the `loads` function to create a dictionary from a JSON formatted string.

`str`, `bytes`, or `bytearray` objects can be converted to a `dict`.

Use the `dumps` function to create a JSON formatted string from a dictionary.

By default, the `json` module performs the following object conversions:

JSON object	Python object
`object`	`dict`

JSON object	Python object
`array`	`list`, `tuple`
`string`	`str`
`number (int)`	`int`
`number (real)`	`float`
`true`	`True`
`false`	`False`
`null`	`None`

A simple example:

dumps_loads.py

```
#!/usr/bin/env python3
import json

data = '{"name": "Jill", "age": 33, "state": "MD"}'
j = json.loads(data)
print(type(j), j["state"])
print(type(j["age"]))

d = {"name": "Joe", "age": 57, "state": "PA"}
j = json.dumps(d)
print(type(j), j)
```

The output of dumps_loads.py would be:

```
<class 'dict'> MD
<class 'int'>
<class 'str'> {"state": "PA", "age": 57, "name": "Joe"}
```

The loads Function

The function raises a `JSONDecodeError` exception if the data being deserialized is invalid.

The `loads` function has several optional parameters, that you can verify in the following table:

`s`	A JSON formatted `str`, `bytes`, or `bytearray` object to deserialize. This parameter is required.
`object_hook`	A function to invoke that implements custom decoding. The return value will be used in place of `dict`

`object_pairs_hook`	A function to invoke that implements custom decoding of ordered list of pairs. The return value will be used in place of `dict`. Takes priority over `object_hook`
`parse_float`	A function to apply to each JSON float being decoded. Default is `float(data)`
`parse_int`	A function to apply to each JSON int being decoded. Default is `int(data)`
`parse_constant`	Can be `'-Infinity'`, `'Infinity'`, `'NaN'` and is used to raise an exception when decoding invalid JSON numbers
`cls`	A custom `JSONDecoder` subclass to be used. Default is `JSONDecoder`
`**kw`	Keyword arguments to be passed to the constructor of the class specified in the `cls` parameter

The following code shows how to use the `loads` function:

loads.py

```
#!/usr/bin/env python3
import decimal
import json

data = '{"x": 0.1, "y": 0.2}'
j = json.loads(data)
j_x = j["x"]
j_y = j["y"]
print(type(j), type(j_x), j_x, sep=", ")
print(j_x + j_y)

j = json.loads(data, parse_float=decimal.Decimal)
j_x = j["x"]
j_y = j["y"]
print(type(j), type(j_x), j_x, sep=", ")
print(j_x + j_y)

data = '{"name": "Jill", "age": 33, "state": "MD"}'
j = json.loads(data, parse_int=str)
print(type(j["age"]), j["age"], sep=", ")
```

The output of the preceding program is:

```
<class 'dict'>, <class 'float'>, 0.1
0.30000000000000004
<class 'dict'>, <class 'decimal.Decimal'>, 0.1
0.3
<class 'str'>, 33
```

The load Function

The `load` function deserializes JSON data stored in a text file.

Its functionality is similar to the `loads` function, because it also raises a `JSONDecodeError` exception on invalid data, and shares most of the optional parameters with it.

The `load` function also supports the `fp` parameter, which is the file containing JSON data.

As of Python 3.6, a binary file can also be deserialized.

The encondings supported by the `load` function are: UTF-8, UTF-16, or UTF-32.

The following example shows how to use the `load` function:

load_json_colors.py

```
#!/usr/bin/env python3
import decimal
import json

with open("colors.json", "r") as m:
    j = json.load(m, parse_float=decimal.Decimal)

print(type(j))
print(type(j["colors"]))
print(j["colors"][0]["name"])
value = j["colors"][0]["value"]
print(type(value), value)

print()

for color in j["colors"]:
    if "e" in color["name"]:
        print(color["name"])
```

The output of the preceding program is:

```
<class 'dict'>
<class 'list'>
red
<class 'str'> #f00

red
green
```

```
blue
magenta
yellow
```

The dumps Function

The `dumps` function converts a Python object into a string representation in JSON.

The following table summarizes the available parameters for the `dumps` function.

`obj`	The object to serialize as a JSON formatted `str`. This parameter is required.
`skipkeys`	Ignore non-built-in data types as keys when `True`, raise a `TypeError` otherwise
`ensure_ascii`	Escape all non-ASCII characters when `True`, convert them as-is otherwise
`check_circular`	Enable or disable circular reference checks. If disabled then circular references raise an `OverflowError`
`allow_nan`	Use the JavaScript values (`NaN`, `Infinity`, `-Infinity`) for the range float values (`nan`, `inf`, `-inf`) when `True`, raise a `ValueError`, otherwise
`indent`	The indent level for JSON data. Valid values are a string, an integer , or `None`
`separators`	A tuple in the form of (`item_separator`, `key_separator`). The default is (`', ', ': '`)
`default`	A function to invoke for objects that cannot be serialized. The function should return a JSON encoded value or a *TypeError*. *TypeError* is raised when nothing is specified
`sort_keys`	Sort the output when `True`.
`cls`	Custom `JSONEncoder` subclass to use. Default is `JSONEncoder`
`**kw`	Keyword arguments to pass to the constructor of the class specified in the `cls` parameter

The dump Function

The `dump` function is similar to the `dumps` function but outputs the JSON representation to a file.

The extra `fp` parameter controls the file to write the JSON.

As keys in key-value pairs of JSON are always `str`, when converting a `dict` to JSON, the function converts all the keys of the `dict` to `str`.

As a result, if a `dict` is converted to JSON and then back to a `dict` then the new `dict` may not equal the original.

The following code shows how to use the `dump` function:

tweedle_dump.py

```
#!/usr/bin/env python3
import decimal
import json
import requests

seps = (',', ':')

# get the current location of the ISS
url = "http://api.open-notify.org/iss-now.json"
response = requests.get(url)
if response.status_code == 200:
    data = json.loads(response.content.decode())
    with open("api_data.json", "w") as r:
        json.dump(data, r, skipkeys=True,
                  separators=seps, indent=2)
else:
    print("response status:", response.status_code)

print("Done")
```

Running the preceding program creates a file with the following content:

```
{
  "message":"success",
  "timestamp":1565112018,
  "iss_position":{
    "longitude":"110.7632",
    "latitude":"-36.0800"
  }
}
```

Note

The preceding example uses the `requests` Python module to perform HTTP requests to the API.

If the module is not installed then this example throws an error. To fix it install the module with `pip install requests`.

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `json/solutions` directory.

Exercise 1

Create a program that reads `books.json`.

- Prompt the user to enter a book title until they decide to quit.
- If the title is in the JSON data, print the data for that book.
- If the book is not in the JSON data, the program must indicate it.

Sample output:

```
Enter the title of a book (q to quit): Odyssey
Odyssey Info:
  year: 800
  pages: 374
  language: Greek
  author: Homer
  country: Greece
```

Exercise 2

Write a program that saves a dictionary as a JSON file.

- The dictionary should contain the word frequency (words as keys, frequency as values) read from the `cyclone` file.
- When saving the JSON file, the indentation level should be tab characters.
- The program should display the word that appeared most often in the file.

Sample output:

```
'the' occurred 93 times
```

Exercise 3

Retrieve and operate on tasks from https://jsonplaceholder.typicode.com/todos.

- Get the task list as JSON from the preceding API URL.
- Save the data to two local files, one that contains completed tasks and another one that contains incomplete tasks.
- The JSON data written to the files should be formatted as minimally as possible (no spaces, newlines, etc).
- The program should display the number of tasks we have completed. Sample output:

```
90 of 200 tasks are done
```

Exercise 4

The following API endpoint returns JSON data that contains a random trivia fact about a number sent to it: http://numbersapi.com/a_number/?json&notfound=floor.

- Write a program that makes an HTTP request to the API.
- The program should display the fact that was returned in the JSON data.

Another endpoint will return mathematical data about a number:

- http://numbersapi.com/a_number/math/?json&notfound=floor

```
sending request http://numbersapi.com/42/?json&notfound=floor...
42 is the result given by the web search engines Google, Wolfram Alpha and Bing
 when the query \"the answer to life the universe and everything\" is entered as a
 search.
```

Summary

- You can import the `json` module to work with JSON data.
- To deserialize and transform JSON data into a Python dictionary, you can use the `load` and `loads` functions.
- You can use the `dump` and `dumps` functions to convert Python objects into a JSON representation.

Chapter 12

Debugging

Goal

Detect and identify errors in Python applications.

Objectives

- Use the Python standard library's `pdb` module to debug Python programs.
- Launch the debugger from an interactive Python shell.
- Learn and use the various debugger commands within the debugging process.
- Set breakpoints and evaluate the current context of the code being debugged.

Sections

- Debugging

Debugging

Objectives

- Use the Python standard library's `pdb` module to debug Python programs.
- Launch the debugger from an interactive Python shell.
- Learn and use the various debugger commands within the debugging process.
- Set breakpoints and evaluate the current context of the code being debugged.

Introduction

A common practice among developers is to insert print statements in various places of code to get an idea of what is happening in the application. This approach is not wrong, but it is often better to use a debugger.

The `pdb` module defines an interactive source code debugger for Python programs.

- It supports setting breakpoints within the code and single stepping line by line through Python code.
- It also supports post-mortem debugging to better understand what happened when an exception was raised.

There are several ways of starting the debugger:

- From an interactive Python shell
- From the command line

All the exercises of this chapter use the following application for debugging purposes.

simple_program.py

```
#!/usr/bin/env python3


def sum_args(*args):
    total = 0
    for value in args:
        total += value
    return total


def main():
    a = 5
    b = 3
    result = sum_args(a, b)
    print(result)
```

```
if __name__ == "__main__":
    main()
```

References

You can find more information about the `pdb` module in the Python documentation.
https://docs.python.org/3/library/pdb.html

Launching Debugger From the Interactive Shell

- The following interactive shell demonstrates the use of the `pdb.run()` method to start the Python debugger.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> import simple_program
>>> pdb.run('simple_program.main()')
> <string>(1)<module>()
(Pdb)
```

- The preceding `pdb.run(simple_program.main()')` statement starts the debugger, and places the execution of the `main()` function under the control of the debugger.
- The prompt in the debugger is `(Pdb)`.
- You can use the `help` command to list the available commands.

Debugger Commands

The `step` command executes the current line, and stops at the first possible occasion. Either in a function call, or on the next line in the current function.

- The short version of the `step` command is `s`.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> import simple_program
>>> pdb.run('simple_program.main()')
> <string>(1)<module>()
(Pdb) step
--Call--
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(11)main()
-> def main():
(Pdb) s
```

```
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(12)main()
-> a = 5
(Pdb)
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(13)main()
-> b = 3
(Pdb)
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(14)main()
-> result = sum_args(a, b)
(Pdb)
--Call--
>
/home/student/AD141/AD141-apps/debugger/examples/simple_program.py(4)sum_args()
-> def sum_args(*args):
(Pdb)
```

Notice that you can press the enter key to reuse previous commands introduced in the debugger.

While the step command steps through every line of code, the `next` command works slightly differently.

The `next` command continues the execution of the application until one of the following circumstances occur:

- The debugger reaches the next line in the current function.
- The function returns.

The difference between `next` and `step` is that `step` stops inside a called function, while `next` executes called functions, only stopping at the next line in the current function.

- The short version of the `next` command is `n`.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> import simple_program
>>> pdb.run('simple_program.main()')
> <string>(1)<module>()
(Pdb) step
--Call--
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(11)main()
-> def main():
(Pdb) next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(12)main()
-> a = 5
(Pdb) next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(13)main()
-> b = 3
(Pdb) next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(14)main()
-> result = sum_args(a, b)
(Pdb) next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(15)main()
-> print(result)
```

```
(Pdb) next
8
--Return--
>
/home/student/AD141/AD141-apps/debugger/examples/simple_program.py(15)main()->None
-> print(result)
(Pdb) quit
>>>
```

Notice that the `next` command did not enter the `sum_args` method call, contrary to the preceding `step` example.

Listing Source

You can use the `list` command to:

- Display the source code around the current active line in the debugger.
- Display source code included within a provided range of lines.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> import simple_program
>>> pdb.run('simple_program.main()')
> <string>(1)<module>()
(Pdb) step
--Call--
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(11)main()
-> def main():
(Pdb) list
6           for value in args:
7               total += value
8           return total
9
10
11 ->       def main():
12          a = 5
13          b = 3
14          result = sum_args(a, b)
15          print(result)
16
(Pdb) q
>>> exit()
$
```

- The preceding `list` command displays 5 lines of code before, and after the currently active line.
- The → symbol indicates the active line.

The following example shows how you can use the `list` command with a range of lines.

```
(Pdb) list 11,13
11 ->       def main():
12          a = 5
13          b = 3
(Pdb)
```

Breakpoints

Developers can use breakpoints to stop the debugger on specific points in the code.

- You can use the `break` command to set a breakpoint.
  - The short version of the `break` command is `b`.
- The `continue` command automatically continues the program execution until the next breakpoint. You can use it instead of the `step` or `next` commands.

The following example sets a breakpoint on line 7 of the source code (`total += value`), and uses the `continue` command to jump to the next breakpoint.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> import simple_program
>>> pdb.run('simple_program.main()')
> <string>(1)<module>()
(Pdb) step
--Call-
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(11)main()
-> def main():
(Pdb) break 7
Breakpoint 1 at
/home/student/AD141/AD141-apps/debugger/examples/simple_program.py:7
(Pdb) continue
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(7)sum_args()
-> total += value
(Pdb) continue
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(7)sum_args()
-> total += value
(Pdb) continue
8
>>> exit()
$
```

In the preceding example, the last `continue` command causes the program to run to completion because there are no more breakpoints defined.

Evaluating the Current Context

One of the benefits of breakpoints is the ability to stop the debugger to check the values of variables with the currently running context.

You can use the p command to evaluate an expression in the current context, and print the expression value.

The following example prints the value of the variables within the `sum_args` function at the breakpoint assigned to line 7 of the code.

```
$ python3
Python 3.9.10 (main, Feb 9 2022, 00:00:00)
[GCC 11.2.1 20220127 (Red Hat 11.2.1-9)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import pdb
>>> import simple_program
>>> pdb.run('simple_program.main()')
> <string>(1)<module>()
(Pdb) step
--Call-
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(11)main()
-> def main():
(Pdb) break 7
Breakpoint 1 at /home/student/AD141/AD141-apps/debugger/examples/
simple_program.py:7
(Pdb) continue
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(7)sum_args()
-> total += value
(Pdb) p args, value, total
((5, 3), 5, 0)
(Pdb) continue
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(7)sum_args()
-> total += value
(Pdb) p args, value, total
((5, 3), 3, 5)
(Pdb) continue
8
>>> exit()
$
```

- You can use the `clear` command to clear all breakpoints set in the debugger.

Launching Debugger From the Command Line

It is also very common to run the pdb module from the command line by using the `-m` option.

```
$ python3 -m pdb simple_program.py
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(4)<module>()
-> def sum_args(args):
(Pdb) *next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(11)<module>()
-> def main():
(Pdb) next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(18)<module>()
-> if name == "main":
(Pdb) next
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(19)<module>()
```

```
-> main()
(Pdb) next
8
--Return--
> /home/student/AD141/AD141-apps/debugger/examples/
simple_program.py(19)<module>()->None
-> main()
(Pdb) next
--Return--
> <string>(1)<module>()->None
(Pdb) next
The program finished and will be restarted
> /home/student/AD141/AD141-apps/debugger/examples/simple_program.py(4)<module>()
-> def sum_args(args):
(Pdb) *q
$
```

The debugger in the preceding example, begins controlling the program at the point at which the program is loaded.

- This is different from the previous examples that always gained control of the program at the moment the `main()` method was about to be invoked.
- Also notice how the use of the `next` command does not enter either the `main` or the `sum_args` functions.
  - This is due to the previous examples using the following Python statement:

```
pdb.run('simple_program.main()')
```

Exercises

The solution files for these exercises are in the `AD141-apps` repository, within the `debugger/solutions` directory.

Exercise 1

Debug an example from previous chapters by starting the debugger from within the interactive Python shell.

- Step through the code by using a combination of the `step` and `next` commands to get used to their behavior.

Exercise 2

Use the `list` command with no arguments at some point when debugging the preceding code again. Display the code around the currently executing line.

- Repeat the process by passing a range large enough to the `list` command to display the entire file on the screen.

Exercise 3

Debug a program of your choice. Assign several breakpoints with the debugger by using the `b` command.

- Use the `continue` command to stop the execution at the breakpoints.
- Use the `p` command to print out the values of the variables available within the context of the current line.

Exercise 4

Repeat Exercise 3, but start the debugger from the command line by using the `-m` option.

References

You can find more information about the `pdb` module in the Python documentation.
https://docs.python.org/3/library/pdb.html

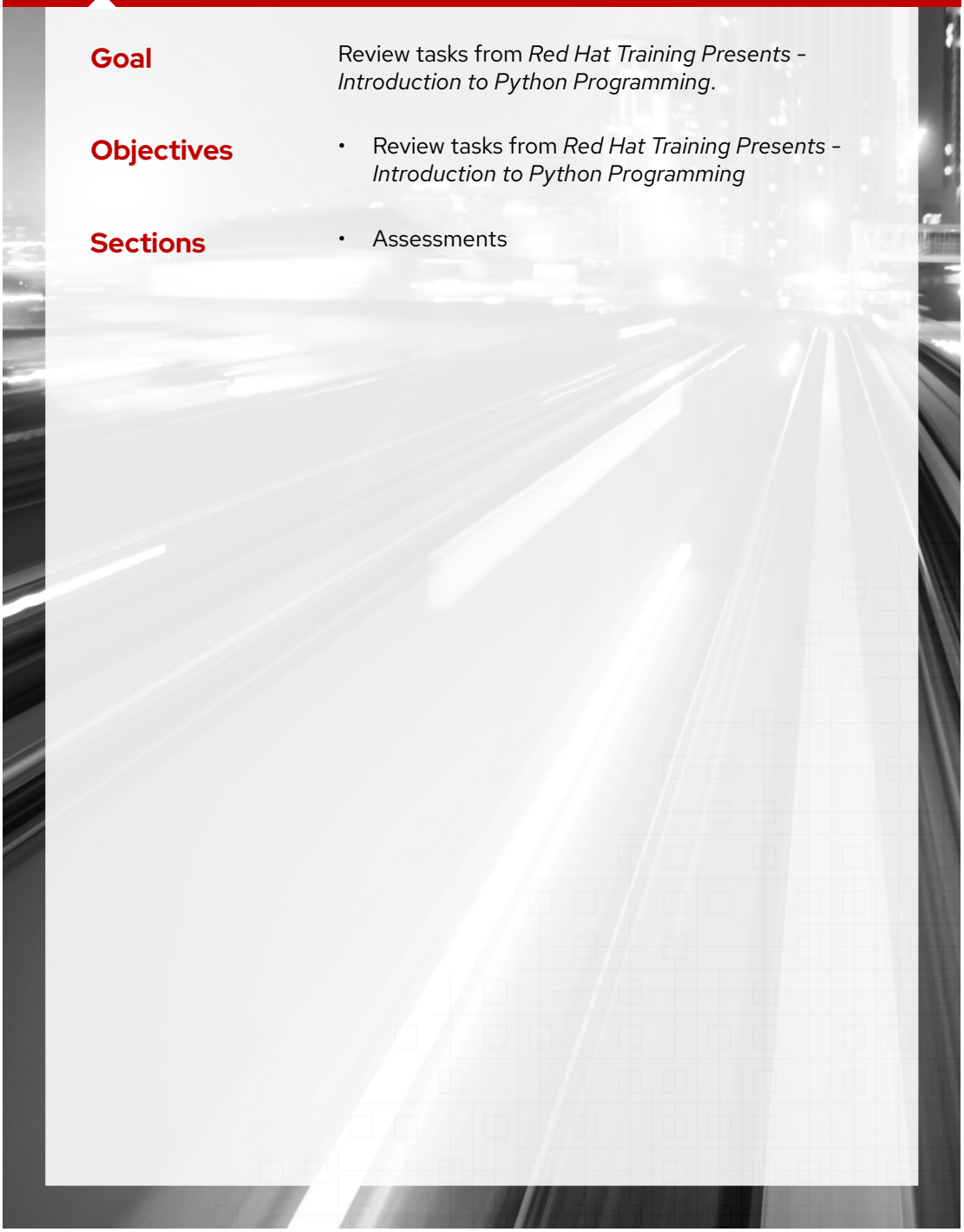
Summary

- You can use the `pdb` module to inspect and locate errors in your Python applications.
- You can perform interactive or post-mortem debugging sessions.

Chapter 13

Assessments

Goal	Review tasks from *Red Hat Training Presents - Introduction to Python Programming*.
Objectives	• Review tasks from *Red Hat Training Presents - Introduction to Python Programming*
Sections	• Assessments



Assessments

Objectives

After completing this section, you should be able to demonstrate the knowledge and skills learned in *Red Hat Training Presents - Introduction to Python Programming*.

Assessment 1: 99 Bottles of water

Create an application that sings the "99 bottles of water" song.

You must write a Python program that prints out all the verses of the song, exactly as the following lyrics show:

```
99 bottles of water on the wall!
99 bottles of water!
Take one down
And pass it around
98 bottles of water on the wall!

98 bottles of water on the wall!
98 bottles of water!
Take one down
And pass it around
97 bottles of water on the wall!

.
.
.

2 bottles of water on the wall!
2 bottles of water!
Take one down
And pass it around
1 bottle of water on the wall!

1 bottle of water on the wall!
1 bottle of water!
Take one down
And pass it around
No more bottles of water on the wall!
```

When developing your solution, consider the following tips:

- The same verse is repeated, each time with fewer bottles, until the bottle counter reaches zero.
- Note how the last two verses read `1 bottle` and not `1 bottles`, and `No more bottles` instead of `0 bottles`.

Assessment 2: High-Low Guessing Game

In the High-Low Guessing Game, the player attempts to guess an unknown number from 1 to 100. After each guess, the player is told whether their guess is too high (greater than the unknown number), or too low (less than the unknown number). If they guess the number, they win.

Write a program that plays this game with the user. The program should accept guesses, rejecting any outside the range of 1 to 100 and any non-integer value. The program should track how many valid and invalid guesses were made, and print out both numbers at the end.

```
$ ./guess.py
I'm thinking of a number from 1 to 100
Try to guess my number: 50
50 is too low - please guess again: 60
60 is too low - please guess again: Albequerque
Albequerque is not a valid guess - please guess again: 90
90 is too high - please guess again: 1776
1776 is not a valid guess - please guess again: 77
77 is correct!  You guessed my number in 4 guesses and made 2 invalid guesses.
$
```

Both the game and the application are over at this point. Playing again requires running the program again.

```
$ ./guess.py
I'm thinking of a number from 1 to 100
Try to guess my number: 12
12 is correct!  You guessed my number in 1 guess.
$
```

The program does not need to exactly follow the sample output above or below.

Summary

- Python is a high-level, interpreted, dynamically-typed programming language that relies on indentation to determine the control flow structure of your applications.
- You can use different statements and data structures to develop your application's logic.
- To organize your code into reusable pieces, you can create functions or modules.
- You can build software solutions that are closer to the problem domain by using object-oriented programming.