

Introduction to Python 3.8



with examples and
hands-on exercises

WEBUCATOR

Copyright © 2020 by Webucator. All rights reserved.

No part of this manual may be reproduced or used in any manner without written permission of the copyright owner.

Version: PYT138.1.1.2

Class Files

Download the class files used in this manual at

<https://www.webucator.com/class-files/index.cfm?versionId=4775>.

Errata

Corrections to errors in the manual can be found at <https://www.webucator.com/books/errata.cfm>.

Table of Contents

LESSON 1. Python Basics.....	1
Getting Familiar with the Terminal.....	2
Running Python.....	5
Running a Python File.....	7
📄 Exercise 1: Hello, world!	9
Literals.....	10
Python Comments.....	10
Data Types.....	11
📄 Exercise 2: Exploring Types	13
Variables.....	14
📄 Exercise 3: A Simple Python Script	17
Constants.....	18
Deleting Variables.....	18
Writing a Python Module.....	18
print() Function.....	21
Collecting User Input.....	23
📄 Exercise 4: Hello, You!	25
Reading from and Writing to Files.....	26
📄 Exercise 5: Working with Files	28
LESSON 2. Functions and Modules.....	31
Defining Functions.....	31
Variable Scope.....	34
Global Variables.....	36
Function Parameters.....	37
📄 Exercise 6: A Function with Parameters	41
Default Values.....	42
📄 Exercise 7: Parameters with Default Values	43
Returning Values.....	44
Importing Modules.....	45
Methods vs. Functions.....	48

LESSON 3. Math.....	49
Arithmetic Operators.....	49
📄 Exercise 8: Floor and Modulus.....	53
Assignment Operators.....	54
Precedence of Operations.....	55
Built-in Math Functions.....	56
The math Module.....	60
The random Module.....	62
📄 Exercise 9: How Many Pizzas Do We Need?.....	65
📄 Exercise 10: Dice Rolling.....	67
LESSON 4. Python Strings.....	69
Quotation Marks and Special Characters.....	70
String Indexing.....	74
📄 Exercise 11: Indexing Strings.....	75
Slicing Strings.....	76
📄 Exercise 12: Slicing Strings.....	78
Concatenation and Repetition.....	80
📄 Exercise 13: Repetition.....	82
Combining Concatenation and Repetition.....	84
Python Strings are Immutable.....	85
Common String Methods.....	86
String Formatting.....	91
📄 Exercise 14: Playing with Formatting.....	100
Formatted String Literals (f-strings).....	101
Built-in String Functions.....	103
📄 Exercise 15: Outputting Tab-delimited Text.....	105

LESSON 5. Iterables: Sequences, Dictionaries, and Sets.....	111
Definitions.....	111
Sequences.....	112
Lists.....	112
Sequences and Random.....	116
📄 Exercise 16: Remove and Return Random Element.....	117
Tuples.....	118
Ranges.....	121
Converting Sequences to Lists.....	122
Indexing.....	122
📄 Exercise 17: Simple Rock, Paper, Scissors Game.....	124
Slicing.....	126
📄 Exercise 18: Slicing Sequences.....	128
min(), max(), and sum().....	130
Converting Sequences to Strings with str.join(seq).....	131
Splitting Strings into Lists.....	131
Unpacking Sequences.....	134
Dictionaries.....	134
The len() Function.....	142
📄 Exercise 19: Creating a Dictionary from User Input.....	143
Sets.....	145
*args and **kwargs.....	146
LESSON 6. Virtual Environments, Packages, and pip.....	149
📄 Exercise 20: Creating, Activating, Deactivating, and Deleting a Virtual Environment....	150
Packages with pip.....	152
📄 Exercise 21: Working with a Virtual Environment.....	154

LESSON 7. Flow Control.....	159
Conditional Statements.....	159
Compound Conditions.....	163
The is and is not Operators.....	164
all() and any().....	165
Ternary Operator.....	165
In Between.....	166
Loops in Python.....	166
📄 Exercise 22: All True and Any True	174
break and continue.....	176
Looping through Lines in a File.....	178
📄 Exercise 23: Word Guessing Game	182
📄 Exercise 24: for...else	193
The enumerate() Function.....	196
Generators.....	197
List Comprehensions.....	208
LESSON 8. Exception Handling.....	215
Exception Basics.....	215
Wildcard except Clauses.....	217
Getting Information on Exceptions.....	218
📄 Exercise 25: Raising Exceptions	219
The else Clause.....	222
The finally Clause.....	223
Using Exceptions for Flow Control.....	224
📄 Exercise 26: Running Sum	226
Raising Your Own Exceptions.....	228
LESSON 9. Python Dates and Times.....	231
Understanding Time.....	231
The time Module.....	233
Time Structures.....	236
Times as Strings.....	240
Time and Formatted Strings.....	241
Pausing Execution with time.sleep().....	242
The datetime Module.....	244
datetime.datetime Objects.....	248
📄 Exercise 27: What Color Pants Should I Wear?	251
datetime.timedelta Objects.....	252
📄 Exercise 28: Report on Departure Times	254

LESSON 10. File Processing.....	261
Opening Files.....	261
📄 Exercise 29: Finding Text in a File	265
Writing to Files.....	269
📄 Exercise 30: Writing to Files	271
📄 Exercise 31: List Creator	273
The os Module.....	277
Walking a Directory.....	282
The os.path Module.....	284
A Better Way to Open Files.....	287
📄 Exercise 32: Comparing Lists	291
LESSON 11. PEP8 and Pylint.....	295
PEP8.....	295
Pylint.....	301

LESSON 1

Python Basics

Topics Covered

- ✓ How Python works.
- ✓ Python's place in the world of programming languages.
- ✓ Python literals.
- ✓ Python comments.
- ✓ Variables and Python data types.
- ✓ Simple modules.
- ✓ Outputting data with `print()`.
- ✓ Collecting user input.

The pythons had entered into Mankind. No man knew at what moment he might be Possessed!

– *Plague of Pythons, Frederik Pohl*

Introduction

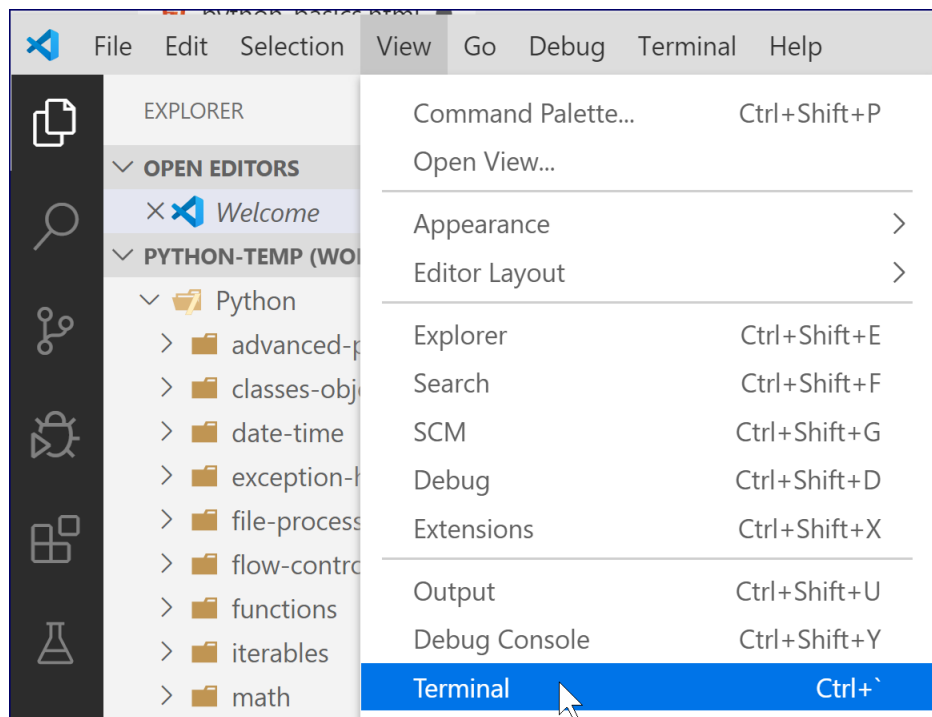
Python, which first appeared in 1991, is one of the most popular programming languages used today.¹ Python is a high-level programming language, meaning that it uses a syntax that is relatively human readable, which gets translated by a Python Interpreter into a language your computer can understand. Examples of other popular high-level programming languages are C#, Objective-C, Java, PHP, and JavaScript. Interestingly, all of these other languages, unlike Python, share a C-like syntax. If you use one or more of those languages, you may find Python's syntax a little strange. But give it a little time. You'll find it's quite programmer friendly.

1. <https://pypl.github.io/PYPL.html>

Getting Familiar with the Terminal

Python developers need to be comfortable navigating around and running commands at the terminal.² We'll walk you through some basics:

1. Open a terminal. In Visual Studio Code, you can open a terminal by selecting **Terminal** from the **View** menu:



You can also open a terminal by pressing **Ctrl+`**. The **`** key is on the upper left of most keyboards. It looks like this:



The terminal will open at the root of your Visual Studio Code workspace. If you're working in the workspace we had you set up, that should be in a **Webucator\Python** directory (the

2. The generic term for the various terminals is *command line shell* ([https://en.wikipedia.org/wiki/Shell_\(computing\)](https://en.wikipedia.org/wiki/Shell_(computing))). Visual Studio Code will select an appropriate terminal for you. On Windows, that will most likely be **Command Prompt** or **PowerShell**. On a Mac, it will most likely be **bash** or **Zsh**. The names “prompt,” “command prompt,” “shell,” and “terminal” are used interchangeably.

words *folder* and *directory* are used interchangeably). The prompt on Windows will read something like:

```
PS C:\Webucator\Python>
```

On a Mac, it will show some combination of your computer name, the directory you are in, and your username, followed by a \$ or % sign. For example:

```
NatsMac:Python natdunn$
```

```
natdunn@NatsMac:Python %
```

2. Use `cd` to **change directories**. From the `Webucator\Python` directory, run:

```
PS C:\Webucator\Python> cd python-basics
```

```
PS ..\Python\python-basics>
```

Your prompt now shows that you are in the `python-basics` directory.

3. Move up to the parent directory by running:

```
cd ..
```

You will now be back in your `Python` directory.

4. Run:

```
cd python-basics/Demos
```

Your prompt will show that you are in the `Demos` directory. Depending on your environment, it may also show one or more directories above the `Demos` directory. To get the full path to your current location, run `pwd` for **p**resent **w**orking **d**irectory:

```
pwd
```

On Windows, that will look something like this:

```
PS ..\python-basics\Demos> pwd
```

```
Path
```

```
----
```

```
C:\Webucator\Python\python-basics\Demos
```

On a Mac, it will look something like this:

```
natdunn@NatsMac Demos % pwd
/Users/natdunn/Documents/Webucator/Python/python-basics/Demos
```

5. Run `cd ..` to back up to the `python-basics` directory.
6. Type `cd De` and then press the **Tab** key. On Windows, you should see something like this:

```
PS ..\Python\python-basics> cd .\Demos\
```

The “.” at the beginning of the path represents the *current* (or *present*) directory. So, `.\Demos` refers to the `Demos` directory within the current directory. A Mac won’t include the current directory in the path. When you press **Tab**, it will just fill out the rest of the folder name.

```
cd Demos
```

Press **Enter** to run the command. Then run `cd ..` to move back up to the `python-basics` directory.

7. Each of our lesson folders will contain `Demos`, `Exercises`, and `Solutions` folders. Some may contain additional folders. To see the contents of the current directory, run `dir` on Windows or `ls` on Mac/Linux:³

Windows PowerShell

```
PS ..\Python\python-basics> dir
```

Directory: C:\Webucator\Python\python-basics

Mode	LastWriteTime	Length	Name
d-----	2/18/2020 6:13 AM		data
d-----	2/18/2020 6:13 AM		Demos
d-----	2/18/2020 5:09 AM		Exercises
d-----	2/18/2020 6:13 AM		Solutions

Mac Terminal

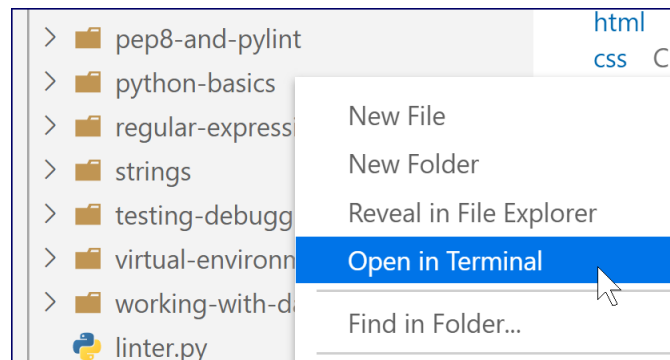
```
NatsMac:python-basics natdunn$ ls
Demos      Exercises  Solutions  data
```

3. The `ls` command works in Windows PowerShell as well.

8. Play with switching between directories using the `cd` command until you feel comfortable navigating the terminal.

Visual Studio Shortcut

Visual Studio provides a shortcut for opening a specific directory at the terminal. Simply right-click on the directory in the **Explorer** panel and select **Open in Terminal**:



Running Python

Python runs on Microsoft Windows, Mac OS X, Linux, and other Unix-like systems. The first thing to do is to make sure you have a recent version of Python installed:

1. Open the terminal in Visual Studio Code.
2. Run `python -V`:

```
PS C:\Webucator\Python> python -V
Python 3.8.1
```

If you have Python 3.6 or later, you are all set.

If you do not have Python 3.6 or later installed, download it for free at <https://www.python.org/downloads>. After running through the installer, run `python -V` at the terminal again to make sure Python installed correctly.

Python Versions on Macs

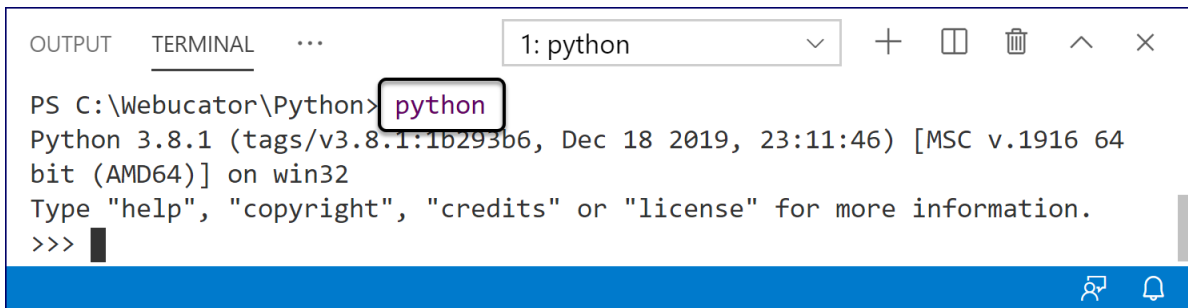
Your Mac will likely have a version of Python 2 already installed. After you install Python 3, you may find that running `python -V` still shows the Python 2 version. In that case, try running

`python3 -V`. That should output the version of Python 3 that you have. If it does, then you should use the `python3` command instead of the `python` command to run Python 3.

If you would prefer to be able to use the `python` command for Python 3 (and who wouldn't), visit <https://www.webucator.com/blog/2020/02/mapping-python-to-python-3-on-your-mac/> to see how you can map `python` to `python3`.

❖ Python Interactive Shell

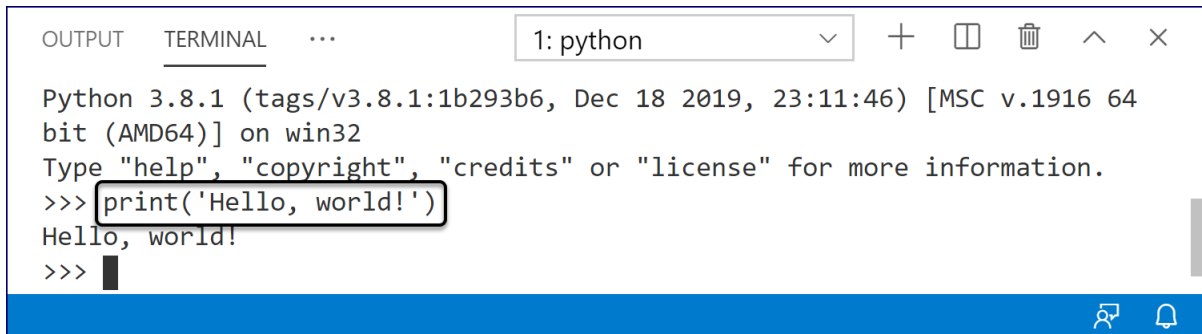
You can run Python in *Interactive Mode* by running `python` at the terminal:



```
OUTPUT  TERMINAL  ...  1: python
PS C:\Webucator\Python> python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This will open up the *Python shell*, at which you can run Python commands. For example, to print out “Hello, world!”, you would run:

```
print('Hello, world!')
```



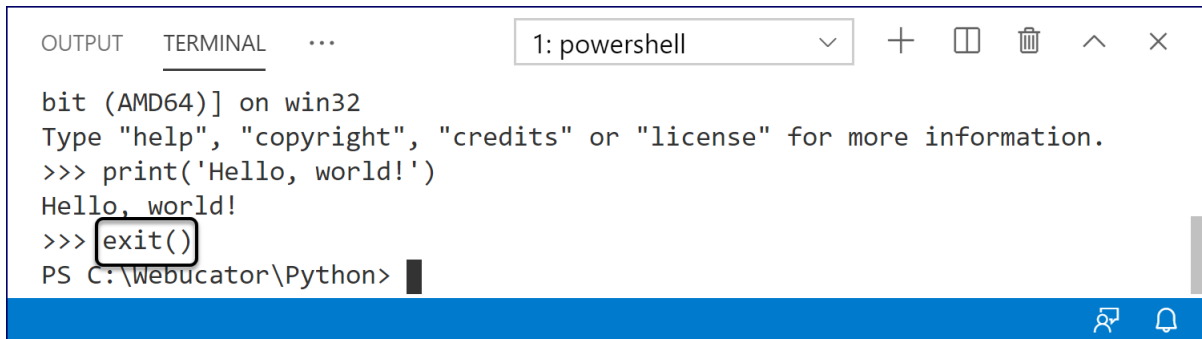
```
OUTPUT  TERMINAL  ...  1: python
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello, world!')
Hello, world!
>>>
```

You can tell that you are at the Python prompt by the three right-angle brackets:

```
>>>
```

To exit the Python shell, run:

```
exit()
```

A screenshot of a Windows PowerShell terminal window. The title bar shows '1: powershell'. The terminal content shows a Python session: 'bit (AMD64)] on win32', 'Type "help", "copyright", "credits" or "license" for more information.', '>>> print('Hello, world!')', 'Hello, world!', '>>> exit()' (where 'exit()' is highlighted with a red box), and 'PS C:\Webucator\Python>'. The terminal has a blue background and a white foreground. There are icons for window management and search in the top right corner.

Now, you are back at the regular prompt.

Running a Python File

While working at the Python shell can be useful in some scenarios, you will often be writing Python files, so you can save, re-run, and share your code. Let's get started with a simple "Hello, world!" demo using your editor. We will open a *script*, which is simply a file with a .py extension that contains Python code. After the demonstration, you will add another line of code to the script in an exercise.

Here is the script we are going to run:

Demo 1: python-basics/Demos/hello_world.py

```
1. # Say Hello to the World
2. print("Hello, world!")
```

Code Explanation

The `print()` function simply outputs content either to standard output (e.g., the terminal) or to a file if specified.

To run this code:

1. Open the terminal at `python-basics/Demos`.
2. Run `python hello_world.py`
3. The "Hello, world!" message will be displayed.

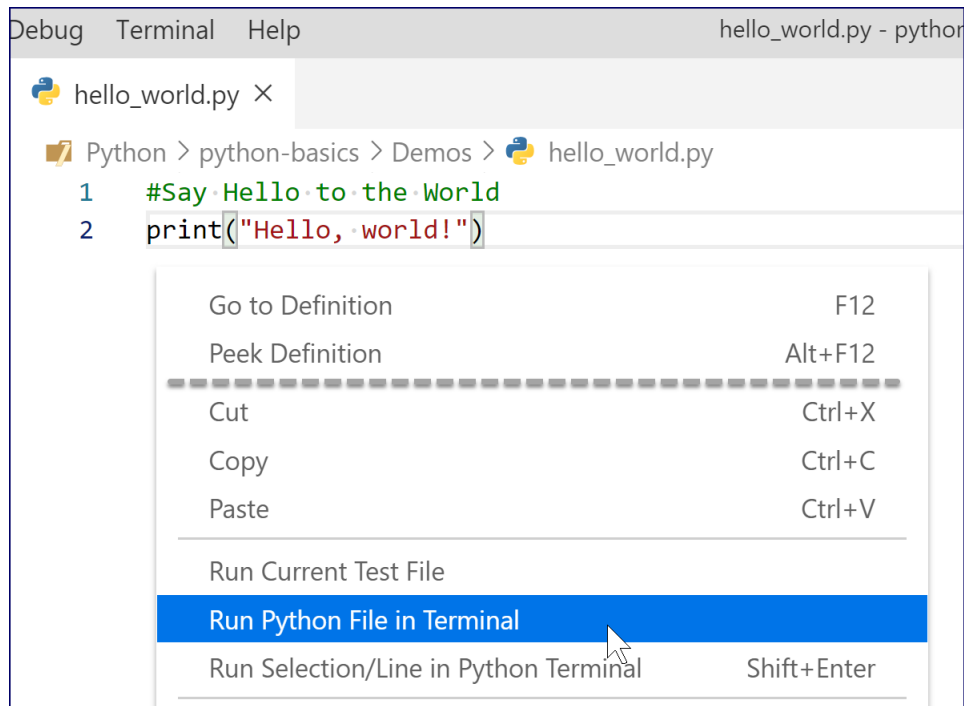
Here it is in the terminal:

```
OUTPUT  DEBUG CONSOLE  TERMINAL

PS ...\\python-basics\\Demos> python hello_world.py
Hello, world!
```

Right-click and Run

Another way to run a file from within Visual Studio Code is to right-click on the open file and select **Run Python File at Terminal**:



If you don't see this option, then you don't have the Python extension installed.

Exercise 1: Hello, world!

 5 to 10 minutes

1. Open `python-basics/Demos/hello_world.py` in your editor.
2. Add the following line after the “Hello, world!” line:

```
print("Hello again, world!")
```

3. Save your changes.
4. Run the Python file just as you did earlier for the demonstration.
5. The output should look like this:

```
Hello, world!  
Hello again, world!
```

Solution: python-basics/Solutions/hello_again_world.py

```
1.  # Say Hello to the World (twice!)
2.  print("Hello, world!")
3.  print("Hello again, world!")
```

Code Explanation

The extra line of code will cause a second message to be printed to the standard output.

Literals

When a value is hard coded into an instruction, as "Hello" is in `print("Hello")`, that value is called a *literal* because the Python interpreter should not try to interpret it but should take it *literally*. If we left out the quotation marks (i.e., `print(Hello)`), Python would output an error because it would not know how to interpret the name `Hello`.

Either single quotes or double quotes can be used to create *string* literals. Just make sure that the open and close quotation marks match.

Literals representing numbers (e.g., 42 and 3.14) are not enclosed in quotation marks.

Python Comments

In the previous demo, you may have noticed this line of code:

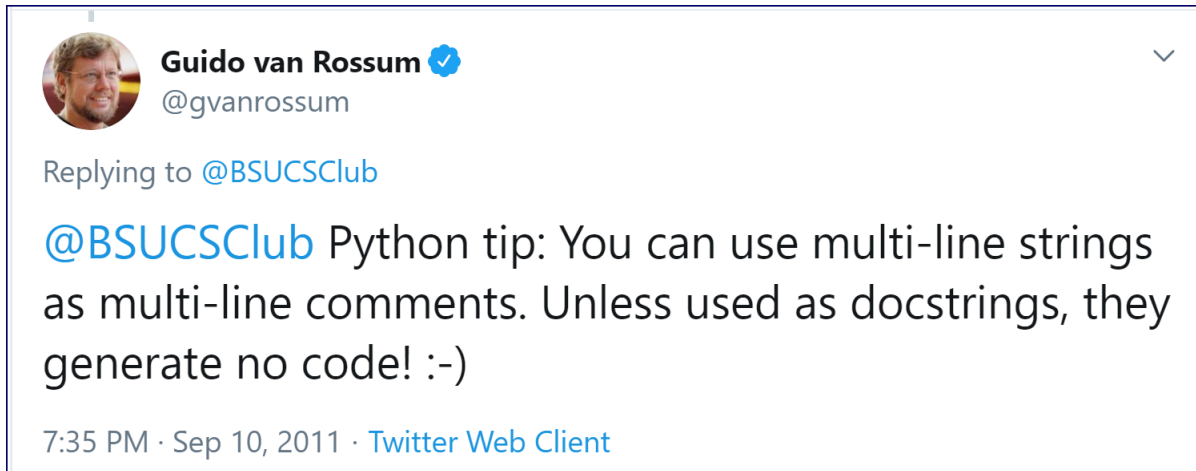
```
# Say Hello to the World
```

That number sign (or hash or pound sign) indicates a comment. Everything that trails it on the same line will be ignored.

❖ Multi-line Comments

There is no official syntax for creating multi-line comments; however, Guido van Rossum, the creator of Python, tweeted this tip⁴ as a workaround:

4. <https://twitter.com/gvanrossum/status/112670605505077248?lang=en>



Multi-line strings are created with triple quotes, like this:

```
"""This is a
very very helpful and informative
comment about my code!"""
```

Because multi-line strings generate no code, they can be used as pseudo-comments. In certain situations, these pseudo-comments can get confused with *docstrings*, which are used to auto-generate Python documentation, so we recommend you avoid using them until you become familiar with docstrings. Instead, use:

```
# This is a
# very very helpful and informative
# comment about my code!
```

Data Types

In Python programming, objects have different *data types*. The data type determines both what an object can do and what can be done to it. For example, an object of the data type *integer* can be subtracted from another integer, but it cannot be subtracted from a *string* of characters.

In the following list, we show the basic data types used in Python. Abbreviations are in parentheses.

1. boolean (bool) – A True or False value.
2. integer (int) – A whole number.
3. float (float) – A decimal.

4. `string (str)` – A sequence of Unicode⁵ characters.
5. `list (list)` – An ordered sequence of objects, similar to an array in other languages.
6. `tuple (tuple)` – A sequence of fixed length, in which an element's position is meaningful.
7. `dictionary (dict)` – An unordered grouping of key-value pairs.
8. `set (set)` – An unordered grouping of values.

We will cover all of these data types in detail.

5. Unicode is a 16-bit character set that can handle text from most of the world's languages.



Exercise 2: Exploring Types



10 to 15 minutes

In this exercise, you will use the built-in `type()` function to explore different data types.

1. Open the terminal.
2. Start the Python shell by typing `python` and then pressing **Enter**:

```
PS ..\python-basics\Demos> python
```

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit  
(AMD64)] on win32
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

You are now in the Python shell.

3. To check the type of an object, use the `type()` function. For example, `type(3)` will return `<class 'int'>`:

```
>>> type(3)
```

```
<class 'int'>
```

4. Find the types of all of the following:

A. 3

B. 3.1

C. '3'

D. 'pizza'

E. True

F. ('1', '2', '3')

G. ['1', '2', '3']

H. {'1', '2', '3'}

Solution

When you're done, the output should appear as follows:

```
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type('3')
<class 'str'>
>>> type('pizza')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(('1', '2', '3'))
<class 'tuple'>
>>> type(['1', '2', '3'])
<class 'list'>
>>> type({'1', '2', '3'})
<class 'set'>
```

Don't worry if you're not familiar with all of the preceding data types. We will cover them all.

Class and Type

You may wonder at Python's use of the word "class" when outputting a data type. In Python, "class" and "type" mean the same thing.

Variables

Variables are used to hold data in memory. In Python, variables are untyped, meaning you do not need to specify the data type when creating the variable. You simply assign a value to a variable. Python determines the type by the value assigned.

❖ Variable Names

Variable names are case sensitive, meaning that `age` is different from `Age`. By convention, variable names are written in all lowercase letters and words in variable names are separated by underscores

(e.g., `home_address`). Variable names must begin with a letter or an underscore and may contain only letters, digits, and underscores.

Keywords

The following list of keywords have special meaning in Python and may not be used as variable names:

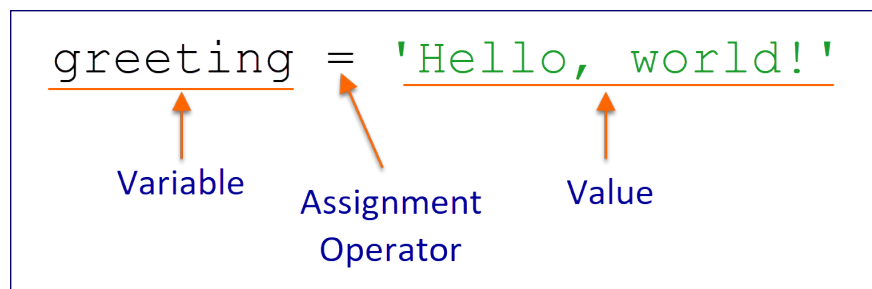
Python Keywords

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

❖ Variable Assignment

There are three parts to variable assignment:

1. Variable name.
2. Assignment operator.
3. Value assigned. This could be any data type.



Here is the “Hello, world!” script again, but this time, instead of outputting a literal, we assign a string to a variable and then output the variable:

Demo 2: python-basics/Demos/hello_variables.py

```
1. greeting = "Hello, world!"
2. print(greeting)
```

Code Explanation

Run this Python file at the terminal. The output will be the same as it was in the previous demo (see page 7):

```
Hello, world!
```

❖ Simultaneous Assignment

A very cool feature of Python is that it allows for simultaneous assignment. The syntax is as follows:

```
var_name1, var_name2 = value1, value2
```

This can be useful as a shortcut for assigning several values at once, like this:

```
>>> first_name, last_name, company = "Nat", "Dunn", "Webucator"
>>> first_name
'Nat'
>>> last_name
'Dunn'
>>> company
'Webucator'
```




Exercise 3: A Simple Python Script



5 to 10 minutes

In this exercise, you will write a simple Python script from scratch. The script will create a variable called `today` that stores the day of the week.

1. In your editor create a new file and save it as `today.py` in the `python-basics/Exercises` folder.
2. Create a variable called `today` that holds the current day of the week as literal text (i.e., in quotes).
3. Use the `print()` function to output the variable value.
4. Save your changes.
5. Test your solution.

Solution: `python-basics/Solutions/today.py`

```
1. today = "Monday"
2. print(today)
```

Constants

In programming, a constant is like a variable in that it is an identifier that holds a value, but, unlike variables, constants are not variable, they are constant. Good name choices, right?

Python doesn't really have constants, but as a convention, variables that are meant to act like constants (i.e., their values are not meant to be changed) are written in all capital letters. For example:

```
PI = 3.141592653589793
RED = "FF0000" # hexadecimal code for red
```

Deleting Variables

Although it's rarely necessary to do so, variables can be deleted using the `del` statement, like this:

```
>>> a = 1
>>> print(a)
1
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Notice that trying to print `a` results in an error, because after `del a`, the `a` variable no longer exists.

Writing a Python Module

A Python module is simply a file that contains code that can be reused. The `today.py` file is really a module, albeit a very simple one. A module can be run by itself or as part of a larger program. It is too early to get into all the aspects of code reuse and modular programming, but you want to start with good habits, one of which is to use a `main()` function in your programs.

❖ The main() Function

Let's look at the basic syntax of a function. A function is created using the `def` keyword like this:

Demo 3: `python-basics/Demos/indent_demo.py`

```
1. def main():
2.     print("I am part of the function.")
3.     print("I am also part of the function.")
4.     print("Hey, me too!")
5. print("Sad not to be part of the function. I've been outdented.")
6.
7. main()
```

Code Explanation

Running this Python file will render the following:

```
Sad not to be part of the function. I've been outdented.
I am part of the function.
I am also part of the function.
Hey, me too!
```

Notice that the first line of output is the line that is not part of the function. That is because the function does not run until it is called, and it is called after the `print()` function that outputs “Sad not to be part of the function. I’ve been outdented.”

The code is read by the Python interpreter from top to bottom, but the function definition just defines the function; it does not *invoke the function* (programmer speak for “call the function to execute”).

Definition of main () function. Function is not executed until called.

```
1 def main():
2     print("I am part of the function.")
3     print("I am also part of the function.")
4     print("Hey, me too!")
5 print("Sad not to be part of the function. I've been outdented.")
6
7 main()
```

Call to print () function.

Call to main () function.

Visit <http://bit.ly/pythontutor-indentdemo> to see this demo in pythontutor.com⁶, a web application for visualizing how Python executes code.

The screenshot displays the Python Tutor interface for Python 3.6. At the top, a code editor shows a function definition:

```
1 def main():
2     print("I am part of the function.")
3     print("I am also part of the function.")
4     print("Hey, me too!")
5     print("Sad not to be part of the function. I've been outdented.")
6
7 main()
```

A green arrow points to line 7, indicating the next line to execute. A legend below the code explains the arrows: a green arrow for "line that just executed" and a red arrow for "next line to execute".

Below the code editor is a progress bar and navigation buttons: "<< First", "< Prev", "Next >", and "Last >>". The progress bar indicates "Step 4 of 8".

The "Print output" section shows the output of the current step: "Sad not to be part of the function. I've been outdented.".

The "Frames" and "Objects" sections show the current state of the program. The "Frames" section lists the "Global frame" and the "main" frame. The "Objects" section shows the "function main()". An arrow points from the "main" frame to the "function main()" object.

A few things to note about functions:

6. <http://www.pythontutor.com/visualize.html>

1. Functions are created using the `def` keyword. The content that follows the `def` keyword on the same line is called the *function signature*.
2. The convention for naming functions is the same as that for variables: use all lowercase letters and separate words with underscores.
3. In the function definition, the function name is followed by a pair of parentheses, which may contain parameters (more on that soon), and a colon.
4. The contents of the function starts on the next line and must be indented. Either spaces or tabs can be used for indenting, but spaces are preferred.
5. The first line of code after the function definition that is not indented is not part of the function and effectively marks the end of the function definition.
6. Functions are invoked using the function name followed by the parentheses (e.g., `indent_demo()`).

There is nothing magic about the name “main”. It is simply the name used by convention for the function that starts the program or module running.⁷

Grouping of Statements

A programming *statement* is a unit of code that does something. The following code shows two statements:

```
greeting = "Hello!"  
print(greeting)
```

In Python, statements are grouped using indenting. As we just saw with the `main()` function, lines that are indented below the function signature are part of the function. It is important to understand this: *changes in indentation level denote code groups*. You must be careful with your indenting.

`print()` Function

You have already seen how to output data using the built-in `print()` function. Now, let’s see how to output a variable and a literal string together. The `print()` function can take multiple arguments. By

7. Although only a convention, the name “main()” was not chosen arbitrarily. It is used because modules refer to themselves as “__main__”, so it seems fitting to get them started with a corresponding “main()” function.

default, it will output them all separated by single spaces. For example, the following code will output "H e l l o !"

```
print('H', 'e', 'l', 'l', 'o', '!')
```

This functionality allows for the combination of literal strings and variables as shown in the following demo:

Demo 4: python-basics/Demos/variable_and_string_output.py

```
1. def main():
2.     today = "Monday"
3.     print("Today is", today, ".")
4.
5. main()
```

Code Explanation

Run the Python file. It should render the following:

```
Today is Monday .
```

Notice the extra space before the period in the output of the last demo:

Today is Monday .

We'll get rid of that soon.

❖ Named Arguments

As we have seen with `print()`, functions can take multiple arguments. These arguments can be *named* or *unnamed*. To illustrate, let's look at some more arguments the `print()` function can take:

```
print('H', 'e', 'l', 'l', 'o', '!', sep=' ', end='\n')
```

Those last two arguments are **named arguments**.

- `sep` is short for “separator.” It specifies the character that separates the list of objects to output. The default value is a single space, so specifying `sep=" "` doesn’t change the default behavior at all.
- `end` specifies the character to print at the very end (i.e., after the last printed object). The default is a newline character (denoted with `\n`). You can use an empty string (e.g., `' '`) to specify that nothing should be printed at the end.

The following demo shows how the `sep` argument can be used to get rid of the extra space we saw in the previous example:

Demo 5: `python-basics/Demos/variable_and_string_output_fixed_spacing.py`

```
1. def main():
2.     today = "Monday"
3.     print("Today is ", today, ".", sep="")
4.
5. main()
```

Code Explanation

Run the Python file. It should render the following:

```
Today is Monday.
```

Collecting User Input

Functions may or may not return values. The `print()` function, for example, does not return a value.

Python provides a built-in `input()` function, which takes a single argument: the prompt for the user’s input. Unlike `print()`, the `input()` function *does* return a value: the input from the user as a string. The following demo shows how to use it to prompt the user for the day of the week.

Demo 6: `python-basics/Demos/input.py`

```
1. def main():
2.     today = input("What day is it? ")
3.     print("Wow! Today is ", today, "? Awesome!", sep="")
4.
5. main()
```

Code Explanation

Run the Python file. It should immediately prompt the user:

```
What day is it?
```

Enter the day and press **Enter**. It will output something like:

```
Wow! Today is Monday? Awesome!
```

The full output will look like this:

```
What day is it? Monday  
Wow! Today is Monday? Awesome!
```



Exercise 4: Hello, You!



5 to 10 minutes

In this exercise, you will greet the user by name.

1. Open a new script. Save it as `hello_you.py` in `python-basics/Exercises`.
2. Write code to prompt for the user's name.
3. After the user has entered their name, output a greeting.
4. Save your changes.
5. Test your solution.

Solution: python-basics/Solutions/hello_you.py

```
1. def main():
2.     your_name = input("What is your name? ")
3.     print("Hello, ", your_name, "!", sep="")
4.
5. main()
```

Code Explanation

The code should work like this:

```
PS ..\python-basics\Solutions> python hello_you.py
What is your name? Nat
Hello, Nat!
```

Reading from and Writing to Files

To built-in `open()` method is used to open a file from the file system. We will cover this in the **File Processing** lesson (see page 261). For now, you just need to know how to read from a file and how to write to a file.

❖ Reading from a File

The following code shows the steps to:

1. Open a file and assign the file to a *file handler*.
2. Read the content of the file into a variable.
3. Print that variable.
4. Close the file.

```
f = open("my-file.txt") # Open my-file.txt and assign result to f.
content = f.read() # Read contents of file into content variable.
print(content) # Print content.
f.close() # Close the file.
```

Because we referenced the file name directly, Python will look in the current directory for the file. If the file is located in a different directory, you must provide the path, either as an absolute or relative path.⁸

with Blocks

It is important to close the file to free up the memory space the handler is taking up. It's also easy to forget to do so. Fortunately, Python provides a structure that makes explicitly closing the file unnecessary:

```
with open("my-file.txt") as f:
    content = f.read()
    print(content)
```

When Python reaches the end of the `with` block, it understands that the file is no longer necessary and automatically closes it.

❖ Writing to a File

In addition to the path to the file, the `open()` function takes a second parameter: `mode`, which indicates whether the file is being opened for reading ("`r`"), writing ("`w`"), or appending ("`a`"). The default value for `mode` is "`r`", which is why we didn't need to pass a value in when opening the file for reading. If we want to write to a file, we do have to pass in a value: "`w`".

```
with open("my-file2.txt", "w") as f:
    f.write("Hello, world!!!!")
```

When you run the code above, it will open `my-file2.txt` if it exists and overwrite the file with the text you write to it. If it doesn't find a file named `my-file2.txt`, it will create it.

8. Absolute paths start from the top of the file system and work their way downwards towards the referenced file. Relative paths start from the current location (the location of the file containing the path) and work their way to the referenced file from that location.

Exercise 5: Working with Files

 10 to 15 minutes

In this exercise, you will open two files that contain lists of popular boys and girls names from 1880,⁹ read their contents into two variables, and then write the combined content of the two files into a new file.

1. Open a new script. Save it as `files.py` in `python-basics/Exercises`.
2. Write code to open `python-basics/data/1880-boys.txt` and read its contents into a variable called `boys`.
3. Write code to open `python-basics/data/1880-girls.txt` and read its contents into a variable called `girls`.
4. Write code to open a new file named `1880-all.txt` in the `python-basics/data` folder and write the combined contents of the `boys` and `girls` variables into the file. Note that you can combine the two strings like this:

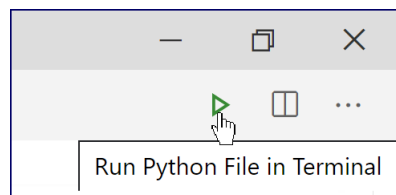
```
boys + "\n" + girls
```

That will place one newline between the content in `boys` and the content in `girls`.

5. Save your file.
6. Test your solution. When you run it, it should create the new file. Look in the `data` folder for the `1880-all.txt` file. Does it exist? If so, open it up. Does it have a list of all the boys and girls names?

Run Python File in Terminal

You may have discovered that you can run Python files in the terminal using the green triangle in the upper-right of Visual Studio Code:



⁹. The lists of names come from <https://www.ssa.gov/oact/babynames/>.

By default, VS Code will run the file from the Workspace root folder and search for any files being referenced with relative paths from that folder. As a result, you may get `FileNotFoundError` errors. You can fix this by changing a setting:

1. From the **File** menu, select **Preferences > Settings**.
2. Search for “execute in file dir”.
3. Check the **Python > Terminal: Execute in File Dir** setting.

Python > Terminal: Execute In File Dir

☒ When executing a file in the terminal, whether to use execute in the file's directory, instead of the current open folder.

Solution: python-basics/Solutions/files.py

```
1.  with open("../data/1880-boys.txt") as f_boys:
2.      boys = f_boys.read()
3.
4.  with open("../data/1880-girls.txt") as f_girls:
5.      girls = f_girls.read()
6.
7.  with open("../data/1880-all.txt", "w") as f:
8.      f.write(boys + "\n" + girls)
```

Conclusion

In this lesson, you have begun to work with Python. Among other things, you have learned to use variables, to output data, to collect user input, and to write simple Python functions and modules.

LESSON 2

Functions and Modules

Topics Covered

- ☑ Defining and calling functions.
- ☑ Parameters and arguments.
- ☑ Default values for parameters.
- ☑ Variable scope.
- ☑ Return values.
- ☑ Creating and importing modules.

It is your duty as a magistrate, and I believe and hope that your feelings as a man will not revolt from the execution of those functions on this occasion.

– *Frankenstein, Mary Shelley*

Introduction

You have seen some of Python's built-in functions. In this lesson, you will learn to write your own.

Defining Functions

We discussed functions a little in the last lesson, but let's quickly review the syntax. Functions are defined in Python using the `def` keyword. The syntax is as follows:

```
def function_name():
    # content of function is indented
    do_something()
# This is no longer part of the function
do_something_else()
```

Here is a modified solution to the “Hello, You!” exercise:

Demo 7: functions/Demos/hello_you.py

```
1.  def say_hello():
2.      your_name = input("What is your name? ")
3.      print("Hello, ", your_name, "!", sep=" ")
4.
5.  def main():
6.      say_hello()
7.
8.  main()
```

Code Explanation

The code works in the same way, but the meat of the program has been moved out of the `main()` function and into another function. This is common. Usually, the `main()` function handles the flow of the program, but the actual “work” is done by other functions in the module.

The following expanded demo further illustrates how the `main()` function can be used to control flow.

Demo 8: functions/Demos/hello_you_expanded.py

```
1.  def say_hello():
2.      your_name = input("What is your name? ")
3.      insert_separator()
4.      print("Hello, ", your_name, "!", sep="")
5.
6.  def insert_separator():
7.      print("===")
8.
9.  def recite_poem():
10.     print("How about a Monty Python poem?")
11.     insert_separator()
12.     print("Much to his Mum and Dad's dismay")
13.     print("Horace ate himself one day.")
14.     print("He didn't stop to say his grace,")
15.     print("He just sat down and ate his face.")
16.
17. def say_goodbye():
18.     print("Goodbye!")
19.
20. def main():
21.     say_hello()
22.     insert_separator()
23.     recite_poem()
24.     insert_separator()
25.     say_goodbye()
26.
27. main()
```

Code Explanation

The preceding code will render the following:

```
What is your name? Nat
===
Hello, Nat!
===
How about a Monty Python poem?
===
Much to his Mum and Dad's dismay
Horace ate himself one day.
He didn't stop to say his grace,
He just sat down and ate his face.
===
Goodbye!
```

Not All Modules are Programs

Not all Python modules are programs. Some modules are only meant to be used as helper files for other programs. Sometimes these modules are more or less generic, providing functions that could be useful to many different types of programs. And sometimes these modules are written to work with a specific program. Modules that aren't programs probably wouldn't have a `main()` function.

Variable Scope

Question: Why doesn't the `say_goodbye()` function use the user's name (e.g., `print("Goodbye, ", your_name)`)?

Answer: It doesn't know what `your_name` is.

Variables declared within a function are *local* to that function. Consider the following code:

Demo 9: functions/Demos/local_var.py

```
1. def set_x():
2.     x = 1
3.
4. def get_x():
5.     print(x)
6.
7. def main():
8.     set_x()
9.     get_x()
10.
11. main()
```

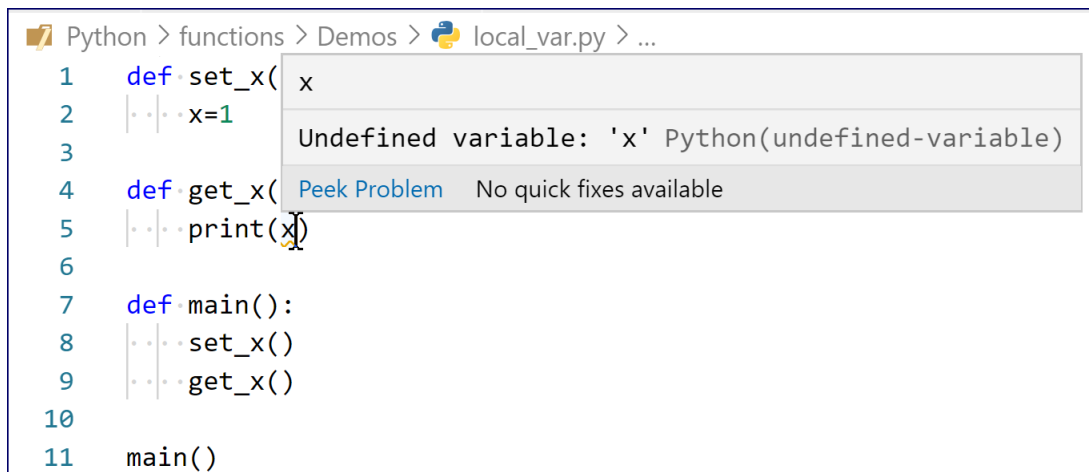
Code Explanation

Run this and you'll get an error similar to the following:

```
NameError: name 'x' is not defined
```

That's because `x` is defined in the `set_x()` function and is therefore *local* to that function.

A good Python IDE, like Visual Studio Code, will let you know when it detects such an error. In VS Code, a squiggly underline will appear beneath the undefined variable. Depending on how your settings are configured, you may be able to hover over the variable to see the error:



Global Variables

Global variables are defined outside of a function as shown in the following demo:

Demo 10: functions/Demos/global_var.py

```
1.  x = 0
2.
3.  def set_x():
4.      x = 1
5.      print("from set_x():", x)
6.
7.  def get_x():
8.      print("from get_x():", x)
9.
10. def main():
11.     set_x()
12.     get_x()
13.
14. main()
```

Code Explanation

`x` is first declared outside of a function, which means that it is a *global* variable.

Question: What do you think the `get_x()` function will print: 0 or 1?

Answer: It will print 0. That's because the `x` in `set_x()` is not the same as the global `x`. The former is local to the `set_x()` function.

Global variables, by default, can be referenced but not modified within functions:

- When a variable is *referenced* within a function, Python first looks for a local variable by that name. If it doesn't find one, then it looks for a global variable.
- When a variable is *assigned* within a function, it will be a local variable, even if a global variable with the same name already exists.

To modify global variables within a function, you must explicitly state that you want to work with the global variable. That's done using the `global` keyword, like this:

Demo 11: functions/Demos/global_var_in_function.py

```
1.  x = 0
2.
3.  def set_x():
4.      global x
5.      x = 1
6.
7.  def get_x():
8.      print(x)
9.
10. def main():
11.     set_x()
12.     get_x()
13.
14. main()
```

Code Explanation

Now, the `set_x()` function explicitly references the global variable `x`, so the code will print 1.

Naming global variables?

Some developers feel that any use of global variables is bad programming. While we won't go that far, we do have two recommendations:

1. Prefix your global variables with an underscore¹⁰ (e.g., `_x`). That makes it clear that the variable is global and minimizes the chance of it getting confused with a local variable of the same name. It is also a convention that lets developers know that those variables are not meant to be used outside the module (i.e., by programs importing the module).
2. When possible, rather than using global variables, design your code so that values can be passed from one function to another using parameters (see next section).

Function Parameters

Consider the `insert_separator()` function in the `hello_you_expanded.py` file that we saw earlier:

10. <https://www.python.org/dev/peps/pep-0008/#global-variable-names>

```
def insert_separator():  
    print("===")
```

What if we wanted to have different types of separators? One solution would be to create multiple functions, like `insert_large_separator()` and `insert_small_separator()`, but that can get pretty tiresome and hard to maintain. A better solution is to use function parameters using the following syntax:

```
def function_name(param1, param2, param3):  
    do_something(param1, param2, param3)
```

Here is our “Hello, You!” program using parameters:

Demo 12: functions/Demos/hello_you_with_params.py

```
1.  def say_hello(name):  
2.      print('Hello, ', name, '!', sep='')  
3.  
4.  def insert_separator(s):  
5.      print(s, s, s, sep="")  
6.  
7.  def recite_poem():  
8.      print("How about a Monty Python poem?")  
9.      insert_separator("-")  
10.     print("Much to his Mum and Dad's dismay")  
11.     print("Horace ate himself one day.")  
12.     print("He didn't stop to say his grace,")  
13.     print("He just sat down and ate his face.")  
14.  
15. def say_goodbye(name):  
16.     print('Goodbye, ', name, '!', sep='')  
17.  
18. def main():  
19.     your_name = input('What is your name? ')  
20.     insert_separator("-")  
21.     say_hello(your_name)  
22.     insert_separator("=")  
23.     recite_poem()  
24.     insert_separator("=")  
25.     say_goodbye(your_name)  
26.  
27. main()
```

Code Explanation

Now that `insert_separator()` takes the separating character as an argument, we can use it to separate lines with any character we like.

We have also modified `say_hello()` and `say_goodbye()` so that they receive the name of the person they are addressing as an argument. This has a couple of advantages:

1. We can move `your_name = input('What is your name? ')` to the `main()` function so we can pass `your_name` into both `say...` functions.
2. We can move the call to `insert_separator()` out of the `say_hello()` function as the separator doesn't have anything to do with saying "hello."

Parameters vs. Arguments

The terms *parameter* and *argument* are often used interchangeably, but there is a difference:

Parameters are the variables in the function definition. They are sometimes called *formal parameters*.

Arguments are the values passed into the function and assigned to the parameters. They are sometimes called *actual parameters*.

```
Python > functions > Demos > parameters_vs_arguments.py > ...
1 def ask_something(something):
2     print(something)
3
4
5
6 ask_something("What is your quest?")
-
```

❖ Using Parameter Names in Function Calls

When calling a function, you can specify the parameter by name when passing in an argument. When you do so, it's called passing in a *keyword argument*. For example, you can call the following `divide()` function in several ways:

```
def divide(numerator, denominator):  
    return numerator / denominator  
  
divide(10, 2)  
divide(numerator=10, denominator=2)  
divide(denominator=2, numerator=10)  
divide(10, denominator=2)
```

As you can see, using keyword arguments allows you to pass in the arguments in an arbitrary order. You can combine non-keyword arguments with keyword arguments in a function call, but you must pass in the non-keyword arguments first.

Later, we'll see that a function can be written to require keyword arguments.



Exercise 6: A Function with Parameters

🕒 15 to 25 minutes

In this exercise, you will write a function for adding two numbers together.

1. Open `functions/Exercises/add_nums.py` in your editor and review the code.
2. Now, run the file in Python. The output should look like this:

```
3 + 6 = 9
10 + 12 = 22
```

3. Replace the two crazy `add_...()` functions with an `add_nums()` function that accepts two numbers, adds them together, and outputs the equation.

Exercise Code: `functions/Exercises/add_nums.py`

```
1. def add_3_and_6():
2.     total = 3 + 6
3.     print('3 + 6 = ', total)
4.
5. def add_10_and_12():
6.     total = 10 + 12
7.     print('10 + 12 = ', total)
8.
9. def main():
10.     add_3_and_6()
11.     add_10_and_12()
12.
13. main()
```

Code Explanation

The first function adds 3 and 6. The second function adds 10 and 12. These functions are only useful if you want to add those specific numbers.

Solution: functions/Solutions/add_nums.py

```
1. def add_nums(num1, num2):
2.     total = num1 + num2
3.     print(num1, '+', num2, ' = ', total)
4.
5. def main():
6.     add_nums(3, 6)
7.     add_nums(10, 12)
8.
9. main()
```

Code Explanation

The `add_nums()` function is flexible and reusable. It can add any two numbers.

Default Values

Parameters that do not have default values require arguments to be passed in. You can assign default values to parameters using the following syntax:

```
def function_name(param=default):
    do_something(param)
```

For example, the following code would make the “=” sign the default separator:

```
def insert_separator(s="="):
    print(s, s, s, sep="")
```

When an argument is not passed into a parameter that has a default value, the default is used.

See `functions/Demos/hello_you_with_defaults.py` to see a working demo.



Exercise 7: Parameters with Default Values

🕒 15 to 25 minutes

In this exercise, you will write a function that can add two, three, four, or five numbers together.

1. Open `functions/Exercises/add_nums_with_defaults.py` in your editor.
2. Notice the `add_nums()` function takes five arguments, adds them together, and outputs the sum.
3. Modify `add_nums()` so that it can accept all of the following calls:

```
add_nums(1, 2)
add_nums(1, 2, 3, 4, 5)
add_nums(11, 12, 13, 14)
add_nums(101, 201, 301)
```

4. For now, it's okay for the function to print out 0s for values not passed in, like this:

```
1 + 2 + 0 + 0 + 0 = 3
1 + 2 + 3 + 4 + 5 = 15
11 + 12 + 13 + 14 + 0 = 50
101 + 201 + 301 + 0 + 0 = 603
```

Exercise Code: `functions/Exercises/add_nums_with_defaults.py`

```
1. def add_nums(num1, num2, num3, num4, num5):
2.     total = num1 + num2 + num3 + num4 + num5
3.     print(num1, '+', num2, '+', num3, '+', num4, '+', num5, ' = ', total)
4.
5. def main():
6.     add_nums(1, 2, 0, 0, 0)
7.     add_nums(1, 2, 3, 4, 5)
8.     add_nums(11, 12, 13, 14, 0)
9.     add_nums(101, 201, 301, 0, 0)
10.
11. main()
```

Solution: functions/Solutions/add_nums_with_defaults.py

```
1. def add_nums(num1, num2, num3=0, num4=0, num5=0):
2.     total = num1 + num2 + num3 + num4 + num5
3.     print(num1, '+', num2, '+', num3, '+', num4, '+', num5, ' = ', total)
4.
5. def main():
6.     add_nums(1, 2)
7.     add_nums(1, 2, 3, 4, 5)
8.     add_nums(11, 12, 13, 14)
9.     add_nums(101, 201, 301)
10.
11. main()
```

Code Explanation

We have given the last three parameters default values of 0, making them optional. The first two parameters don't have default values, so they are still required.

Returning Values

Functions can return values. The `add_nums()` function we have been working with does more than add the numbers passed in, it also prints them out. You can imagine wanting to add numbers for some other purpose than printing them out. Or you might want to print the results out in a different way. We can change the `add_nums()` function so that it just adds the numbers together and returns the sum. Then our program can decide what to do with that sum. Take a look at the following code:

Demo 13: functions/Demos/add_nums_with_return.py

```
1. def add_nums(num1, num2, num3=0, num4=0, num5=0):
2.     total = num1 + num2 + num3 + num4 + num5
3.     return total
4.
5. def main():
6.     result = add_nums(1, 2)
7.     print(result)
8.     result = add_nums(result, 3)
9.     print(result)
10.    result = add_nums(result, 4)
11.    print(result)
12.    result = add_nums(result, 5)
13.    print(result)
14.    result = add_nums(result, 6)
15.    print(result)
16.
17. main()
```

Code Explanation

The `add_nums()` function now returns the sum to the calling function via the `return` statement. We assign the result to a local variable named `result`. Then we print `result` and pass it back to `add_nums()` in subsequent calls.

Note that once a function has returned a value, the function is finished executing and control is transferred back to the code that invoked the function.

Importing Modules

As we saw, part of the beauty of writing functions is that they can be reused. Imagine you write a really awesome function. Or even better, a module with a whole bunch of really awesome functions in it. You'd want to make those functions available to other modules so you (and other Python developers) could make use of them elsewhere.

Modules can import other modules using the `import` keyword as shown in the following example:

Demo 14: functions/Demos/import_example.py

```
1. import add_nums_with_return
2.
3. total = add_nums_with_return.add_nums(1, 2, 3, 4, 5)
4. print(total)
```

Code Explanation

Now, the `add_nums()` function from `add_nums_with_return.py` is available in `import_example.py`; however, it must be prefixed with “`add_nums_with_return.`” (the module name) when called.

The `main()` Function

It is common for a module to check to see if it is being imported by checking the value of the special `__name__` variable. Such a module will only run its `main()` function if `__name__` is equal to `'__main__'`, indicating that it is not being imported. The code usually goes at the bottom of the module and looks like this:

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

Note that those are two underscores before and after `name` and before and after `main`.

A short video explanation of this is available at https://bit.ly/python_main.

Another way to import functions from another module is to use the following syntax:

```
from module_name import function1, function2
```

For example:

Demo 15: functions/Demos/import_example2.py

```
1.  from add_nums_with_return import add_nums
2.
3.  total = add_nums(1, 2, 3, 4, 5)
4.  print(total)
```

When you use this approach, it is not necessary to prefix the module name when calling the function. However, it's possible to have naming conflicts, so be careful.

Another option, which is helpful for modules with long names, is to create an alias for a module, so that you do not have to type its full name:

```
import add_nums_with_return as anwr

total = anwr.add_nums(1, 2, 3, 4, 5)
```

Using aliases is also a way of preventing naming conflicts. If you import `do_this` from `foo` and `do_this` from `bar`, you can use an alias to call one of them `do_that`:

```
from foo import do_this
from bar import do_this as do_that
```

❖ Module Search Path

The Python interpreter must locate the imported modules. When `import` is used within a script, the interpreter searches for the imported module in the following places sequentially:

1. The current directory (same directory as the script doing the importing).
2. The library of standard modules.¹¹
3. The paths defined in `sys.path`, which you can see by running the following code at the Python shell:

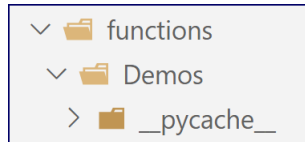
```
>>> import sys
>>> sys.path
```

This will output a list of paths, which are searched in order for the imported module.

11. <https://docs.python.org/3/tutorial/modules.html#standard-modules>

❖ pyc Files

Files with a `.pyc` extension are *compiled* Python files. They are automatically created in a `__pycache__` folder the first time a file is imported:



These files are created so that modules you import don't have to be compiled every time they run. You can just leave those files alone. They will automatically be created/updated each time you import a module that is new or has been changed.

Methods vs. Functions

You have already seen some built-in functions, like `print()` and `input()`. You have also written some of your own, like `insert_separator()` and `divide()`. In the upcoming lessons, you will learn to use many more of Python's built-in functions. You will also learn about *methods*, which are similar to functions, except that they are called on an object using the syntax `object_name.method_name()`.

An example of a simple built-in function is `len()`, which returns the length of the passed-in object:

```
>>> len('Webucator')
9
```

An example of a method of a string object is `upper()`, which returns the string it is called upon in all uppercase letters:

```
>>> 'Webucator'.upper()
'WEBUCATOR'
```

Again, you will work with many functions and methods in upcoming lessons.

Conclusion

In this lesson, you have learned to define functions with or without parameters. You have also learned about variable scope and how to import modules.

LESSON 3

Math

Topics Covered

- ✓ Basic math in Python.
- ✓ The `math` module.
- ✓ The `random` module.

I had been to school most all the time and could spell and read and write just a little, and could say the multiplication table up to six times seven is thirty-five, and I don't reckon I could ever get any further than that if I was to live forever. I don't take no stock in mathematics, anyway.

– *Adventures of Huckleberry Finn*, Mark Twain

Introduction

Python includes some built-in math functions and some additional built-in libraries that provide extended math (and related) functionality. In this lesson, we'll cover the built-in functions and the `math` and `random` libraries.

Arithmetic Operators

The following table lists the arithmetic operators in Python. Most will be familiar. We'll explain the others.

Arithmetic Operators

Operator	Description
+	Addition
	5 + 2 returns 7
-	Subtraction
	5 - 2 returns 3
*	Multiplication
	5 * 2 returns 10
/	Division
	5/2 returns 2.5
%	Modulus
	5 % 2 returns 1
**	Power
	5**2 is 5 to the power of 2. It returns 25
//	Floor Division
	5 // 2 returns 2

Here are the examples in Python:

```
>>> 5+2
7
>>> 5-2
3
>>> 5*2
10
>>> 5/2
2.5
>>> 5%2
1
>>> 5**2
25
>>> 5//2
2
```

❖ Modulus and Floor Division

You may remember doing this in elementary school math:

$$\begin{array}{r} 2^r 1 \\ 2 \overline{) 5} \end{array}$$

The remainder of 5 divided by 2 is 1.

The *modulus* operator (%) is used to find the remainder after division:

```
>>> 5 % 2
1
>>> 11 % 3
2
>>> 22 % 4
2
>>> 22 % 3
1
>>> 10934 % 324
242
```

Modulus and Negative Numbers

The results of modulus operations with negative numbers can be surprising, and are different in different programming languages. Python uses the following formula: $x - y*(x//y)$, which will always return a positive number. This is mostly academic. You will likely never have to deal with this, so we will not dig further into it.

The *floor division* operator (//) is the same as regular division, but rounded down:

```
>>> 5 // 2
2
>>> 11 // 3
3
>>> 22 // 4
5
>>> 22 // 3
7
>>> 10934 // 324
33
>>> -5 // 2 # rounded down, meaning towards negative infinity
-3
```



Exercise 8: Floor and Modulus

⌚ 5 to 10 minutes

In this exercise, you will write a small function called `divide()` that takes a numerator and denominator and prints out a response that a fifth grader would understand (e.g., “5 divided by 2 equals 2 with a remainder of 1”).

1. Open `math/Exercises/floor_modulus.py` in your editor.
2. Write the `divide()` function.
3. Run the module. It should output the following:

```
5 divided by 2 equals 2 with a remainder of 1
6 divided by 3 equals 2 with a remainder of 0
12 divided by 5 equals 2 with a remainder of 2
1 divided by 2 equals 0 with a remainder of 1
```

Exercise Code: `math/Exercises/floor_modulus.py`

```
1.  # write the divide() function
2.
3.  def main():
4.      divide(5, 2)
5.      divide(6, 3)
6.      divide(12, 5)
7.      divide(1, 2)
8.
9.  main()
```

```
1.  def divide(num, den):
2.      remainder = num % den
3.      floor = num // den
4.      print(num, "divided by", den, "equals",
5.            floor, "with a remainder of", remainder)
6.
7.  def main():
8.      divide(5, 2)
9.      divide(6, 3)
10.     divide(12, 5)
11.     divide(1, 2)
12.
13.  main()
```

Assignment Operators

The following table lists the assignment operators in Python.

Assignment Operators

Operator	Description
=	Basic assignment
	a = 2
+=	One step addition and assignment
	a += 2 same as a = a + 2
-=	One step subtraction and assignment
	a -= 2 same as a = a - 2
*=	One step multiplication and assignment
	a *= 2 same as a = a * 2
/=	One step division and assignment
	a /= 2 same as a = a / 2

Here are the examples from the table above in a Python script:

```

>>> a = 5
>>> a
5
>>> a += 2
>>> a
7
>>> a -= 2
>>> a
5
>>> a *= 2
>>> a
10
>>> type(a)
<class 'int'>
>>> a /= 2
>>> a
5.0
>>> type(a) # Notice division returns a float
<class 'float'>

```

Notice that `a` changes from an integer to a float when division is performed on it.

Precedence of Operations

The order of operations for arithmetic operators is:

1. `**`
2. `*`, `/`, `//`, `%`
3. `+`, `-`

You can use parentheses to change the order of operations and give an operation higher precedence. For example:

- `6 + 3 / 3` is equal to `6 + 1` and will yield 7.
- But `(6 + 3) / 3` is equal to `9 / 3` and will yield 3.

Operations of equal precedence are evaluated from left to right, so `6 / 2 * 3` and `(6 / 2) * 3` are the same.

Built-in Math Functions

Python's built-in functions¹² include several math functions.

❖ `int(x)`

`int(x)` returns `x` converted to an integer.

When converting floats, `int(x)` strips everything after the decimal point, essentially rounding down for positive numbers and rounding up for negative numbers.

When converting strings, the string object must be an accurate representation of an integer (e.g., `'5'`, but not `'5.0'`).

```
>>> int(5)
5
>>> int('5')
5
>>> int(5.4)
5
>>> int(5.9)
5
>>> int(-5.9)
-5
>>> int('5.4')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '5.4'
```

❖ `float(x)`

`float(x)` returns `x` converted to a float.

12. <https://docs.python.org/3/library/functions.html>


```
>>> float(5)
5.0
>>> float('5')
5.0
>>> float(5.4)
5.4
>>> float('5.4')
5.4
>>> float('-5.99')
-5.99
>>> float(-5.99)
-5.99
```

❖ **abs(x)**

`abs(x)` returns the absolute value of `x` as an integer or a float.

```
>>> abs(-5)
5
>>> abs(5)
5
>>> abs(-5.5)
5.5
>>> abs(5.5)
5.5
```

❖ **min() and max()**¹³

The `min(args)` function returns the smallest value of the passed-in arguments.

The `max(args)` function returns the largest value of the passed-in arguments.

13. The `min()` and `max()` functions can also compare strings.

```
>>> min(2, 1, 3)
1
>>> min(3.14, -1.5, -300)
-300
>>> max(2, 1, 3)
3
>>> max(3.14, -1.5, -300)
3.14
```

min() and max() with Iterables

The `min()` and `max()` functions can also take an *iterable* containing values to compare. We will cover this in the **Iterables** lesson (see page 130).

❖ `pow(base, exp[, mod])`

Passing two arguments to the `pow()` function is the equivalent of using the power operator (`**`):

```
pow(base, exp)
```

is the same as

```
base**exp
```

For example:

```
>>> pow(4, 2)
16
>>> 4**2
16
```

`pow()` can take a third argument: `mod`. `pow(base, exp, mod)` is functionally equivalent to `base**exp % mod`:

```
>>> pow(4, 2, 3)
1
>>> 4**2 % 3
1
```

Square Brackets in Code Notation

Square brackets in code notation indicate that the contained portion is optional. Consider the `pow()` function signature:

```
pow(base, exp[, mod])
```

This means that the `mod` parameter is optional.

❖ `round(number[, ndigits])`

`round(number[, ndigits])` returns `number` as a number rounded to `ndigits` digits after the decimal. If `ndigits` is `0` or omitted then the function rounds to the nearest integer. If `ndigits` is `-1` then it rounds to the nearest 10 (e.g., `round(55, -1)` returns `60`).

```
>>> round(55, -1)
60
>>> round(3.14)
3
>>> round(-3.14)
-3
>>> round(3.14, 1)
3.1
>>> round(3.95, 1)
4.0
>>> round(1111, 0)
1111
```

❖ `sum(iter[, start])`

The `sum()` function takes an iterable (e.g., a list) and adds up all of its elements. We will cover this in the **Iterables** lesson (see page 131).

The math Module

The `math` module is built in to Python and provides many useful methods. We cover some of them here.¹⁴

To access any of these methods, you must first import `math`:

```
>>> import math
```

❖ Common Methods of the math Module

`math.ceil(x)`

`x` rounded up to the nearest whole number as an integer.

```
>>> math.ceil(5.4)
6
>>> math.ceil(-5.4)
-5
>>>
```

`math.floor(x)`

`x` rounded down to the nearest whole number as an integer.

```
>>> math.floor(5.6)
5
>>> math.floor(-5.6)
-6
```

`math.trunc(x)`

`x` with the fractional truncated, effectively rounding towards 0 to the nearest whole number as an integer.

^{14.} To get a full list of the `math` module's methods, import `math` and then type `help(math)` in the Python shell or visit <https://docs.python.org/3/library/math.html>.

```
>>> math.trunc(5.6)
5
>>> math.trunc(-5.6)
-5
```

math.fabs(x)

The absolute value of float `x`. This is similar to the built-in `abs(x)` function except that `math.fabs(x)` always returns a float whereas `abs(x)` returns a number of the same data type as `x`.

```
>>> math.fabs(-5)
5.0
>>> abs(-5)
5
```

math.factorial(x)

The factorial of `x`. This is often written as `x!`, but not in Python!

```
>>> math.factorial(3)
6
>>> math.factorial(5)
120
```

math.pow(x, y)

`x` raised to the power `y` as a float.

```
>>> math.pow(5, 2)
25.0
```

math.sqrt(x)

The square root of `x` as a float.

```
>>> math.sqrt(25)
5.0
```

The `math` module also contains two constants: `math.pi` and `math.e`, for π and e as used in the natural logarithm.¹⁵

❖ Additional math Methods

The `math` library offers many additional methods, including:

- Logarithmic methods (e.g., `math.log(x)`)
- Trigonometric methods (e.g., `math.sin(x)`)
- Angular conversion methods (e.g., `math.degrees(x)`)
- Hyperbolic methods (e.g., `math.sinh(x)`)

The random Module

The `random` module is also built into Python and includes methods for creating and selecting random values.

To access any of these methods, you must first import `random`:

```
>>> import random
```

❖ Common Methods of the random Module¹⁶

`random.random()`

Random float between 0 and 1.

```
>>> random.random()
0.5715141345521301
```

^{15.} https://en.wikipedia.org/wiki/Natural_logarithm

^{16.} To get a full list of the `random` module's methods, import `random` and then type `help(random)` in the Python shell or visit <https://docs.python.org/3/library/random.html>.

random.randint(a, b)

Random integer between (and including) a and b.

```
>>> random.randint(1, 10)
7 # integer between 1 and 10
```

random.randrange(b)

Random integer between 0 and b-1.

```
>>> random.randrange(10)
3 # integer between 0 and 9
```

random.randrange(a, b)

Random integer between a and b-1.

```
>>> random.randrange(1, 10)
5 # integer between 1 and 9
```

random.randrange(a, b, step)

Random integer at step between (and including) a and b-1.

```
>>> random.randrange(1, 10, 2)
3 # one of 1, 3, 5, 7, 9
```

random.uniform(a, b)

Random float between (and including) a and b.

```
>>> random.uniform(1, 10)
8.028088082797572 # a float between 1 and 10
```

random.choice(seq) and random.shuffle(seq)

`random.choice(seq)` returns a random element in the sequence `seq`.

`random.shuffle(seq)` shuffles the sequence `seq` in place.

Sequences are covered in an upcoming lesson. (see page 112)

❖ Seeding

`random.seed(a)` is used to initialize the random number generator. The value of `a` will determine how random numbers are selected. The following code illustrates this:

```
>>> import random
>>> random.seed(1)
>>> random.randint(1, 100)
18
>>> random.randint(1, 100)
73
>>> random.randint(1, 100)
98
# reseed
>>> random.seed(1)
>>> random.randint(1, 100)
18
>>> random.randint(1, 100)
73
>>> random.randint(1, 100)
98
```

Notice that the random numbers generated depend on the seed. If you run this same code locally, you should get the same random integers. This can be useful for testing.

By default, `random.seed()` uses the current system time to ensure that `seed()` is seeded with a different number every time it runs, so that the random numbers generated will be different each time.



Exercise 9: How Many Pizzas Do We Need?

⌚ 15 to 25 minutes

In this exercise, you will write a program from scratch. Your program should prompt the user to input the information required:

- The number of people.
- The number of slices each person will eat.
- The number of slices in each pizza pie.

Using that information, your program must calculate how many pizzas are needed to feed everyone. It should work like this:


```
How many people are eating? 5
How many slices per person? 2.5
How many slices per pie? 8
You need 2 pizzas to feed 5 people.
There will be 3.5 leftover slices.
```

```
How many people are eating? 25
How many slices per person? 2
How many slices per pie? 8
You need 7 pizzas to feed 25 people.
There will be 6.0 leftover slices.
```

Solution: math/Solutions/pizza_slices.py

```
1.  import math
2.
3.  def main():
4.      people = int(input("How many people are eating? "))
5.      slices_per_person = float(input("How many slices per person? "))
6.      slices = slices_per_person * people
7.
8.      slices_per_pie = int(input("How many slices per pie? "))
9.      pizzas = math.ceil(slices / slices_per_pie)
10.
11.     print("You need", pizzas, "pizzas to feed", people, "people.")
12.
13.     total_slices = slices_per_pie * pizzas
14.     slices_left = total_slices - slices
15.     print("There will be", slices_left, "leftover slices.")
16.
17.  main()
```

Exercise 10: Dice Rolling

 15 to 25 minutes

In this exercise, you will write a dice-rolling program from scratch. The program should include two functions: `main()` and `roll_die()`. The `roll_die()` function should take one parameter: `sides`, and return a random roll between 1 and `sides`

The `main()` function should call the `roll_die()` function three times and keep a tally of the total. After each roll, it should print out the value of the roll, the number of rolls, and the total. At the end, it should output the average of the three rolls. The output will be similar to the following:

```
You rolled a 3
Total after first roll: 3
You rolled a 5
Total after 2 rolls: 8
You rolled a 3
Total after 3 rolls: 11
Your average roll was 3.67
Thanks for playing.
```

Solution: math/Solutions/dice.py

```
1.  import random
2.
3.  def roll_die(sides=6):
4.      num_rolled = random.randint(1, sides)
5.      return num_rolled
6.
7.  def main():
8.      sides = 6
9.      total = 0
10.
11.     num_rolls = 1
12.     roll = roll_die(sides)
13.     print("You rolled a", roll)
14.     total += roll
15.     print("Total after first roll:", total)
16.
17.     num_rolls += 1
18.     roll = roll_die(sides)
19.     print("You rolled a", roll)
20.     total += roll
21.     print("Total after", num_rolls, "rolls:", total)
22.
23.     num_rolls += 1
24.     roll = roll_die(sides)
25.     print("You rolled a", roll)
26.     total += roll
27.     print("Total after", num_rolls, "rolls:", total)
28.
29.     average = round(total / num_rolls, 2)
30.     print("Your average roll was", average)
31.
32.     print("Thanks for playing.")
33.
34.  main()
```

Conclusion

In this lesson, you have learned to do basic math in Python and to use the `math` and `random` modules for extended math functionality.

LESSON 4

Python Strings

Topics Covered

- ☑ String basics.
- ☑ Special characters.
- ☑ Multi-line strings.
- ☑ Indexing and slicing strings.
- ☑ Common string operators and methods.
- ☑ Formatting strings.
- ☑ Built-in string functions.

‘Yes,’ they say to one another, these so kind ladies, ‘he is a stupid old fellow, he will see not what we do, he will never observe that his sock heels go not in holes any more, he will think his buttons grow out new when they fall, and believe that strings make themselves.’

— *Little Women*, Louisa May Alcott

Introduction

According to the Python documentation¹⁷, “Strings are immutable sequences of Unicode code points.” Less technically speaking, strings are sequences of characters.¹⁸ The term *sequence* in Python refers to an ordered set. Other common sequence types are lists, tuples, and ranges, all of which we will cover.

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

18. Note that there is no “character” type in Python. A single character is just a string of length 1. So, when you index a string, you get multiple one-character strings (or strings of length 1).

Quotation Marks and Special Characters

Strings can be created with single quotes or double quotes. There is no difference between the two.

❖ Escaping Characters

To create a string that contains a single quote (e.g., `Where'd you get the coconuts?`), enclose the string in double quotes:

```
phrase = "Where'd you get the coconuts?"
```

Likewise, to create a string that contains a double quote (e.g., `The soldier asked, "Are you suggesting coconuts migrate?"`), enclose the string in single quotes:

```
phrase = 'The soldier asked, "Are you suggesting coconuts migrate?"'
```

Sometimes you will want to output single quotes within a string denoted by single quotes or double quotes within a string denoted by double quotes. In such cases, you will need to *escape* the quotation marks using a backslash (`\`), like this:

```
>>> phrase = "The soldier said, \"You've got two empty halves of a coconut.\""
>>> print(phrase)
The soldier said, "You've got two empty halves of a coconut."

# or

>>> phrase = 'The soldier said, "You\'ve got two empty halves of a coconut."'
>>> print(phrase)
The soldier said, "You've got two empty halves of a coconut."
```

Notice that the printed output does not contain the backslashes.

Special Characters

The backslash can also be used to escape characters that have special meaning, such as the backslash itself:

```
>>> phrase = "Use an extra backslash to output a backslash: \\"
>>> print(phrase)
Use an extra backslash to output a backslash: \
```

Two other common special characters are the newline (`\n`) and horizontal tab (`\t`):

```
>>> print('Equation\tSolution\n 55 x 11\t 605\n 132 / 6\t 22')
Equation      Solution
 55 x 11      605
 132 / 6      22
```

Escape Sequences

Escape Sequence	Meaning
\'	Single quote
\"	Double quote
\\	Backslash
\n	Newline
\t	Horizontal tab

Raw Strings

Sometimes, a string might have a lot of backslashes in it that are just meant to be plain old backslashes. The most common example is a file path. For example:

```
'C:\news\today.txt'
```

Watch what happens when that string is assigned to a variable:

```
>>> my_path = 'C:\news\today.txt'
>>> print(my_path)
C:
ews      oday.txt
```

When we print `my_path`, the `\n` and `\t` characters get printed as a newline and a tab.

Using the “r” (for **raw** data) prefix on the string, we ensure that all backslashes are escaped:

```
>>> my_path = r'C:\news\today.txt'
>>> print(my_path)
C:\news\today.txt
```

If you examine the variable directly without printing it, you see that each backslash is escaped with another backslash:

```
>>> my_path
'C:\\news\\today.txt'
```

Note that backslashes will always escape single and double quotation marks, even in a raw string, so you cannot end a raw string with a single backslash:

Bad

```
my_path = r'C:\my\new\'
```

Good

```
my_path = r'C:\my\new\\'
```

Raw strings can come in particularly handy when working with regular expressions.

❖ Triple Quotes

Triple quotes are used to create multi-line strings. You generally use three double quotes¹⁹ as shown in the following example:

Demo 16: strings/Demos/triple_quotes.py

```
1.  print("-----")
2.  LUMBERJACK SONG
3.
4.  I'm a lumberjack
5.  And I'm O.K.
6.  I sleep all night
7.  And I work all day.
8.  -----")
```

19. Three single quotes will work as well, but double quotes are recommended. More information at <https://www.python.org/dev/peps/pep-0008/#string-quotes>.

Code Explanation

The preceding code will render the following:

```
-----  
LUMBERJACK SONG  
  
I'm a lumberjack  
And I'm O.K.  
I sleep all night  
And I work all day.  
-----
```

Note that quotation marks can be included within triple-quoted strings without being escaped with a backslash.

In some cases when using triple quotes, you may want to break up your code with a newline without having that newline show up in your output. You can escape the actual newline with a backslash as shown in the following demo:

Demo 17: strings/Demos/triple_quotes_newline_escaped.py

```
1.  print("""We're knights of the Round Table, \  
2.  we dance whene'er we're able.  
3.  We do routines and chorus scenes \  
4.  with footwork impeccable,  
5.  We dine well here in Camelot, \  
6.  we eat ham and jam and Spam a lot.""")
```

Code Explanation

The preceding code will render the following:

```
We're knights of the Round Table, we dance whene'er we're able.  
We do routines and chorus scenes with footwork impeccable,  
We dine well here in Camelot, we eat ham and jam and Spam a lot.
```

Notice that the backslashes at the end of lines 1, 3, and 5 prevent line breaks in the output.

String Indexing

Indexing is the process of finding a specific element within a sequence of elements through the element's position. Remember that strings are basically sequences of characters. We can use indexing to find a specific character within the string.

If we consider the string from left to right, the first character (the left-most) is at position 0. If we consider the string from right to left, the first character (the right-most) is at position -1. The following table illustrates this for the phrase "MONTY PYTHON".

	M	O	N	T	Y		P	Y	T	H	O	N
Left to Right:	0	1	2	3	4	5	6	7	8	9	10	11
Right to Left:	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

The following demonstration shows how to find characters by position using indexing.

Demo 18: strings/Demos/string_indexing.py

```
1. phrase = "Monty Python"
2.
3. first_letter = phrase[0] # [M]onty Python
4. print(first_letter)
5.
6. last_letter = phrase[-1] # Monty Pytho[n]
7. print(last_letter)
8.
9. fifth_letter = phrase[4] # Mont[y] Python
10. print(fifth_letter)
11.
12. third_letter_from_end = phrase[-3] # Monty Pyt[h]on
13. print(third_letter_from_end)
```

Code Explanation

The expected output for each `print` statement is shown in square brackets in the comment. Running the file should result in:

```
M
n
y
h
```



Exercise 11: Indexing Strings

🕒 10 to 20 minutes

In this exercise, you will write a program that gets a specific character from a phrase entered by the user.

1. Open `strings/Exercises/indexing.py`.
2. Modify the `main()` function so that it:
 - A. Prompts the user to enter a phrase.
 - B. Tells the user what phrase they entered (e.g., Your phrase is 'Hello, World!').
 - C. Prompts the user for a number.
 - D. Tells the user what character is at that position in the user's phrase (e.g., Character number 4 is o).
3. Here is the program completed by the user:

```
Choose a phrase: Hello, world!  
Your phrase is 'Hello, world!'  
Which character? [Enter number] 4  
Character 4 is o
```

Challenge

As a Python programmer, you understand that the “o” in “Hello” is at position 4, because Python starts counting with 0. However, regular people will think that the character at position 4 is “l” and they will think your program is wrong. Fix your program so that it responds as the user expects. Also, to make it a little prettier, output the character in single quotes.

The program should work like this:

```
Choose a phrase: Hello, world!  
Your phrase is 'Hello, world!'  
Which character? [Enter number] 4  
Character 4 is 'l'
```

Solution: strings/Solutions/indexing.py

```
1. def main():
2.     phrase = input("Choose a phrase: ")
3.     print("Your phrase is '", phrase, "'", sep=" ")
4.     pos = int(input("Which character? [Enter number] "))
5.     print("Character number", pos, "is", phrase[pos])
6.
7.     main()
```

Challenge Solution: strings/Solutions/indexing_challenge.py

```
1. def main():
2.     phrase = input("Choose a phrase: ")
3.     print("Your phrase is '", phrase, "'", sep=" ")
4.     pos = int(input("Which character? [Enter number] ")) - 1
5.     print("Character ", pos+1, " is '", phrase[pos], "'", sep=" ")
6.
7.     main()
```

Slicing Strings

Often, you will want to get a sequence of characters from a string (i.e., a *substring*). In Python, you do this by getting a slice of the string using the following syntax:

```
substring = orig_string[first_pos:last_pos]
```

This returns a slice that starts with the character at `first_pos` and includes all the characters up to *but not including* the character at `last_pos`.

If `first_pos` is left out, then it is assumed to be 0. So `'hello'[:3]` would return `"hel"`.

If `last_pos` is left out, then it is assumed to be the length of the string, or in other words, one more than the last position of the string. So `'hello'[3:]` would return `"lo"`.

The following demonstration shows how to get substrings using slicing.

Demo 19: strings/Demos/string_slicing.py

```
1. phrase = "Monty Python"
2.
3. first_5_letters = phrase[0:5] # [Monty] Python
4. print(first_5_letters)
5.
6. letters_2_thru_4 = phrase[1:4] # M[ont]y Python
7. print(letters_2_thru_4)
8.
9. letter_5_to_end = phrase[4:] # Mont[y Python]
10. print(letter_5_to_end)
11.
12. last_3_letters = phrase[-3:] # Monty Pyt[hon]
13. print(last_3_letters)
14.
15. first_3_letters = phrase[:3] # [Mon]ty Python
16. print(first_3_letters)
17.
18. three_letters_before_last = phrase[-4:-1] # Monty Py[tho]n
19. print(three_letters_before_last)
20.
21. copy_of_string = phrase[:] # [Monty Python]
22. print(copy_of_string)
```

Code Explanation

The expected output for each `print` statement is shown in square brackets in the comment. Running the file should result in:

```
Monty
ont
y Python
hon
Mon
tho
Monty Python
```



Exercise 12: Slicing Strings

🕒 10 to 20 minutes

In this exercise, you will write a program that gets a substring (or slice) from a phrase entered by the user.

1. Open `strings/Exercises/slicing.py`.
2. Modify the `main()` function so that it:
 - A. Prompts the user to enter a phrase.
 - B. Tells the user what phrase they entered (e.g., `Your phrase is 'Hello, World!'`).
 - C. Prompts the user for a start number.
 - D. Prompts the user for an end number.
 - E. Tells the user the substring (within single quotes) that starts with the start number and ends with the end number.
3. Here is the output of the program:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Character to start with? [Enter number] 4
Character to end with? [Enter number] 9
Your substring is 'o, wor'
```

Challenge

As with the last exercise, make your program respond as users would expect.

The new program should work like this:

```
Choose a phrase: Hello, world!
Your phrase is 'Hello, world!'
Character to start with? [Enter number] 4
Character to end with? [Enter number] 9
Your substring is 'lo, wo'
```


Solution: strings/Solutions/slicing.py

```
1. def main():
2.     phrase = input("Choose a phrase: ")
3.     print("Your phrase is '", phrase, "'", sep="")
4.     pos1 = int(input("Character to start with? [Enter number] "))
5.     pos2 = int(input("Character to end with? [Enter number] ")) + 1
6.     print("Your substring is '", phrase[pos1:pos2], "'", sep="")
7.
8.     main()
```

Challenge Solution: strings/Solutions/slicing_challenge.py

```
1. def main():
2.     phrase = input("Choose a phrase: ")
3.     print("Your phrase is '", phrase, "'", sep="")
4.     pos1 = int(input("Character to start with? [Enter number] ")) - 1
5.     pos2 = int(input("Character to end with? [Enter number] "))
6.     print("Your substring is '", phrase[pos1:pos2], "'", sep="")
7.
8.     main()
```

Concatenation and Repetition

❖ Concatenation

Concatenation is a fancy word for stringing strings together. In Python, concatenation is done with the + operator. It is often used to combine variables with literals as in the following example:

Demo 20: strings/Demos/concatenation.py

```
1. user_name = input("What is your name? ")
2. greeting = "Hello, " + user_name + "!"
3. print(greeting)
```

Code Explanation

The preceding code will render the following:


```
What is your name? Nat
Hello, Nat!
```

❖ Repetition

Repetition is the process of repeating a string some number of times. In Python, repetition is done with the `*` operator.

Demo 21: strings/Demos/repetition.py

```
1. one_knight_says = "nee"
2. many_knights_say = one_knight_says * 20
3. print(many_knights_say)
```

Code Explanation

The preceding code will render the following:

```
neeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneene
```

Exercise 13: Repetition

 5 to 10 minutes

Remember our `insert_separator()` function from the “Hello, You!” programs. It looked like this:

```
def insert_separator(s=" "):  
    print(s, s, s, sep=" ")
```

Using repetition, we can improve `insert_separator()` so that the number of times the separating character shows up is passed into the function.

1. Open `strings/Exercises/hello_you.py` in your editor.
2. Modify the `insert_separator()` function so that the number of times the separating character shows up is passed in as a parameter. It should default to show up 30 times.
3. Modify the calls to `insert_separator()` so that they pass in an argument to the new parameter.

Solution: strings/Solutions/hello_you.py

```
1. def say_hello(name):
2.     print('Hello, ', name, '!', sep='')
3.
4. def insert_separator(s="=", repeat=30):
5.     print(s * repeat)
6.
7. def recite_poem():
8.     print("How about a Monty Python poem?")
9.     insert_separator("-", 20)
10.    print("Much to his Mum and Dad's dismay")
11.    print("Horace ate himself one day.")
12.    print("He didn't stop to say his grace,")
13.    print("He just sat down and ate his face.")
14.
15. def say_goodbye(name):
16.     print('Goodbye, ', name, '!', sep='')
17.
18. def main():
19.     your_name = input('What is your name? ')
20.     insert_separator("-", 20)
21.     say_hello(your_name)
22.     insert_separator()
23.     recite_poem()
24.     insert_separator()
25.     say_goodbye(your_name)
26.
27. main()
```

Combining Concatenation and Repetition

Concatenation and repetition can be combined. Repetition takes precedence, meaning it occurs first. Consider the following:

```
>>> 'a' + 'b' * 3 + 'c'
'abbbc'
```

Notice the output is “abbbc”. In other words, “b” will be repeated three times before it is concatenated with “a” and “c”.

We can force the concatenation to take place first by using parentheses:

```
>>> ('a' + 'b') * 3 + 'c'
'abababc'
```

Notice the output is “abababc”. In other words, “a” will be concatenated with “b”, then “ab” will be repeated three times, and finally “ababab” will be concatenated with “c”.

The following demo shows an example of combining concatenation with repetition:

Demo 22: strings/Demos/combining_concatenation_and_repetition.py

```
1. flower = input("What is your favorite flower? ")
2. reply = "A " + flower + (" is a " + flower) * 2 + "."
3. print(reply)
```

Code Explanation

The preceding code will render the following:

```
What is your favorite flower? dandelion
A dandelion is a dandelion is a dandelion.
```

Python Strings are Immutable

Python strings are immutable, meaning that they cannot be changed. However, it is easy to make a copy of a string and then assign the copy to the same variable as the original string.

To illustrate, we will use Python’s built-in `id()` function, which returns the identity of an object:

```
>>> name = 'Nat'
>>> id(name)
1670060967728
>>> name += 'haniel'
>>> name
'Nathaniel'
>>> id(name)
1670060968240
```

Notice that the id of name changes when we modify the string in the variable.

Because strings are immutable, methods that operate on strings (i.e., string methods) cannot modify the string in place. Instead, they return a value. Sometimes that value is a modified version of the string, but it is important to understand that the original string is unchanged. Consider, for example, the `upper()` method, which returns a string in all uppercase letters:

```
>>> name = 'Nat'
>>> name.upper() # Returns uppercase copy of 'Nat'
'NAT'
>>> name # Original variable is unchanged
'Nat'
```

If you want to change the original variable, you must assign the returned value back to the variable:

```
>>> name = 'Nat'
>>> name = name.upper()
>>> name
'NAT'
```

Common String Methods

❖ String Methods that Return a Copy of the String

Methods that Change Case

- `str.capitalize()` – Returns a string with only the first letter capitalized.
- `str.lower()` – Returns an all lowercase string.

- `str.upper()` – Returns an all uppercase string.
- `str.title()` – Returns a string with each word beginning with a capital letter followed by all lowercase letters.
- `str.swapcase()` – Returns a string with the case of each letter swapped.

```
>>> 'hELLo'.capitalize()
'Hello'
>>> 'hELLo'.lower()
'hello'
>>> 'hELLo'.upper()
'HELLO'
>>> 'hello world'.title()
'Hello World'
>>> 'hELLo'.swapcase()
'Hello'
```

`str.replace(old, new[, count])`

A string with `old` replaced by `new` `count` times.

```
>>> 'mommy'.replace('m', 'b')
'bobby'
>>> 'mommy'.replace('m', 'b', 2)
'bobmy'
```

Square Brackets in Code Notation

As we mentioned in the **Math** lesson (see page 59), square brackets in code notation indicate that the contained portion is optional. To illustrate, consider the `str.replace()` method:

```
str.replace(old, new[, count])
```

This means that the `count` parameter, which indicates the maximum number of replacements to be made, is optional.

The syntax allows for nesting optional parameters within optional parameters. For example:

```
str.find(sub[, start[, end]])
```

This syntax indicates that `end` cannot be specified unless `start` is also specified. The outside square brackets in `[, start[, end]]` indicate that the whole section is optional. The inside square brackets indicate that `end` is optional even after `start` is specified. If it were written as `[start, end]`, it would indicate that `start` and `end` are optional, but if one is included, the other must be included as well.

Methods that Strip Characters

- `str.strip([chars])` – Returns a string with leading and trailing chars removed.
- `str.lstrip([chars])` – Returns a string with leading chars removed.
- `str.rstrip([chars])` – Returns a string with trailing chars removed.

`chars` defaults to whitespace.

```
>>> ' hello '.strip()
'hello'
>>> 'hello'.lstrip('h')
'ello'
>>> 'hello'.rstrip('o')
'hell'
```

❖ String Methods that Return a Boolean

- `str.isalnum()` – Returns `True` if all characters are letters or numbers.
- `str.isalpha()` – Returns `True` if all characters are letters.
- `str.islower()` – Returns `True` if string is all lowercase.
- `str.isupper()` – Returns `True` if string is all uppercase.
- `str.istitle()` – Returns `True` if string is title case.


```
>>> 'Hello World!'.isalnum()
False
>>> 'Hello World!'.isalpha()
False
>>> 'hello'.islower()
True
>>> 'HELLO'.isupper()
True
>>> 'Hello World!'.istitle()
True
```

`str.isspace()`

True if string is made up of only whitespace.

```
>>> ' '.isspace()
True
>>> ' hi '.isspace()
False
```

`str.isdigit()`, `str.isdecimal()`, and `str.isnumeric()`

The `str.isdigit()`, `str.isdecimal()`, and `str.isnumeric()` all check to see if a string has only numeric characters.

1. All three will return True if the string contains only Arabic digits (i.e., 0 through 9):

```
'42'.isdigit() # True
'42'.isdecimal() # True
'42'.isnumeric() # True
```

2. All three will return False if any character is non-numeric:

```
'4.2'.isdigit() # False
'4.2'.isdecimal() # False
'4.2'.isnumeric() # False
```

Beyond that, the difference is mostly academic for most Python developers:

1. `'2³'.isnumeric()` and `'2³'.isdigit()` return `True`, but `'2³'.isdecimal()` returns `False`.
2. `'¼'.isnumeric()` returns `True`, but `'¼'.isdigit()` and `'¼'.isdecimal()` return `False`.

So, which should you use? It doesn't make much difference really, but `isdigit()` is the most popular, perhaps because the name most closely matches the intention of the function.

If you're really curious about it, run `strings/Demos/numbers.py`, which will create a `numbers.txt` file in the same folder showing tabular results, like this (but with much more data):

Char	isdigit	isdecimal	isnumeric
0	TRUE	TRUE	TRUE
1	TRUE	TRUE	TRUE
2	TRUE	TRUE	TRUE
3	TRUE	TRUE	TRUE
²	TRUE	FALSE	TRUE
¼	FALSE	FALSE	TRUE

`str.startswith()` and `str.endswith()`

- `str.startswith(prefix[, start[, end]])` – Returns `True` if string starts with `prefix`.
- `str.endswith(prefix[, start[, end]])` – Returns `True` if string ends with `suffix`.

Both of these methods start looking at `start` index and end looking at `end` index if `start` and `end` are specified.

```
>>> 'hello'.startswith('h')
True
>>> 'hello'.endswith('o')
True
```

is... Methods

All the `is...` methods shown in the table above return `False` for *empty* strings, meaning strings with zero length.

❖ String Methods that Return a Number

String Methods that Return a Position (Index) of a Substring

- `str.find(sub[, start[, end]])` – Returns the lowest index where `sub` is found. Returns -1 if `sub` isn't found.
- `str.rfind(sub[, start[, end]])` – Returns the highest index where `sub` is found. Returns -1 if `sub` isn't found.
- `str.index(sub[, start[, end]])` – Same as `find()`, but errors when `sub` is not found.
- `str.rindex(sub[, start[, end]])` – Same as `rfind()`, but errors when `sub` is not found.

All of these methods start looking at `start` index and end looking at `end` index if `start` and `end` are specified.

```
>>> 'Hello World!'.find('l')
2
>>> 'Hello World!'.rfind('l')
9
>>> 'Hello World!'.index('l')
2
>>> 'Hello World!'.rindex('l')
9
```

`str.count(sub[, start[, end]])`

Returns the number of times `sub` is found. Start looking at `start` index and end looking at `end` index.

```
>>> "Hello World!".count('l')
3
```

String Formatting

Python includes powerful options for formatting strings.

❖ The `format()` Method

One common way to format strings is to use the `format()` method combined with the Format Specification Mini-Language²⁰.

Let's start with a simple example and then we'll explain the mini-language in detail:

```
>>> '{0} is an {1} movie!'.format('Monty Python', 'awesome')
'Monty Python is an awesome movie!'
```

The curly braces are used to indicate a replacement field, which takes position arguments specified by index (as in the preceding example) or by name (as in the following example):

```
>>> '{movie} is an {adjective} movie!'.format(movie='Monty Python',
...                                           adjective='awesome')
'Monty Python is an awesome movie!'
```

The field names (position arguments) can be omitted:

```
'{} is an {} movie!'.format('Monty Python', 'awesome')
```

When the field names are omitted, the first replacement field is at index 0, the second at index 1, and so on.

These examples really just show another form of concatenation and could be rewritten like this:

```
'Monty Python' + ' is an ' + awesome + ' movie!'

# or

movie = 'Monty Python'
adjective = 'awesome'
movie + ' is an ' + adjective + ' movie!'
```

When doing a lot of concatenation, using the `format()` method can be cleaner. However, as the name implies, the `format()` method does more than just concatenation; it also can be used to *format* the replacement strings. It is mostly used for formatting numbers.

20. <https://docs.python.org/3/library/string.html#formatspec>

❖ Format Specification

The format specification is separated from the field name or position by a colon (:), like this:

```
{field_name:format_spec}
```

Because the field name is often left out, it is commonly written like this:

```
{:format_spec}
```

The format specification²¹ is:

```
[[fill]align][sign][width][,][.precision][type]
```

That looks a little scary, so let's break it down from right to left.

Type

```
[[fill]align][sign][width][,][.precision][type]
```

Type is specified by a one-letter specifier, like this:

```
'{:s} is an {:s} movie!'.format('Monty Python', 'awesome')
```

The `s` indicates that the replacement field should be formatted as a string. There are many different types, but, unless you are a mathematician or scientist²², the most common types you'll be working with are strings, integers, and floats.

The default formatting for strings and integers are string format (`s`) and decimal integer (`d`), which are generally what you will want, so you can leave the one-letter type specifier off. Consider the following:

21. This is actually a slightly simplified version of the format specification. For the full format specification, see <https://docs.python.org/3/library/string.html#formatspec>.

22. If you are a mathematician or scientist, you can see all the different available types at <https://docs.python.org/3/library/string.html#formatspec>.

Demo 23: strings/Demos/formatting_types.py

```
1.  # Full formatting strings.
2.  sentence = 'On a scale of {0:d} to {1:d}, I give {2:s} a {3:d}.'
3.  sentence = sentence.format(1, 5, 'Monty Python', 6)
4.  print(sentence)
5.
6.  # Simplify by removing field names (indexes).
7.  sentence = 'On a scale of {:d} to {:d}, I give {:s} a {:d}.'
8.  sentence = sentence.format(1, 5, 'Monty Python', 6)
9.  print(sentence)
10.
11. # Further simplify by removing default type specifiers.
12. sentence = 'On a scale of {:} to {:}, I give {:} a {:}.'
13. sentence = sentence.format(1, 5, 'Monty Python', 6)
14. print(sentence)
15.
16. # And with the field name and type specifier gone, we can
17. # remove the colon separator.
18. sentence = 'On a scale of {} to {}, I give {} a {}.'
19. sentence = sentence.format(1, 5, 'Monty Python', 6)
20. print(sentence)
```

Code Explanation

The final line of code has the advantage of being brief, but the disadvantage of being obscure. As a rule, we prefer clarity over brevity. We can make it clearer using field names:

```
>>> 'On a scale of {low} to {high}, {movie} is a {rating}.'.format(
    low=1, high=5, movie='Monty Python', rating=6)
'On a scale of 1 to 5, Monty Python is a 6.'
```

Floats

You will typically format floats as fixed point numbers using `f` as the one-letter specifier, which has a default precision of 6. If neither type nor precision is specified, floats will be as precise as they need to be to accurately represent the value.

Fixed point type specified:

```
>>> import math
>>> 'pi equals {:.f}'.format(math.pi)
'pi equals 3.141593'
```

No type specified:

```
>>> import math
>>> 'pi equals {}'.format(math.pi)
'pi equals 3.141592653589793'
```

Another formatting option for floats is percentage (%). We will cover that shortly.

Precision

```
[[fill]align][sign][width][,][.precision][type]
```

The precision is specified before the type and is preceded by a decimal point, like this:

```
>>> import math
>>> 'pi equals {:.2f}'.format(math.pi)
'pi equals 3.14'
```

Separating the Thousands

```
[[fill]align][sign][width][,][.precision][type]
```

Insert a comma before the precision value to separate the thousands with commas, like this:

```
>>> '{:,.2f}'.format(1000000)
'1,000,000.00'
```

Width

```
[[fill]align][sign][width][,][.precision][type]
```

The width is an integer indicating the minimum number of characters of the resulting string. If the passed-in string has fewer characters than the specified width, *padding* will be added. By default, padding is added *after* strings and *before* numbers, so that strings are aligned to the left and numbers are aligned to the right.

Here are some examples:

```
>>> '{:5}'.format('abc')
'abc '
>>> '{:5}'.format(123)
' 123'
>>> '{:5.2f}'.format(123)
'123.00'
```

In all three cases, the width of the formatted string is set to 5. Notice the padding on the first two examples: after the string and before the number.

In the final example, we format the number 123, but the format type has been specified as fixed point (f) with a precision of 2. So, the resulting string ('123.00') is six characters long – longer than the specified width, so it just returns the full string without padding.

Sign

```
[[fill]align][sign][width][,][.precision][type]
```

By default, negative numbers are preceded by a negative sign, but positive numbers are not preceded by a positive sign. To force the sign to show up, add a plus sign (+) before the precision, like this:

```
>>> 'pi equals {:.2f}'.format(math.pi)
'pi equals +3.14'
```


Alignment

```
[[fill]align][sign][width][,][.precision][type]
```

You can change the default alignment by preceding the width (and sign if there is one) with one of the following options:

Alignment

Options	Meaning
<	Left aligned (default for strings).
>	Right aligned (default for numbers).
=	Padding added between sign and digits. Only valid for numbers.
^	Centered.

Some examples:

```
>>> '{:>5}'.format('abc')
'   abc'

>>> '{:<5}'.format(123)
'123   '

>>> '{:^5}'.format(123)
'  123  '

>>> '{:=+5}'.format(123)
'+ 123'
```

Fill

```
[[fill]align][sign][width][,][.precision][type]
```

By default, spaces are used for padding, but this can be changed by inserting a fill character before the alignment option, like this (note the period after the colon):

```
>>> '{: ^10.2f}'.format(math.pi)
'...3.14...'
```

And now with a dash:

```
>>> '{:- ^10.2f}'.format(math.pi)
'---3.14---
```

Percentage Type

As mentioned earlier, another option for type is percentage (%):

```
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> grade
0.72
>>> '{: .2f}'.format(grade)
'0.72'
>>> '{: .2%}'.format(grade)
'72.00%'
>>> '{: .0%}'.format(grade)
'72%'
```

❖ Long Lines of Code

The Python Style Guide²³ suggests that lines of code should be limited to 79 characters. This can be difficult as each line of code is considered a new statement; however, it can usually be accomplished through some combination of:

1. Breaking method arguments across multiple lines.
2. Concatenation.
3. Triple-quoted multi-line strings.

All three methods are shown in the following file:

23. <https://www.python.org/dev/peps/pep-0008/#maximum-line-length>

Demo 24: strings/Demos/long_code_lines.py

```
1.  # EXAMPLE 1: Breaking method arguments across multiple lines
2.  phrase = ("On a scale of {} to {}, I give {} a {}".format(1, 5,
3.                                                    "Monty Python", 6))
4.  print(phrase)
5.
6.  location = "ponds"
7.  items = "swords"
8.  beings = "masses"
9.  adjective = "farcical"
10.
11. # EXAMPLE 2: Concatenation
12. quote = ("Listen, strange women lyin' in {} " +
13.          "distributin' {} is no basis for a system of " +
14.          "government. Supreme executive power derives from " +
15.          "a mandate from the {}, not from some {} " +
16.          "aquatic ceremony.").format(location, items,
17.                                     beings, adjective)
18.
19. print(quote)
20.
21. # EXAMPLE 3: Triple quotes
22. quote = """Listen, strange women lyin' in {} \
23. distributin' {} is no basis for a system of \
24. government. Supreme executive power derives from \
25. a mandate from the {}, not from some {} \
26. aquatic ceremony.""".format(location, items,
27.                             beings, adjective)
28.
29. print(quote)
```

Code Explanation

Remember that the backwards slashes at the end of each line escape the newline character.

Also, notice that the arguments passed to `format()` are broken across lines and vertically aligned to make it clear that they are related.

Run the file to see the resulting strings.



Exercise 14: Playing with Formatting

🕒 10 to 20 minutes

In this exercise, you will practice formatting strings. Here are two options for practicing:

Option 1

1. Run `'{}'.format('')` at the Python shell.
2. Try formatting different values in different ways. For example, try running:

```
>>> '{:.0%}'.format(.87)
```

Option 2

1. From strings/Demos run `python formatter.py`
2. Try different format specifications with different numbers, like this:

```
Format to try: {:f}
Entry to format: math.pi
Result: 3.141593
Enter for another or 'q' to quit.
Format to try: {}
Entry to format: math.pi
Result: 3.141592653589793
Enter for another or 'q' to quit.
Format to try: {:.2f}
Entry to format: math.pi
Result: 3.14
Enter for another or 'q' to quit.
Format to try: {:.2f}
Entry to format: 1000000
Result: 1000000.00
Enter for another or 'q' to quit.
Format to try: {:,.2f}
Entry to format: 1000000
Result: 1,000,000.00
Enter for another or 'q' to quit.
Format to try: {:>20}
Entry to format: 'abc'
Result: abc
```

Formatted String Literals (f-strings)

Formatted string literals or *f-strings*²⁴ use a syntax that implicitly embeds the `format()` function within the string itself. The coding tends to be less verbose.

The following demonstration compares string concatenation and string formatting with the f-string syntax.

Demo 25: strings/Demos/f_strings.py

```
1.  import math
2.  user_name = input("What is your name? ")
3.
4.  # Concatenation:
5.  greeting = "Hello, " + user_name + "!"
6.
7.  # The format() method:
8.  greeting = "Hello, {}".format(user_name)
9.
10. # f-string:
11. greeting = f"Hello, {user_name}!"
12. print(greeting)
13.
14. # format specification is also available:
15. pi_statement = f"pi is {math.pi:.4f}"
16. print(pi_statement)
```

Code Explanation

The curly braces within the f-string contain the variable name and optionally a format specification. The string literal is prepended with an `f`.

Everything you learned earlier about formatting can be applied to the f-string because the same formatting function is called. Practice with f-strings by running the following lines of code:

24. f-strings were introduced in Python 3.6.

```

>>> import math
>>> movie = 'Monty Python'
>>> adjective = 'awesome'
>>> f'{movie} is an {adjective} movie!'
'Monty Python is an awesome movie!'
>>> low = 1
>>> high = 5
>>> rating = 6
>>> f'On a scale of {low} to {high}, {movie} is a {rating}.'
'On a scale of 1 to 5, Monty Python is a 6.'
>>> f'pi equals {math.pi:f}'
'pi equals 3.141593'
>>> f'pi equals {math.pi}'
'pi equals 3.141592653589793'
>>> f'pi equals {math.pi:.2f}'
'pi equals 3.14'
>>> f'{1000000:,.2f}'
'1,000,000.00'
>>> f"{'abc':20}"
'abc'
>>> f'{123:20}'
'123'
>>> f'{123:5.2f}'
'123.00'
>>> f'pi equals {math.pi:+.2f}'
'pi equals +3.14'
>>> f"{'abc':>20}"
'abc'
>>> f'{123:<20}'
'123'
>>> f'{123:=+20}'
'+123'
>>> f'pi equals {math.pi:.^10.2f}'
'pi equals ...3.14...'
>>> f'pi equals {math.pi:-^10.2f}'
'pi equals ---3.14---'
>>>
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> f'{grade:.2f}'
'0.72'
>>> f'{grade:.2%}'
'72.00%'

```

```
>>> f'{grade:.2f}'  
'0.72'
```

Built-in String Functions

str(object)

Converts object to a string.

```
>>> str('foo') # 'foo' is already a string  
'foo'  
>>> str(999)  
'999'  
>>> import math  
>>> str(math.pi)  
'3.141592653589793'
```

len(string)

Returns the number of characters in the string.²⁵

```
>>> len('foo')  
3
```

min() and max()²⁶

The `min(args)` function returns the smallest value of the passed-in arguments.

The `max(args)` function returns the largest value of the passed-in arguments.

^{25.} As we will see later, the `len()` function can also take objects other than strings.

^{26.} The `min()` and `max()` functions can also compare numbers.

```
>>> min('w', 'e', 'b')
'b'
>>> min('a', 'B', 'c')
'B'
>>> max('w', 'e', 'b')
'w'
>>> max('a', 'B', 'c')
'c'
```

Note that all uppercase letters come before lowercase letters (e.g., `min('Z', 'a')` returns `'Z'`).

`min()` and `max()` with Iterables

The `min()` and `max()` functions can also take an *iterable* containing values to compare. We will cover this in the **Iterables** lesson (see page 130).



Exercise 15: Outputting Tab-delimited Text

🕒 25 to 40 minutes

In this exercise, you will write a program that repeatedly prompts the user for a Company name, Revenue, and Expenses and then outputs all the information as tab-delimited text. Here is the program after it has run:

```
Company: Pepperpots
Revenue: 1200000
Expenses: 999002
Again? Press ENTER to add a row or Q to quit.
Company: Ni Knights
Revenue: 19
Expenses: 24
Again? Press ENTER to add a row or Q to quit.
Company: Round Knights
Revenue: 777383
Expenses: 777382
Again? Press ENTER to add a row or Q to quit. q
```

Company	Revenue	Expenses	Profit
Pepperpots	\$1,200,000.00	\$999,002.00	\$200,998.00
Ni Knights	\$ 19.00	\$ 24.00	\$ -5.00
Round Knights	\$777,383.00	\$777,382.00	\$ 1.00

In this exercise, you can use the `format()` method or f-strings or any combination of the two.

1. Open `strings/Exercises/tab_delimited_text.py`.
2. Modify the `addheaders()` function so that it creates a header row and appends it to `_output`. The four headers should each take up 10 spaces, be aligned to the center, and be separated by tabs, like this:

```
' Company \t Revenue \t Expenses \t Profit \n'
```

Don't just copy that string. Use the `format()` method.

3. Modify the `addrows()` function so that it adds a row to `_output` by prompting the user for values for company, revenue, and expenses, and then calculating profit.
 - A. All "columns" should be 10 spaces wide.
 - B. The company name should be a left-aligned string.

- C. The other three columns should be formatted in U.S. dollars (e.g., \$1,200,000.00) and right-aligned.
4. Save and run the file. Try entering data for at least three companies.

Exercise Code: strings/Exercises/tab_delimited_text.py

```
1.  _output = ""
2.
3.  def addheaders():
4.      # Write your code here
5.      pass
6.
7.  def addrow():
8.      # Write your code here
9.
10.     # The rest of the function prompts the user to add another row
11.     # or quit. On quitting, it prints _output. Leave it as is.
12.
13.     again = input("Again? Press ENTER to add a row or Q to quit. ")
14.     if again.lower() != "q":
15.         addrow()
16.     else:
17.         print(_output)
18.
19.  def main():
20.      # Call addheaders() and addrow()
21.      addheaders()
22.      addrow()
23.
24.  main()
```

Solution: strings/Solutions/tab_delimited_text.py

```
1.  _output = ""
2.
3.  def add_headers():
4.      global _output
5.      c_header = "{:^10}".format("Company")
6.      r_header = "{:^10}".format("Revenue")
7.      e_header = "{:^10}".format("Expenses")
8.      p_header = "{:^10}".format("Profit")
9.      _output += "{}\t{}\t{}\t{}\n".format(c_header, r_header,
10.                                          e_header, p_header)
11.
12.  def add_row():
13.      global _output
14.
15.      c = input("Company: ")
16.      r = float(input("Revenue: "))
17.      e = float(input("Expenses: "))
18.      p = r - e # profit
19.
20.      c_str = "{:<10}".format(c)
21.      r_str = "${:>10,.2f}".format(r)
22.      e_str = "${:>10,.2f}".format(e)
23.      p_str = "${:>10,.2f}".format(p)
24.
25.      new_row = "{}\t{}\t{}\t{}\n".format(c_str, r_str, e_str, p_str)
26.
27.      _output += new_row
    -----Lines 28 through 39 Omitted-----
```

Solution: strings/Solutions/tab_delimited_text_f_string.py

```
1.  _output = ""
2.
3.  def add_headers():
4.      global _output
5.      c_header = f"{'Company':^10}"
6.      r_header = f"{'Revenue':^10}"
7.      e_header = f"{'Expenses':^10}"
8.      p_header = f"{'Profit':^10}"
9.      _output += f"{c_header}\t{r_header}\t{e_header}\t{p_header}\n"
10.
11. def add_row():
12.     global _output
13.
14.     c = input("Company: ")
15.     r = float(input("Revenue: "))
16.     e = float(input("Expenses: "))
17.     p = r - e # profit
18.
19.     c_str = f"{c:<10}"
20.     r_str = f"${r:>10,.2f}"
21.     e_str = f"${e:>10,.2f}"
22.     p_str = f"${p:>10,.2f}"
23.
24.     new_row = f"{c_str}\t{r_str}\t{e_str}\t{p_str}\n"
25.
26.     _output += new_row
    -----Lines 27 through 38 Omitted-----
```

Conclusion

In this lesson, you have learned to manipulate and format strings.

LESSON 5

Iterables: Sequences, Dictionaries, and Sets

Topics Covered

- ☑ Types of iterables available in Python.
- ☑ Lists.
- ☑ Tuples.
- ☑ Ranges.
- ☑ Dictionaries.
- ☑ Sets.
- ☑ The `*args` and `**kwargs` parameters.

I did not like this iteration of one idea--this strange recurrence of one image, and I grew nervous as bedtime approached and the hour of the vision drew near.

— *Jane Eyre*, Charlotte Bronte

Introduction

Iterables are objects that can return their members one at a time. The iterables we will cover in this lesson are lists, tuples, ranges, dictionaries, and sets.

Definitions

Here are some quick definitions to provide an overview of the different types of objects we will be covering in this lesson. Don't worry if the meanings aren't entirely clear now. They will be when you finish the lesson.

1. *Sequences* are iterables that can return members based on their position within the iterable. Examples of sequences are strings, lists, tuples, and ranges.

2. *Lists* are *mutable* (changeable) sequences similar to arrays in other programming languages.
3. *Tuples* are *immutable* sequences.
4. *Ranges* are *immutable* sequences of numbers often used in *for loops*.
5. *Dictionaries* are mappings that use arbitrary keys to map to values. Dictionaries are like associative arrays in other programming languages.
6. *Sets* are *mutable* unordered collections of distinct *immutable* objects. So, while the set itself can be modified, it cannot be populated with objects that can be modified.

Sequences

Sequences are iterables that can return members based on their position within the iterable. You have already learned about one type of sequence: *strings*. Remember string indexing (see page 74):

```
>>> 'Hello, world!'[1]
'e'
```

That's one way of getting at one member of a sequence.

The sequences we cover in this lesson are:

1. Lists
2. Tuples
3. Ranges

Lists

Python's lists are similar to arrays in other languages. Lists are created using square brackets, like this:

```
colors = ["red", "blue", "green", "orange"]
```

❖ List Methods

Some of the most common list methods are shown in the following list:

- `mylist.append(x)` – Appends `x` to `mylist`.

- `mylist.remove(x)` – Removes first element with value of `x` from `mylist`. Errors if no such element is found.
- `mylist.insert(i, x)` – Inserts `x` at position `i`.
- `mylist.count(x)` – Returns the number of times that `x` appears in `mylist`.
- `mylist.index(x)` – Returns the index position of the first element in `mylist` whose value is `x` or a `ValueError` if no such element exists.
- `mylist.sort()` – Sorts `mylist`.
- `mylist.reverse()` – Reverses the order of `mylist`.
- `mylist.pop(n)` – Removes and returns the element at position `n` in `mylist`. If `n` is not passed in, the last element in the list is popped (removed and returned).
- `mylist.clear()` – Removes all elements from `mylist`.
- `mylist.copy()` – Returns a copy of `mylist`.
- `mylist.extend(anotherlist)` – Appends `anotherlist` onto `mylist`.

The following code illustrates how these methods are used. Try this out at the Python shell:

```

>>> colors = ["red", "blue", "green", "orange"]
>>> colors
['red', 'blue', 'green', 'orange']
>>> colors.append("purple") # Append purple to colors
>>> colors
['red', 'blue', 'green', 'orange', 'purple']
>>> colors.remove("green") # Remove green from colors
>>> colors
['red', 'blue', 'orange', 'purple']
>>> colors.insert(2, "yellow") # Insert yellow in position 2
>>> colors
['red', 'blue', 'yellow', 'orange', 'purple']
>>> colors.index("orange") # Get position of orange
3
>>> colors.sort() # Sort colors in place
>>> colors
['blue', 'orange', 'purple', 'red', 'yellow']
>>> colors.reverse() # Reverse order of colors
>>> colors
['yellow', 'red', 'purple', 'orange', 'blue']
>>> colors.pop() # Remove and return last element
'blue'
>>> colors.pop(1) # Remove and return element at position 1
'red'
>>> colors # Notice blue and red have been removed
['yellow', 'purple', 'orange']
>>> colors_copy = colors.copy() # Create a copy of colors
>>> colors_copy
['yellow', 'purple', 'orange']
>>> colors.extend(colors_copy) # Append colors_copy to colors
>>> colors
['yellow', 'purple', 'orange', 'yellow', 'purple', 'orange']
>>> colors_copy.clear() # Delete all elements from colors_copy
>>> colors_copy # Notice colors_copy is now empty
[]
>>> del colors_copy # Delete colors_copy
>>> colors_copy # It's gone
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'colors_copy' is not defined

```

❖ Copying a List

Note again how we made a copy of the `colors` list:

```
>>> colors = ["red", "blue", "green", "orange"]
>>> colors_copy = colors.copy()
>>> colors_copy
['red', 'blue', 'green', 'orange']
>>> colors_copy.sort()
>>> colors_copy
['blue', 'green', 'orange', 'red']
>>> colors
['red', 'blue', 'green', 'orange'] # colors remains unsorted
```

You can see that sorting `colors_copy` has no effect on the `colors` list. They are two distinct objects. Compare that to the following, which continues from the code above:

```
>>> colors_copy2 = colors
>>> colors_copy2
['red', 'blue', 'green', 'orange']
>>> colors_copy2.sort()
>>> colors_copy2
['blue', 'green', 'orange', 'red']
>>> colors
['blue', 'green', 'orange', 'red'] # Wait! colors is sorted too!
>>> id(colors)
2147162345152
>>> id(colors_copy)
2147163702400 # different id than colors
>>> id(colors_copy2)
2147162345152 # the same id as colors
```

Do you see what's going on here? When you assign one variable to another, instead of making a copy, it just creates a pointer to that object. If you modify one object, it affects both. The `copy()` method creates a brand new identical object, so modifying the new object has no effect on the original.

❖ Deleting List Elements

The `del` statement can be used to delete elements or slices of elements from a list, like this:

Demo 26: iterables/Demos/del_list.py

```
1. colors = ['red', 'blue', 'green', 'orange', 'black']
2.
3. del colors[0] # deletes first element
4. print(colors) # ['blue', 'green', 'orange', 'black']
5.
6. del colors[1:3] # deletes 2nd and 3rd elements
7. print(colors) # ['blue', 'black']
```

Sequences and Random

In the **Math** lesson, when we learned about the random module (see page 62), we mentioned two methods that we were not yet ready to explore. Now, we are:

random.choice(seq)

Returns random element in seq.

```
>>> import random
>>> colors = ["red", "blue", "green", "orange"]
>>> random.choice(colors)
'orange'
>>> random.choice(colors)
'green'
```

random.shuffle(seq)

Shuffles seq in place.

```
>>> import random
>>> colors = ["red", "blue", "green", "orange"]
>>> random.shuffle(colors)
>>> colors
['green', 'red', 'blue', 'orange']
```

Exercise 16: Remove and Return Random Element

 10 to 20 minutes

In this exercise, you will write a `remove_random()` function that removes a random element from a list and returns it.

1. Open `iterables/Exercises/remove_random.py` in your editor.
2. Write the code for the `remove_random()` function so that it removes and returns a random element from the passed-in list: `the_list`.
3. Modify the `main()` function so that it uses `remove_random()` to remove a random element from the `colors` list and then prints something like the following:

```
The removed color was green.  
The remaining colors are ['red', 'blue', 'orange'].
```

Exercise Code: `iterables/Exercises/remove_random.py`

```
1.  import random  
2.  
3.  def remove_random(the_list):  
4.      pass # replace this with your code  
5.  
6.  def main():  
7.      colors = ['red', 'blue', 'green', 'orange']  
8.      # Your code here  
9.  
10. main()
```

Solution: iterables/Solutions/remove_random.py

```
1. import random
2.
3. def remove_random(the_list):
4.     x = random.choice(the_list)
5.     the_list.remove(x)
6.     return x
7.
8. def main():
9.     colors = ['red', 'blue', 'green', 'orange']
10.    removed_color = remove_random(colors)
11.    print(f'The removed color was {removed_color}.')
12.    print(f'The remaining colors are {colors}.')
13.
14. main()
```

Tuples

Tuples are like lists, but they are *immutable*: once created, they cannot be changed.

Get ready for a lie: Tuples are created using parentheses, like this:

```
MAGENTA = (255, 0, 255)
```

Wait, what?!! Why are you lying to me?

OK, the truth is that tuples are created *with commas* AND don't *require* parentheses. You *can* create a tuple like this:

```
MAGENTA = 255, 0, 255 # Avoid this
```

But just because you *can* doesn't mean you *should*. It's a better idea to get used to including the parentheses, because sometimes you do need them. To illustrate, take a look at the following code:

Demo 27: iterables/Demos/tuples.py

```
1. def show_type(obj):
2.     print(type(obj))
3.
4. # tuple created w/o parens (works but bad practice)
5. MAGENTA = 255, 0, 255
6. show_type(MAGENTA)
7.
8. # When passing a tuple to a function, you need parens:
9. show_type((255, 0, 255))
10.
11. # Passing the tuple w/o parens to a function will error
12. show_type(255, 0, 255)
```

Code Explanation

The preceding code will render the following:

```
<class 'tuple'>
<class 'tuple'>
Traceback (most recent call last):
  File "c:/Webucator/Python/iterables/Demos/tuples.py", line 12, in <module>
    show_type( 255, 0, 255 )
TypeError: show_type() takes 1 positional argument but 3 were given
```

1. On line 5, the MAGENTA tuple is created without using parentheses.
2. By passing MAGENTA to the show_type() function, we see that MAGENTA is indeed a tuple.
3. On line 9, the tuple (255, 0, 255) (constructed with parentheses) is passed to the show_type() function. This works fine.
4. On line 12, the tuple (*well, not really*) 255, 0, 255 (constructed without parentheses) is passed to the show_type() function. In this case, Python passes the values to the show_type() function as three separate arguments. As the function only expects one argument, this results in an error:

```
TypeError: show_type() takes 1 positional argument but 3 were given
```

The takeaway here is: *Always use parentheses when creating tuples.*

Remember Constants

The `MAGENTA` variable above makes a good constant (see page 18). The values represent the amounts of red, green, and blue in the color magenta.

❖ When to Use a Tuple

While lists are used for holding collections of like data, tuples are meant for holding *heterogeneous collections of data*. In tuples, the position of the element is meaningful. To illustrate, let's consider our `MAGENTA` constant again:

```
MAGENTA = (255, 0, 255)
```

The values in the tuple correspond to RGB (red, green, blue) color values. Magenta is created by mixing full red (255) with full blue (255) and no green (0).

Other use cases for tuples include:

1. X-Y Coordinates (e.g, `(55, -23)`)
2. Latitude-Longitude Coordinates (e.g., `(43.0298, -76.0044)`)
3. Geometric Shapes (notice that these are tuples of tuples):

```
line = ((-40, 10), (-80, 170))
triangle = ((140, 200), (180, 270), (335, 180))
rectangle = ((40, 100), (80, 170), (235, 80), (195, 10))
```

Turtle Graphics

For a little fun, open and run the `iterables/Demos/shapes.py` file. It uses `turtle` (<https://docs.python.org/3/library/turtle.html>), a built-in graphics library, to draw shapes from tuples of coordinates.

Empty and Single-element Tuples

You're unlikely to have to create empty or single-element tuples very often, but in case you do...

Empty Tuple

```
t_empty = ()
```

Single Element Tuple

To create a single-element tuple, follow the element with a comma, like this:

```
t_single = ("a",)
```

If you do not include the comma, you just get a string as illustrated in the following code:

```
>>> t1 = ("a",)
>>> type(t1)
<class 'tuple'>
>>> t2 = ("a")
>>> type(t2)
<class 'str'>
```

Ranges

A range is an immutable sequence of numbers often used in for loops, which will be covered in the **Flow Control** lesson (see page 170). Ranges are created using `range()`, which can take one, two, or three arguments:

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

The following examples show the options for creating a range:

```
range(10) # range starting at 0 and ending at 9
range(5, 11) # range starting at 5 and ending at 10
range(0, 13, 3) # range starting at 0, ending at 12, in steps of 3
range(4, -4, -1) # range starting at 4, ending at -3, in steps of -1
```

Note that the stop number is not included in the range. You should read it as "from start up to *but not including* stop."

Converting Sequences to Lists

The easiest way to see how ranges work in the Python shell is to convert them into lists first as shown in the following code:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 11))
[5, 6, 7, 8, 9, 10]
>>> list(range(0, 13, 3))
[0, 3, 6, 9, 12]
>>> list(range(-4, 4))
[-4, -3, -2, -1, 0, 1, 2, 3]
```

We will revisit ranges when we discuss for loops.

Converting a Sequence to a List

You can convert any type of sequence to a list with the `list()` function:

```
>>> coords = (55, -23)
>>> list(coords)
[55, -23]
>>> list("Hello, world!")
['H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
```

Indexing

In the **Strings** lesson, you learned how to find specific characters within a string using indexing and slicing (see page 74). All sequences can be indexed and sliced in this way.

Again, indexing is the process of finding a specific element within a sequence of elements through the element's position. If we consider a sequence from left to right, the first element (the left-most) is at position 0. If we consider a sequence from right to left, the first element (the right-most) is at position -1.

The following code shows how to use indexing with a list, but the same can be done with any sequence type. Try this out in the Python shell:

```
>>> fruit = ['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
>>> fruit[0]
'apple'
>>> fruit[-1]
'watermelon'
>>> fruit[4]
'lemon'
>>> fruit[-3]
'pear'
```



Exercise 17: Simple Rock, Paper, Scissors Game

🕒 15 to 20 minutes

1. Open `iterables/Exercises/roshambo.py` in your editor.
2. Write the `main()` function so that it:
 - A. Creates a sequence with three elements: “Rock”, “Paper”, and “Scissors”.
 - B. Makes a random choice for the computer and stores it in a variable.
 - C. Prompts the user with 1 for Rock, 2 for Paper, 3 for Scissors:
 - D. Prints out the computer’s choice and then the user’s choice.

The program should run like this:

```
1 for Rock, 2 for Paper, 3 for Scissors: 1
Computer: Scissors
User: Rock
```

Exercise Code: `iterables/Exercises/roshambo.py`

```
1. import random
2.
3. def main():
4.     pass # replace this with your code
5.
6. main()
```


Solution: iterables/Solutions/roshambo.py

```
1. import random
2.
3. def main():
4.     roshambo = ["Rock", "Paper", "Scissors"]
5.
6.     computer_choice = random.choice(roshambo)
7.
8.     num = input("1 for Rock, 2 for Paper, 3 for Scissors: ")
9.     num = int(num) - 1
10.    user_choice = roshambo[num]
11.
12.    print("Computer:", computer_choice)
13.    print("User:", user_choice)
14.
15.    main()
```

Slicing

Slicing is the process of getting a slice or segment of a sequence as a new sequence. The syntax is as follows:

```
sub_sequence = orig_sequence[first_pos:last_pos]
```

This returns a slice that starts with the element at `first_pos` and includes all the elements up to *but not including* the element at `last_pos`.

If `first_pos` is left out, then it is assumed to be 0. For example:

```
>>> ["a", "b", "c", "d", "e"][:3]
['a', 'b', 'c']
```

If `last_pos` is left out, then it is assumed to be the length of the sequence, or in other words, one more than the last position of the sequence. For example:

```
>>> ["a", "b", "c", "d", "e"][3:]  
['d', 'e']
```

The following code shows how to slice a list, but the same can be done with any sequence type. Try this out in the Python shell:

```
>>> fruit = ["apple", "orange", "banana", "pear", "lemon", "watermelon"]  
>>> fruit[0:5]  
['apple', 'orange', 'banana', 'pear', 'lemon']  
>>> fruit[1:4]  
['orange', 'banana', 'pear']  
>>> fruit[4:]  
['lemon', 'watermelon']  
>>> fruit[-3:]  
['pear', 'lemon', 'watermelon']  
>>> fruit[:3]  
['apple', 'orange', 'banana']  
>>> fruit[-4:-1]  
['banana', 'pear', 'lemon']  
>>> fruit[:]  
['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
```



Exercise 18: Slicing Sequences

⌚ 10 to 20 minutes

1. Open `iterables/Exercises/slicing.py` in your editor.
2. Write the `split_list()` function so that it returns a list that contains two lists: the first and second half of the original list. For example, when passed `[1, 2, 3, 4]`, `split_list()` will return `[[1, 2], [3, 4]]`.
3. If the original list has an odd number of elements, the function should put the extra element in the first list. For example, when passed `[1, 2, 3, 4, 5]`, `split_list()` will return `[[1, 2, 3], [4, 5]]`.

When you run the program, it should output:

```
["red", "blue", "green"]
['orange', 'purple']
```

Exercise Code: `iterables/Exercises/slicing.py`

```
1.  import math
2.
3.  def split_list(orig_list):
4.      pass # replace this with your code
5.
6.  def main():
7.      colors = ["red", "blue", "green", "orange", "purple"]
8.      colors_split = split_list(colors)
9.      print(colors_split[0])
10.     print(colors_split[1])
11.
12.  main()
```


Solution: iterables/Solutions/slicing.py

```
1.  import math
2.
3.  def split_list(orig_list):
4.      list_len = len(orig_list)
5.      mid_pos = math.ceil(list_len/2)
6.      list1 = orig_list[:mid_pos]
7.      list2 = orig_list[mid_pos:]
8.      return [list1, list2]
    -----Lines 9 through 16 Omitted-----
```

min(), max(), and sum()

min(iter) and max(iter)

The `min(iter)` function returns the smallest value of the passed-in iterable.

The `max(iter)` function returns the largest value of the passed-in iterable.

```
>>> colors = ["red", "blue", "green", "orange", "purple"]
>>> min(colors)
'blue'
>>> max(colors)
'red'
>>> ages = [27, 4, 15, 99, 33, 25]
>>> min(ages)
4
>>> max(ages)
99
```

Note that, for strings, uppercase letters are “smaller” than lowercase letters:

```
>>> min("NatDunn")
'D'
```

min() and max() with Multiple Arguments

The `min()` and `max()` functions can also take multiple arguments to compare. This is covered in the Math (see page 57) and Strings (see page 103) lessons.

sum(iter[, start])

The `sum()` function takes an iterable and adds up all of its elements and then adds the result to `start` (if it is passed in). For example:

```
>>> nums = range(1, 6)
>>> sum(nums) # 1 + 2 + 3 + 4 + 5
15
>>> sum(nums, 10)
25
```

Converting Sequences to Strings with str.join(seq)

The `join()` method of a string joins the elements of a sequence of strings on the given string. For example:

```
>>> colors = ["red", "blue", "green", "orange"]
>>> ','.join(colors)
'red,blue,green,orange'
>>> ', '.join(colors) # space after comma
'red, blue, green, orange'
>>> ':'.join(colors)
'red:blue:green:orange'
>>> ' '.join(colors)
'red blue green orange'
```

Note, the `join()` method will error if any of the elements in the sequence is not a string.

Splitting Strings into Lists

The `split()` method of a string splits the string into substrings. By default it splits on whitespace. For example:

```
>>> sentence = 'We are no longer the Knights Who Say "Ni!'"
>>> list_of_words = sentence.split()
>>> list_of_words
['We', 'are', 'no', 'longer', 'the', 'Knights', 'Who', 'Say', '"Ni!"']
```

`split()` takes an optional `sep` parameter to specify the separator:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(",")
['apple', ' banana', ' pear', ' melon']
```

Notice the extra space before “banana”, “pear”, and “melon”. To get rid of that space, you can specify a multi-character separator, like this:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(", ")
['apple', 'banana', 'pear', 'melon']
```

`split()` takes a second optional parameter, `maxsplit`, to indicate the maximum number of times to split the string. For example:

```
>>> fruit = "apple, banana, pear, melon"
>>> fruit.split(", ", 2)
['apple', 'banana', 'pear, melon']
```

Notice `pear` and `melon` are part of the same string element. That’s because the string stopped splitting after two splits.

The `splitlines()` Method

The `splitlines()` method of a string splits a string into a list on line boundaries (i.e., line feeds (`\n`) and carriage returns (`\r`)). For example:

```
>>> fruit = """apple
banana
pear
melon"""
>>> fruit.splitlines()
['apple', 'banana', 'pear', 'melon']
```

The `splitlines()` method is really useful when reading files. Consider the following text file:

Demo 28: iterables/data/states.txt

```
1.  Alabama AL
2.  Alaska AK
3.  Arizona AZ
4.  Arkansas AR
5.  California CA
-----Lines 6 through 50 Omitted-----
```

Code Explanation

The file is a listing of states in the United States. Each line lists the name of the state and its abbreviation, separated by a newline.

The following code can be used to read this file into a list:

Demo 29: iterables/Demos/states.py

```
1.  def main():
2.      with open('../data/states.txt') as f:
3.          states = f.read().splitlines()
4.
5.      print(f'The file contains {len(states)} states.')
6.
7.  main()
```

Code Explanation

This will read the states from the file into a list. Run the file:

```
PS ..\iterables\Demos> python states.py
The file contains 50 states.
```

As you will soon see, having the data in a list makes it much easier to work with.

Unpacking Sequences

We learned earlier about simultaneous assignment (see page 16), which allows you to assign values to multiple variables at once, like this:

```
>>> first_name, last_name, company = "Nat", "Dunn", "Webucator"
>>> first_name
'Nat'
>>> last_name
'Dunn'
>>> company
'Webucator'
```

You can use the same concept to *unpack* a sequence into multiple variables, like this:

```
>>> about_me = ("Nat", "Dunn", "Webucator")
>>> first_name, last_name, company = about_me
>>> first_name
'Nat'
>>> last_name
'Dunn'
>>> company
'Webucator'
```

Dictionaries

Dictionaries are mappings that use arbitrary keys to map to values. They are like associative arrays in other programming languages. Dictionaries are created with curly braces and comma-delimited key-value pairs, like this:

```
>>> dict = {
    'key1': 'value 1',
    'key2': 'value 2',
    'key3': 'value 3'
}
>>> dict['key2'] = 'new value 2' # assign new value to existing key
>>> dict['key4'] = 'value 4' # assign value to new key
>>> print(dict['key1']) # print value of key
value 1
```

And here is a file showing a dictionary with meaningful content:

Demo 30: iterables/Demos/dict.py

```
1.  grades = {
2.      "English": 97,
3.      "Math": 93,
4.      "Global Studies": 85,
5.      "Art": 74,
6.      "Music": 86
7.  }
8.
9.  grades["Global Studies"] = 87 # assign new value to existing key
10. grades["Gym"] = 100 # assign value to new key
11.
12. print(grades["Math"]) # print value of key
```

❖ Common Dictionary Methods

The code samples for the methods that follow assume this dictionary:

```
grades = {
    "English": 97,
    "Math": 93,
    "Art": 74,
    "Music": 86
}
```

mydict.get(key[, default])

Returns the value for key if it is in mydict. Otherwise, it returns default if passed in or None if it is not.

```
>>> grades.get('English')
97
>>> grades.get('French') # returns None
>>> grades.get('French', 0)
0
```

mydict.pop(key[, default])

Removes and returns the value of key if it is in mydict. Otherwise, it returns default if passed in or a KeyError if it is not.

```
>>> grades.pop('English')
97
>>> grades # Notice 'English' has been removed
{'Math': 93, 'Art': 74, 'Music': 86}
>>> grades.pop('English')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'English'
>>> grades.pop('English', 'Not found')
'Not found'
```

mydict.popitem()

Removes and returns the last key/value pair entered as a tuple.²⁷

```
>>> grades.popitem()
('Music', 86)
>>> grades # Notice 'Music' has been removed
{'Math': 93, 'Art': 74}
```

²⁷. In versions prior to 3.7, an arbitrary item was removed.

`mydict.copy()`

Returns a copy of `mydict`.

This works just like the `copy()` method of lists (see page 115):

```
>>> grades_copy = grades.copy()
>>> grades['English'] = 94 # Add 'English' back to grades
>>> grades['Music'] = 100 # Add 'Music' back to grades
>>> grades # Notice they're back
{'Math': 93, 'Art': 74, 'English': 94, 'Music': 100}
>>> grades_copy # But grades_copy still has the old data
{'Math': 93, 'Art': 74}
```

`mydict.clear()`

Removes all elements from `mydict`.

```
>>> grades.clear()
>>> grades # Dictionary is now empty
{}
```

`update()`

The `update()` method is used to add new key/value pairs to a dictionary or to overwrite the values of existing keys or both. It can take several types of arguments. Consider the following dictionary:

```
grades = {
    "English": 97,
    "Math": 93,
    "Art": 74,
    "Music": 86
}
```

All three of the following statements would modify the value of the "Math" key and add a new "Gym" key:

```
grades.update({"Math": 97, "Gym": 93}) # argument is dict with keys
grades.update(Math=97, Gym=93) # individual arguments
grades.update([('Math', 97), ('Gym', 93)]) # argument is list of tuples
```

The update() Method

Note that when updating a single key in a dictionary, it is possible to use the `update()` method, but it is generally preferable to use the subscript syntax:

```
grades['Math'] = 97
```

setdefault()

The `setdefault(key, default)` method works like this:

- If key does not exist in the dictionary, key is added with a value of default.
- If key exists in the dictionary, the value for key is left unchanged.

The following example illustrates how `setdefault()` works:

Demo 31: iterables/Demos/setdefault.py

```
1.  grades = {
2.      "English": 97,
3.      "Math": 93,
4.      "Art": 74,
5.      "Music": 86
6.  }
7.
8.  grades.setdefault("Art", 87) # Art key exists. No change.
9.  print("Art grade:", grades["Art"])
10.
11. grades.setdefault("Gym", 97) # Gym key is new. Added and set.
12. print("Gym grade:", grades["Gym"])
```

Code Explanation

The preceding code will render the following:

```
Art grade: 74
Gym grade: 97
```

Notice that the value of the Art key did not change. `setdefault()` really means `add_key_unless_key_already_exists()`.

When to Use `setdefault()`

Imagine you are creating a dictionary of grades from data in a file or a database. You cannot be sure what data that source contains, but you need grades for four specific subjects. You can populate your dictionary using the external data, and then use `setdefault()` to make sure you have data for all keys:

```
grades = get_data() # Imaginary function that returns dictionary
grades.setdefault('English') = 0
grades.setdefault('Math') = 0
grades.setdefault('Art') = 0
grades.setdefault('Music') = 0
```

The grade for any one of the keys in the `setdefault()` calls will only be 0 if the dictionary returned by `get_data()` doesn't include that key.

❖ Dictionary View Objects

The following three methods return dictionary view objects:

- `mydict.keys()`
- `mydict.values()`
- `mydict.items()`

Try this out in Python interactive mode:

```
>>> grades = {
    "English": 97,
    "Math": 93,
    "Art": 75
}
>>> grades.keys()
dict_keys(['English', 'Math', 'Art'])
>>> grades.values()
dict_values([97, 93, 75])
>>> grades.items()
dict_items([('English', 97), ('Math', 93), ('Art', 75)])
```

As you can see, `dict_keys` and `dict_values` look like simple lists and `dict_items` looks like a list of tuples. But while they look like lists, they differ in two important ways:

1. Dictionary views do not support indexing or slicing as they have no set order.
2. Dictionary views cannot be modified. They provide dynamic views into a dictionary. When the dictionary changes, the views will change.

Consider the following:

Demo 32: iterables/Demos/dict_views.py

```
1.  grades = {
2.      "English": 97,
3.      "Math": 93,
4.      "Global Studies": 85,
5.      "Art": 74,
6.      "Music": 86
7.  }
8.
9.  gradepoints = grades.values()
10. print("Grade points:", gradepoints)
11.
12. grades["Art"] = 87
13. print("Grade points:", gradepoints)
```

Code Explanation

The output:

```
Grade points: dict_values([97, 93, 85, 74, 86])
Grade points: dict_values([97, 93, 85, 87, 86])
```

Notice that the Art grade changes in the output (from 74 to 87). There was no need to reassign `grade.values()` to `gradepoints` as `gradepoints` provides a dynamic view into the `grades` dictionary.

To get a list from a dictionary view, use the `list()` method:

```
>>> grades = {
    "English": 97,
    "Math": 93,
    "Art": 75
}
>>> list(grades.keys())
['English', 'Math', 'Art']
>>> list(grades.values())
[97, 93, 75]
>>> list(grades.items())
[('English', 97), ('Math', 93), ('Art', 75)]
```

❖ Deleting Dictionary Keys

The `del` statement can be used to delete a specific key from a dictionary, like this:

Demo 33: iterables/Demos/del_dict.py

```
1.  grades = {
2.      "English": 97,
3.      "Math": 93,
4.      "Global Studies": 85,
5.      "Art": 74,
6.      "Music": 86
7.  }
8.
9.  del grades["Math"] # deletes Math key
10. print(grades)
```

The len() Function

The len() function can be used to determine the number of characters in a string or the number of objects in a list, tuple, dictionary, or set:

```
>> len("hello")
5
>>> len( ["a", "b", "c"] )
3
>>> len( (255, 0, 255) )
3
>>> len({"Math": 97, "Music": 86, "Global Studies": 85})
3
```



Exercise 19: Creating a Dictionary from User Input

🕒 15 to 25 minutes

1. Open `iterables/Exercises/gradepoints.py` in your editor.
2. Write the `main()` function so that it:
 - A. Creates a `grades` dictionary and populates it with grades entered by the user in English, Math, Global Studies, Art, and Music.
 - B. Determines the average grade and prints it out. Note, to do this you will need to convert the user input to integers.

The program should run as follows:

```
English grade: 98
Math grade: 89
Global Studies grade: 79
Art grade: 91
Music grade: 84
Your average is 88.2
```

Challenge

After printing the average, ask the user to change the grade in one subject and then get the new average and print it out. **Hint:** you will have to prompt the user twice, once for the subject and once for the grade.

Solution: iterables/Solutions/gradepoints.py

```
1.  def main():
2.      grades = {}
3.      grades["English"] = int(input("English grade: "))
4.      grades["Math"] = int(input("Math grade: "))
5.      grades["Global Studies"] = int(input("Global Studies grade: "))
6.      grades["Art"] = int(input("Art grade: "))
7.      grades["Music"] = int(input("Music grade: "))
8.
9.      gradepoints = grades.values()
10.
11.     average = sum(gradepoints)/len(gradepoints)
12.
13.     print("Your average is", average)
14.
15.  main()
```

Challenge Solution: iterables/Solutions/gradepoints-challenge.py

```
1. def avg(gradepoints):
2.     average = sum(gradepoints)/len(gradepoints)
3.     return average
4.
5. def main():
6.     grades = {}
7.     grades["English"] = int(input("English grade: "))
8.     grades["Math"] = int(input("Math grade: "))
9.     grades["Global Studies"] = int(input("Global Studies grade: "))
10.    grades["Art"] = int(input("Art grade: "))
11.    grades["Music"] = int(input("Music grade: "))
12.
13.    gradepoints = grades.values()
14.
15.    average = avg(gradepoints)
16.
17.    print("Your average is", average)
18.
19.    subject = input("Choose a subject to change your grade: ")
20.    new_grade = input("What is your new " + subject + " grade? ")
21.    grades[subject] = int(new_grade)
22.    average = avg(gradepoints)
23.
24.    print("Your new average is", average)
25.
26.    main()
```

Code Explanation

Note that, as our program now requires calculating the average more than one time, we have moved that functionality into a new `avg()` function.

Sets

Sets are *mutable* unordered collections of distinct *immutable* objects. So, while the set itself can be modified, it cannot be populated with objects that can be modified. You can also think of sets as dictionaries in which the keys have no values. In fact, they can be created with curly braces, just like dictionaries:

```
>>> classes = {"English", "Math", "Global Studies",
               "Art", "Music"}
>>> type(classes)
<class 'set'>
```

Sets are less commonly used than the other iterables we've looked at in this lesson, but one great use for sets is to remove duplicates from a list as shown in the following example:

```
>>> veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"]
>>> v_set = set(veggies) # converts to set and remove duplicates
>>> v_set
{'pepper', 'tomato', 'spinach', 'pea'}
>>> veggies = list(v_set) # converts back to list
>>> veggies
['pepper', 'tomato', 'spinach', 'pea']
```

This is often done in a single step, like this:

```
>>> veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"]
>>> veggies = list(set(veggies)) # remove duplicates
```

You could also create a `remove_dups()` function:

```
def remove_dups(the_list):
    return list(set(the_list))

veggies = ["tomato", "spinach", "pepper", "pea", "tomato", "pea"]
veggies = remove_dups(veggies)
```

veggies will now contain:

```
['pepper', 'tomato', 'spinach', 'pea']
```

***args and **kwargs**

When defining a function, you can include two special parameters to accept an arbitrary number of arguments:

- ***args** – A parameter that begins with a single asterisk will accept an arbitrary number of non-keyworded arguments and store them in a tuple. Often the variable is named ***args**, but you can call it whatever you want (e.g., ***people** or ***colors**). Note that any parameters that come after ***args** in the function signature are keyword-only parameters.
- ****kwargs** – A parameter that begins with two asterisks will accept an arbitrary number of keyworded arguments and store them in a dictionary. Often the variable is named ****kwargs**, but, as with ***args**, you can call it whatever you want (e.g., ****people** or ****colors**). When included, ****kwargs** must be the last parameter in the function signature.

The parameters in a function definition must appear in the following order:

1. Non-keyword-only parameters that are required (i.e., have no defaults).
2. Non-keyword-only parameters that have defaults.
3. ***args**
4. Keyword-only parameters (with or without defaults).
5. ****kwargs**

❖ Using *args

Earlier, we saw a function that looked like this:

```
def add_nums(num1, num2, num3=0, num4=0, num5=0):  
    sum = num1 + num2 + num3 + num4 + num5  
    print(num1, "+", num2, "+", num3, "+",  
          num4, "+", num5, " = ", sum)
```

This works fine if you know that there will be between two and five numbers passed into `add_nums()`, but you can use ***args** to allow the function to accept an arbitrary number of numbers:

Demo 34: iterables/Demos/add_nums.py

```
1. def add_nums(num, *nums):
2.     total = sum(nums, num)
3.     print(f"The sum of {nums} and {num} is {total}.")
4.
5. def main():
6.     add_nums(1, 2)
7.     add_nums(1, 2, 3, 4, 5)
8.     add_nums(11, 12, 13, 14)
9.     add_nums(101, 201, 301)
10.
11. main()
```

Code Explanation

This will output:

```
The sum of (2,) and 1 is 3.
The sum of (2, 3, 4, 5) and 1 is 15.
The sum of (12, 13, 14) and 11 is 50.
The sum of (201, 301) and 101 is 603.
```

1. The `add_nums()` function requires one argument: `num`. It can also take zero or more subsequent arguments, which will all be stored in a single tuple, `nums`.
 2. We use the built-in `sum()` function (see page 131) to get the sum of `nums` and add it to `num`.
 3. The output is not perfect. We will improve it later on.
-

❖ Using ****kwargs**

The ****kwargs** parameter is most commonly used when you need to pass an unknown number of keyword arguments from one function to another. While this can be very useful (e.g., in decorators), it's beyond the scope of this lesson.

Conclusion

In this lesson, you have learned about lists, tuples, ranges, dictionaries, and sets. You also learned about the `*args` and `**kwargs` parameters. Soon, you'll learn to search these iterables for values and to loop through them performing operations on each element they contain one by one.

LESSON 6

Virtual Environments, Packages, and pip

Topics Covered

- ☑ Creating virtual environments.
- ☑ Activating and deactivating virtual environments.
- ☑ Installing packages with pip.
- ☑ Sharing project requirements so others can create a matching virtual environment.
- ☑ Deleting a virtual environment.

But, you see, the Land of Oz has never been civilized, for we are cut off from all the rest of the world. Therefore we still have witches and wizards amongst us.

– *The Wonderful Wizard of Oz*, L. Frank Baum

Introduction

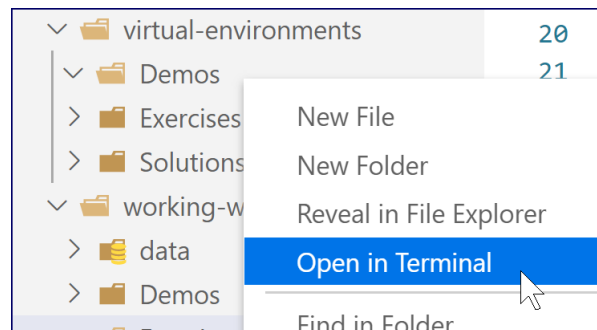
A virtual environment provides a self-contained directory tree with its own Python installation and additional packages necessary for the project(s) being done in that environment. As such, scripts can be run in a virtual environment that have dependencies that are different from those in other development projects that may be running in the standard environment or in separate virtual environments.

Exercise 20: Creating, Activating, Deactivating, and Deleting a Virtual Environment

 20 to 30 minutes

To create a virtual environment, you will use Python’s built-in `venv` module.

1. Open a terminal at `virtual-environments/Demos`:

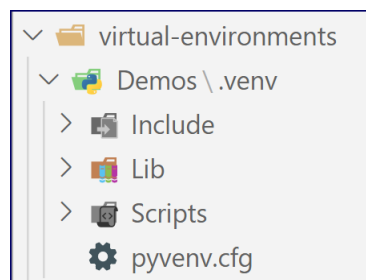


Then run the following command:

```
PS ..\virtual-environments\Demos> python -m venv .venv
```

This will create and populate a new `.venv` directory.²⁸

2. Take a look at the directory contents:



The contents will differ by operating system. Included in this directory is a `Scripts` (Windows) or `bin` (Mac) folder that contains the `python` executable file and scripts for activating the virtual environment.

28. You can name this directory whatever you want, but it is commonly called `.venv` (notice the prepended dot) to indicate that it is a special directory for holding a virtual environment.

3. To work within your virtual environment, you must first activate it. The command for activating a virtual environment varies by operating system. With `virtual-environments/Demos` still open at the terminal, run one of the following:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

4. The prompt text varies by operating system, terminal type, and settings. However, when the virtual environment is activated, its name will always appear enclosed in parentheses before the prompt. Here's what it looks like in PowerShell:

```
(.venv) PS ...\\virtual-environments\\Demos>
```

If you don't see the virtual environment name in parentheses before the prompt, you are not in the virtual environment.

5. You can now invoke the Python interpreter and/or install additional packages (using `pip`) within the virtual environment. Because the `$PATH` environment variable is modified when a virtual environment is activated, the executable files are resolved within the virtual environment. Let's take a look at the value of this environment variable. Run the following command for your environment:

Windows: PowerShell

```
$Env:PATH
```

Windows: Command Prompt

```
echo %PATH%
```

Mac / Linux

```
echo $PATH
```

You should see the path of the `Scripts` (Windows) or `bin` (Mac) directory prepended to the list of paths:

```
(.venv) PS ...\\virtual-environments\\Demos> $Env:PATH  
C:\\Webucator\\Python\\virtual-environments\\Demos\\.venv\\Scripts;C:...
```

Because it is at the beginning of \$PATH, the Scripts or bin directory will be scanned first to resolve references to executable files. Once you deactivate the virtual environment, the Scripts and bin directory will be removed from \$PATH.

6. To deactivate (exit) the virtual environment, run:

```
(.venv) PS ...\\virtual-environments\\Demos> deactivate
```

Now, you are back in the original prompt.

7. When you no longer need a virtual environment, you can delete it by using operating system commands to delete the folder (e.g., .venv) that was automatically built when you created the virtual environment. Note that all your work will be lost, so if you want to keep specific files, you should copy and paste them into another directory before deleting.

Packages with pip

A Python package is a collection of related modules in a folder. The **Python Package Index** (PyPI) is a collection of freely available Python packages, which any developer can install using pip. For example, the playsound package²⁹ can be installed like this:

```
pip install playsound
```

To uninstall a package, use `pip uninstall`:

```
pip uninstall playsound
```

❖ Packages and Virtual Environments

Often, it makes more sense to install packages within a virtual environment to keep your standard Python environment as clean as possible. If you create a new virtual environment for each project and install the necessary packages for each project within its virtual environment, then you know exactly what packages that project requires (in addition to the packages you have installed in the standard environment). You can see a list of the packages installed using:

29. <https://pypi.org/project/playsound/>


```
pip list
```

Run `pip list` at the prompt in your standard environment. You should see something like this (your packages may be different):

```
PS ~\virtual-environments\Demos\> pip list
```

Package	Version
astroid	2.3.3
colorama	0.4.3
isort	4.3.21
lxml	4.5.0
mccabe	0.6.1
pip	20.0.2
pylint	2.4.4
setuptools	45.2.0
six	1.13.0
wrapt	1.11.2

In the next exercise, you will learn how to write these requirements to a `requirements.txt` file and then use that file to set up a new virtual environment. That way, when you share project code with other developers, you can include the `requirements.txt` file so they can easily set up a virtual environment for the project on their computer.



Exercise 21: Working with a Virtual Environment

🕒 15 to 25 minutes

In this exercise you will:

1. Create and activate a virtual environment.
2. Install a package.
3. Create a `requirements.txt` file.
4. Deactivate and delete the virtual environment.
5. Recreate the virtual environment using the `requirements.txt` file.

❖ Instructions

1. Open `virtual-environments/Exercises` in the terminal.
2. Create a virtual environment named `.venv`:

```
(.venv) PS ...\\virtual-environments\\Exercises> python -m venv .venv
```

3. Activate the virtual environment:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

4. Run `pip freeze`. This will output a list of the packages installed in the virtual environment. At this point, it won't return anything as we haven't installed any packages yet.
5. Install the `playsound` package:

```
(.venv) PS ...\\virtual-environments\\Exercises> pip install playsound
```

6. Now, when you run `pip freeze`, you will see the `playsound` package and its version:

```
(.venv) PS ...\\virtual-environments\\Exercises> pip freeze  
playsound==1.2.2
```

`pip freeze` outputs the installed packages in the format `pip` needs to install them. Write the output to a file in the `Exercises` directory by running:

```
(.venv) PS ...\\virtual-environments\\Exercises> pip freeze > requirements.txt
```

7. Open the new `requirements.txt` file in Visual Studio Code. You will see that it contains just one line showing the `playsound` package and its version.
8. In the `Exercises` directory, we have included some sound files and a `demo.py` file that makes use of `playsound` to play the sounds. At the terminal, run:

```
(.venv) PS ...\\virtual-environments\\Exercises> python demo.py
```

You should hear several sounds played.

9. Deactivate the virtual environment by running:

```
(.venv) PS ...\\virtual-environments\\Exercises> deactivate
```

10. Now, run this again:

```
(.venv) PS ...\\virtual-environments\\Exercises> python demo.py
```

Because the `playsound` library is only installed in the virtual environment, you should get an error like this one:

```
ModuleNotFoundError: No module named 'playsound'
```

11. Delete the virtual environment by deleting the `.venv` folder at the command line, in VS Code's Explorer, or using the file system.
12. Now, recreate the virtual environment and activate it. Run:

```
(.venv) PS ...\\virtual-environments\\Exercises> python -m venv .venv
```

Then activate it:

Windows

```
.venv/Scripts/activate
```

Mac / Linux

```
source .venv/bin/activate
```

13. Now, run:

```
(.venv) PS ...\\virtual-environments\\Exercises> python demo.py
```

Because you haven't installed the requirements, you should get an error like this one:

```
ModuleNotFoundError: No module named 'playsound'
```

14. Now, install the requirements:

```
(.venv) PS ...\\virtual-environments\\Exercises> pip install -r requirements.txt
```

15. Run this again:

```
(.venv) PS ...\\virtual-environments\\Exercises> python demo.py
```

This time, it should run without errors.

Still Get an Error?

If you get an error that includes something like the following:

```
ModuleNotFoundError: No module named 'AppKit'
```

Try installing AppKit and PyObjC in the virtual environment:

```
(.venv) PS ...\\virtual-environments\\Exercises> pip install AppKit
```

```
(.venv) PS ...\\virtual-environments\\Exercises> pip install PyObjC
```

Then try running this again:

```
(.venv) PS ...\\virtual-environments\\Exercises> python demo.py
```

In this exercise, we have only installed one package, but you can imagine a project that has many packages. Creating and sharing the `requirements.txt` file makes it easy for you to let other developers quickly create an identical virtual environment.

Conclusion

In this lesson, you have learned to create and use virtual environments, and you have learned to install packages with `pip`.

LESSON 7

Flow Control

Topics Covered

- ✓ if conditions in Python.
- ✓ Loops in Python.
- ✓ Generator functions.
- ✓ List comprehensions.

But the path began nowhere and ended nowhere, and it remained mystery, as the man who made it and the reason he made it remained mystery.

– *The Call of the Wild*, Jack London

Introduction

By default, a program flows line by line in sequential order. We have seen already that we can change this flow by calling functions. The flow can also be changed using conditional statements and loops.

Conditional Statements

Conditional statements (`if-elif-else` conditions) allow programs to output different code based on specific conditions. The syntax is:

```
if some_conditions:
    do_this_1()
    do_this_2()
do_this_after()
```

Notice that the `do_this_1()` and `do_this_2()` function calls are both indented indicating that they are part of the `if` block. They will only run if `some_conditions` evaluates to `True`. The

`do_this_after()` function is not indented. That indicates that the `if` block has ended and it will run regardless of the value of `some_conditions`.

`elif` and `else`

- `elif` (for **else if**) conditions are only evaluated when the `if` condition is `False`. They are evaluated in order up until one evaluates to `True`, at which point, that `elif` block is executed and the rest of the `elif` conditions are skipped. An `if` statement can have zero or more `elif` conditions.
- The `else` block is only executed if the `if` condition and all the `elif` conditions evaluate to `False`. An `if` statement can have zero or one `else` conditions.

The following syntax blocks show an `if-else` and an `if-elif-else` statement:

```
if some_conditions:
    do_this_1()
    do_this_2()
else:
    do_this_3()
do_this_after()
```

```
if some_conditions:
    do_this_1()
    do_this_2()
elif other_conditions:
    do_this_3()
else:
    do_this_4()
    do_this_5()
do_this_after()
```

Values that Evaluate to False

The following values are considered `False`:

1. `None`
2. `0` (or `0.0`)
3. Empty containers such as strings, lists, tuples, etc.

You can use the `bool()` function to demonstrate this:

```
>>> bool(None)
False
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool('')
False
>>> bool([])
False
```

The following table shows Python's comparison operators:

Comparison Operators

Operator	Description
<code>==</code>	Equals
<code>!=</code>	Doesn't equal
<code>></code>	Is greater than
<code><</code>	Is less than
<code>>=</code>	Is greater than or equal to
<code><=</code>	Is less than or equal to
<code>is</code>	Is the same object
<code>is not</code>	Is not the same object

Python also includes these *membership* operators:

`in`

True if value is found in sequence.

```
'a' in ['a', 'b', 'c'] # True
'a' in 'abc' # True
```

not in

True if value is **not** found in sequence.

```
'a' not in ['a', 'b', 'c'] # False  
'd' not in 'abc' # True
```

The following example demonstrates an if-elif-else statement.

Demo 35: flow-control/Demos/if.py

```
1.  def main():  
2.      age = int(input('How old are you? '))  
3.  
4.      if age >= 21:  
5.          print('You can vote and drink.')  
6.      elif age >= 18:  
7.          print('You can vote, but can\'t drink.')  
8.      else:  
9.          print('You cannot vote or drink.')  
10.  
11.  main()
```

Code Explanation

You can see the different results by running this file and entering different ages at the prompt:

```
PS ..\flow-control\Demos> python if.py  
How old are you? 17  
You cannot vote or drink.  
PS ..\flow-control\Demos> python if.py  
How old are you? 19  
You can vote, but can't drink.  
PS ..\flow-control\Demos> python if.py  
How old are you? 21  
You can vote and drink.
```

Compound Conditions

More complex if statements often require that several conditions be checked. The following table shows and or operators for checking multiple conditions and the not operator for negating a boolean value (i.e., turning True to False or vice versa).

Logical Operators

Operator	Name	Example
and	AND	if a and b:
or	OR	if a or b:
not	NOT	if not a:

The following example shows these logical operators in practice:

Demo 36: flow-control/Demos/if2.py

```
1. def main():
2.     age = int(input('How old are you? '))
3.
4.     is_citizen = (input('Are you a citizen? Y or N ').lower() == 'y')
5.
6.     if age >= 21 and is_citizen:
7.         print('You can vote and drink.')
8.     elif age >= 21:
9.         print('You can drink, but can\'t vote.')
10.    elif age >= 18 and is_citizen:
11.        print('You can vote, but can\'t drink.')
12.    else:
13.        print('You cannot vote or drink.')
14.
15.    main()
```

Code Explanation

Notice that `is_citizen` is assigned to a comparison expression:

```
is_citizen = (input('Are you a citizen? Y or N ').lower() == 'y')
```

If the user enters “Y” or “y”, `is_citizen` will be True. Otherwise, it will be False.

The `is` and `is not` Operators

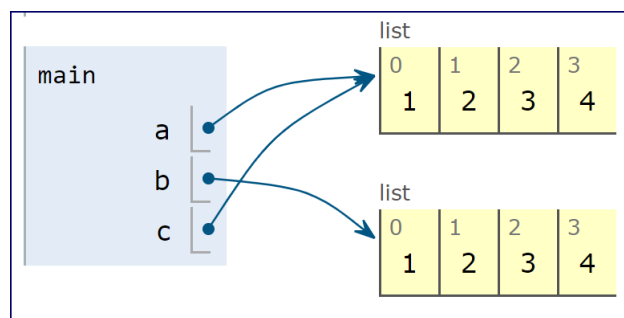
The `is` and `is not` operators differ from the `==` and `!=` operators. The former check whether two objects are the same object. The latter check whether two objects have the same value. The following code illustrates this:

```
>>> a = [1, 2, 3, 4]
>>> b = [1, 2, 3, 4]
>>> c = a
>>> a == b
True
>>> a is b
False
>>> a == c
True
>>> a is c
True
>>> id(a), id(b), id(c)
(1847278372096, 1847279744512, 1847278372096)
>>> a.append(5)
>>> c
[1, 2, 3, 4, 5]
```

Two identical lists are assigned to the `a` and `b` variables, but, because they are assigned separately, they are two separate objects. So, while `a == b` is `True`, `a is b` is `False`.

But `c` is assigned `a`, meaning `c` points to the same location in memory as `a` does, which is why the two variables have the same `id` and why, when we append 5 to `a` and then print `c`, we get the list containing 5.

The following diagram, which was generated at <http://www.pythontutor.com>, shows the status after all the variables have been assigned:



As you can see, `a` and `c` point to the same object:

`all()` and `any()`

The built-in `all()` and `any()` functions are used to loop through an iterable to check the boolean value of its elements.

- The `all()` function returns `True` if *all* the elements of the iterable evaluate to `True`.
- The `any()` function returns `True` if *any* of the elements of the iterable evaluate to `True`.

Ternary Operator

Many programming languages, including Python, have a ternary conditional operator, which is most often used for conditional variable assignment. To illustrate, consider this code:

```
if season == 'summer':
    pant_color = 'white'
else:
    pant_color = 'black'
```

That's very clear, but it takes four lines of code to assign a value to `pant_color`.

The syntax for the ternary operator in Python is:

```
[on_true] if [expression] else [on_false]
```

Here is how we could use this syntax to assign our pants' color:

```
pant_color = 'white' if season == 'summer' else 'black'
```

It's still pretty clear, but much shorter. Note that the expression could be any type of expression, including a function call, that returns a value that evaluates to `True` or `False`. For example, if you had an `is_summer()` function, you could do this:

```
pant_color = 'white' if is_summer() else 'black'
```

In Between

To check if a value is between two values, use the following syntax:

```
if low_value < your_value < high_value:
```

For example:

```
age = 15
if 12 < age < 20:
    print("You are a teenager.")

if 13 <= age <= 19:
    print("You are a teenager.")
```

This same syntax works with other types of objects (e.g., strings and dates) as well.

Loops in Python

As the name implies, loops are used to loop (or iterate) over code blocks. The following section shows examples of the two different types of loops in Python: **while** loops and **for** loops.

You can find the loop examples used in this section in `flow-control/Demos/loops.py`

❖ while Loops

while loops are used to execute a block of code repeatedly while one or more conditions evaluate to **True**.

```
num=0
while num < 6:
    print(num)
    num += 1
```

The preceding code will return:

0
1
2
3
4
5

You can combine while loops with user input to continually prompt the user until he or she enters an acceptable value. The following example illustrates this:

Demo 37: flow-control/Demos/while_input.py

```
1.  def is_valid_age(s):
2.      return s.isdigit() and 1 <= int(s) <= 113
3.
4.  def main():
5.      age = input('How old are you? ')
6.      while not is_valid_age(age):
7.          age = input('Please enter a real age as a number: ')
8.
9.      age = int(age)
10.     if age >= 21:
11.         print('You can vote and drink.')
12.     elif age >= 18:
13.         print('You can vote, but can\'t drink.')
14.     else:
15.         print('You cannot vote or drink.')
16.
17.  main()
```

Code Explanation

Run the file at the terminal and try entering some non-integer values:

```
PS ..\flow-control\Demos> python while_input.py
How old are you? twenty
Please enter a real age as a number: I'm 20
Please enter a real age as a number: 20!
Please enter a real age as a number: 20
You can vote, but can't drink.
```

The following demo shows how to combine `if` conditions and a `while` loop to create a simple game in which the user has to guess a number between 1 and 100. Review the code and the comments, which provide explanations.

Demo 38: flow-control/Demos/guess_the_number.py

```
1.  import random
2.
3.  def is_valid_num(s):
4.      # Make sure the number is a digit between 1 and 100.
5.      return s.isdigit() and 1 <= int(s) <= 100
6.
7.  def main():
8.      # Get a random number between 1 and 100
9.      number = random.randint(1, 100)
10.
11.     # Set guessed_number to False to start.
12.     # We'll set it to True when the user guesses the number
13.     guessed_number = False
14.
15.     # Get the user's guess
16.     guess = input("Guess a number between 1 and 100: ")
17.
18.     # Set num_guesses to 0.
19.     # We'll increment it by 1 after each guess
20.     num_guesses = 0
21.
22.     # We'll keep looping until the user guesses the number
23.     while not guessed_number:
24.         # Make sure the number is valid
25.         if not is_valid_num(guess):
26.             print("I won't count that one.")
27.             guess = input("A number between 1 and 100 please: ")
28.         else: # Number is valid
29.             num_guesses += 1 # Increment num_guesses
30.             guess = int(guess) # Convert user's guess to an int
31.
32.             # If guess is wrong, provide info and ask for another guess
33.             if guess < number:
34.                 guess = input("Too low. Guess again: ")
35.             elif guess > number:
36.                 guess = input("Too high. Guess again: ")
37.             else:
38.                 print("You got it in", num_guesses, "guesses!")
39.                 guessed_number = True # This will break us out of loop
40.
41.     print("Thanks for playing.")
42.
43.  main()
```

❖ for Loops

A for loop is used to loop through an iterator (e.g., a range, list, tuple, dictionary, etc.). The syntax is as follows:

```
for item in sequence_name:
    do_something(item)
```

Looping through a range

```
for num in range(6):
    print(num)

for num in range(0, 6):
    print(num)
```

Both loops above will return:

```
0
1
2
3
4
5
```

Remember that a range goes up to but does not include the *stop* value.

Skipping Steps

```
for num in range(1, 11, 2): # 3rd argument is the step
    print(num)
```

This loop will return:

```
1  
3  
5  
7  
9
```

Stepping Backwards

```
for num in range(10, 0, -1):  
    print(num)
```

This loop will return:

```
10  
9  
8  
7  
6  
5  
4  
3  
2  
1
```

Looping through a list and a tuple

```
nums = [0, 1, 2, 3, 4, 5]  
for num in nums:  
    print(num)
```

```
nums = (0, 1, 2, 3, 4, 5)  
for num in nums:  
    print(num)
```

Both loops above will return:

```
0  
1  
2  
3  
4  
5
```

Looping through a dict

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}  
  
for course in grades:  
    print(course)  
  
for course in grades.keys():  
    print(course)
```

Both of the loops above will return the courses (the keys):

```
English  
Math  
Art  
Music
```

Looping through the values of dict

```
for grade in grades.values():  
    print(grade)
```

This loop will return the grades (the values)

```
97  
93  
74  
86
```

Looping through the items of dict

```
for course, grade in grades.items():  
    print(f'{course}: {grade}')
```

This loop will return the courses (the keys) and grades (the values):

```
English: 97  
Math: 93  
Art: 74  
Music: 86
```

Notice the unpacking of the variables:

```
for course, grade in grades.items():
```

Remember that the `items()` method returns a `dict_items` object, which is a pseudo-list of tuples:

```
dict_items([('English', 97), ('Math', 93), ('Art', 74), ('Music', 86)])
```

The first item in each of the tuples above is the course and the second item is the grade.

Another way of writing the loop shown above would be:

```
for item in grades.items():  
    course = item[0]  
    grade = item[1]  
    print(f'{course}: {grade}')
```

But, the original way, which unpacks the tuples into the `course` and `grade` variables with each iteration, is more *pythonic*.



Exercise 22: All True and Any True

🕒 10 to 15 minutes

In this exercise, you will use loops to write `all_true()` and `any_true()` functions that behave the same way as the built-in `all()` and `any()` functions.

1. Open `flow-control/Exercises/all_and_any.py` in your editor.
2. In the `main()` function, there are calls to `all_true()` and `any_true()`, but those functions have yet to be written. Complete those functions:
 - `all_true()` – should return `True` if (and only if) all elements in the passed-in iterable evaluate to `True`.
 - `any_true()` – should return `True` if at least one element in the passed-in iterable evaluates to `True`.
3. Keep in mind that you want your function to return a value as soon as it can. For example, if you pass a list of 1,000,000 values to `all_true()` and the first value is `False`, the function does not need to continue looping through the list to determine that not all values are `True`.

Exercise Code: `flow-control/Exercises/all_and_any.py`

```
1. def all_true(iterable):
2.     # write function
3.     pass
4.
5. def any_true(iterable):
6.     # write function
7.     pass
8.
9. def main():
10.    a = all_true([1, 0, 1, 1, 1])
11.    b = all_true([1, 1, 1, 1, 1])
12.    c = any_true([0, 0, 0, 1, 1])
13.    d = any_true([0, 0, 0, 0, 0])
14.
15.    print(a, b, c, d) # Should be: False True True False
16.
17. main()
```



```
1.  def all_true(iterable):
2.      # Return False if any item is False
3.      for item in iterable:
4.          if not item:
5.              return False
6.      # If we made it through the loop without returning False,
7.      # then all items are True
8.      return True
9.
10. def any_true(iterable):
11.     # Return True if any item is True
12.     for item in iterable:
13.         if item:
14.             return True
15.     # If we made it through the loop without returning True,
16.     # then all items are False
17.     return False
18.
19. def main():
20.     a = all_true([1, 0, 1, 1, 1])
21.     b = all_true([1, 1, 1, 1, 1])
22.     c = any_true([0, 0, 0, 1, 1])
23.     d = any_true([0, 0, 0, 0, 0])
24.
25.     print(a, b, c, d) # Should be: False True True False
26.
27. main()
```

break and continue

To break out of a loop, insert a break statement:

```
for num in range(11, 20):
    print(num)
    if num % 5 == 0:
        break
```

Because $15 \% 5$ is equal to 0 (i.e., $15 / 5$ has a remainder of 0), the loop will stop after printing 15:


```
11  
12  
13  
14  
15
```

To jump to the next iteration of a loop without executing the remaining statements in the block, insert a `continue` statement:

```
for num in range(1, 12):  
    if num % 5 == 0:  
        continue  
    print(num)
```

This will skip to the next iteration when `num` is divisible by 5, resulting in:

```
1  
2  
3  
4  
6  
7  
8  
9  
11
```

Notice 5 and 10 are not printed.

Here's our Guess-the-Number game modified to use `continue` instead of an `else` block:

Demo 39: flow-control/Demos/guess_the_number_continue.py

```
1.  import random
2.
3.  def is_valid_num(s):
4.      return s.isdigit() and 1 <= int(s) <= 100
5.
6.  def main():
7.      number = random.randint(1, 100)
8.      guessed_number = False
9.      guess = input("Guess a number between 1 and 100: ")
10.     num_guesses = 0
11.
12.     while not guessed_number:
13.         if not is_valid_num(guess):
14.             print("I won't count that one.")
15.             guess = input("A number between 1 and 100 please: ")
16.             continue # Skip to next iteration through loop
17.
18.         num_guesses += 1
19.         guess = int(guess)
20.
21.         if guess < number:
22.             guess = input("Too low. Guess again: ")
23.         elif guess > number:
24.             guess = input("Too high. Guess again: ")
25.         else:
26.             print("You got it in", num_guesses, "guesses!")
27.             guessed_number = True
28.
29.     print("Thanks for playing.")
30.
31.  main()
```

Looping through Lines in a File

In an earlier lesson (see page 132), we showed how to use the `splitlines()` method of a string to split the lines of text read from a file into a list. Let's look again at the `states.txt` file we saw in that lesson:

Demo 40: flow-control/data/states.txt

```
1.  Alabama AL
2.  Alaska AK
3.  Arizona AZ
4.  Arkansas AR
5.  California CA
    -----Lines 6 through 50 Omitted-----
```

Code Explanation

Again, this file is a listing of states in the United States. Each line lists the name of the state and its abbreviation, separated by a tab.

Let's see how we can use the data in this file to write a program that allows the user to enter a state abbreviation and get the state name in return:

Demo 41: flow-control/Demos/get_state.py

```
1.  def get_state(abbreviation):
2.      # Read the data from the file into a list
3.      with open('../data/states.txt') as f:
4.          states = f.read().splitlines()
5.
6.      # Loop through the list
7.      for state_row in states:
8.          # Split each row on the tab character into a
9.          # two-element list
10.         state = state_row.split('\t')
11.
12.         # If the 2nd element matches the passed-in abbreviation
13.         # return the first element
14.         if state[1] == abbreviation.upper():
15.             return state[0]
16.
17.     # If no state matched the abbreviation, return None
18.     return None
19.
20. def main():
21.     # Loop until break statement
22.     while True:
23.         abbr = input('State abbreviation (q to quit): ').upper()
24.
25.         # Allow user to break loop by entering "Q"
26.         if abbr == 'Q':
27.             print('Goodbye!')
28.             break
29.
30.         # Get state name from abbreviation
31.         state = get_state(abbr)
32.
33.         # Inform the user what state, if any, maps to the abbreviation
34.         if state:
35.             print(f'"{abbr}" is the abbreviation for {state}.')
36.         else:
37.             print(f'No state has "{abbr}" as an abbreviation.')
38.
39.     main()
```

Code Explanation

Read through this file carefully, paying particular attention to the comments. Then run the file. Here is a sample result:

```
PS ..\flow-control\Demos> python get_state.py
State abbreviation (q to quit): ny
"NY" is the abbreviation for New York.
State abbreviation (q to quit): ca
"CA" is the abbreviation for California.
State abbreviation (q to quit): as
No state has "AS" as an abbreviation.
State abbreviation (q to quit): tx
"TX" is the abbreviation for Texas.
State abbreviation (q to quit): q
Goodbye!
```



Exercise 23: Word Guessing Game



45 to 120 minutes

In this exercise, you will create a word guessing game similar to hangman, but without a limit on the number of guesses.

You may find this exercise quite challenging. Just do your best and work your way through it one step at a time. Start by carefully reviewing the code. You may also find it helpful to play the game a couple of times by running `python guess_word.py` at the terminal from the `flow-control/Solutions` folder. A completed game would look like this:

```
PS ..\flow-control\Solutions> python guess_word.py
The word contains 9 letters.
Guess a letter or a 9-letter word: A
Sorry. The word has no "A"s.
*****
Guess a letter or a 9-letter word: O
The word has 3 "O"s.
*OO***O**
Guess a letter or a 9-letter word: S
The word has one "S".
*OO*S*O**
Guess a letter or a 9-letter word: A
You already guessed "A".
Guess a letter or a 9-letter word: CK
Invalid entry.
Guess a letter or a 9-letter word: Woodstock
WOODSTOCK is it! It took 4 tries.
```

The starting code is shown below:

Exercise Code: flow-control/Exercises/guess_word.py

```
1. import random
2.
3. def get_word():
4.     """Returns random word."""
5.     words = ['Charlie', 'Woodstock', 'Snoopy', 'Lucy', 'Linus',
6.             'Schroeder', 'Patty', 'Sally', 'Marcie']
7.     return random.choice(words).upper()
8.
9. def check(word, guesses):
10.    """Creates and returns string representation of word
11.    displaying asterisks for letters not yet guessed."""
12.    status = '' # Current status of guess
13.    last_guess = guesses[-1]
14.    matches = 0 # Number of occurrences of last_guess in word
15.
16.    # Loop through word checking if each letter is in guesses
17.    # If it is, append the letter to status
18.    # If it is not, append an asterisk (*) to status
19.    # Also, each time a letter in word matches the last guess,
20.    # increment matches by 1.
21.
22.    # Write a condition that outputs one of the following when
23.    # the user's last guess was "A":
24.    # 'The word has 2 "A"s.' (where 2 is the number of matches)
25.    # 'The word has one "A".'
26.    # 'Sorry. The word has no "A"s.'
27.
28.    return status
29.
30. def main():
31.     word = get_word() # The random word
32.     n = len(word) # The number of letters in the random word
33.     guesses = [] # The list of guesses made so far
34.     guessed = False
35.     print('The word contains {} letters.'.format(n))
36.
37.     while not guessed:
38.         guess = input('Guess a letter or a {}-letter word: '.format(n))
39.         guess = guess.upper()
40.         # Write an if condition to complete this loop.
41.         # You must set guessed to True if the word is guessed.
42.         # Things to be looking for:
43.         # - Did the user already guess this guess?
44.         # - Is the user guessing the whole word?
```

```

45.         # - If so, is it correct?
46.         # - Is the user guessing a single letter?
47.         # - If so, you'll need your check() function.
48.         # - Is the user's guess invalid (the wrong length)?
49.         #
50.         # Also, don't forget to keep track of the valid guesses.
51.
52.     print('{} is it! It took {} tries.'.format(word, len(guesses)))
53.
54. main()

```

1. Open `flow-control/Exercises/guess_word.py` in your editor. The program consists of three functions:

- A. `get_word()` – returns a random secret word to guess. This function is complete.
- B. `check()` – described below. You must complete this function.
- C. `main()` – our main program. You must complete this function.

2. The program selects a secret word and prints: “The word contains *n* letters.” For example:

```
The word contains 6 letters.
```

3. The program then continuously prompts the user to choose a letter or guess the word until the word is guessed: “Guess a letter or a *n*-letter word: ”. For example:

```
Guess a letter or a 6-letter word:
```

4. The program keeps all previous guesses in a `guesses` list.

5. For each guess:

- A. **If the user has already guessed that letter or word**, the program prints “You already guessed “*guess*”.” For example:

```
You already guessed "A".
```

It then prompts for another guess.

- B. **Otherwise, if the user enters a word of the same length as the secret word**, the program records the guess in `guesses` and then checks to see if it is correct.

- i. If the user's guess is correct, the program prints "*GUESS* is it! It took *n* tries." For example:

```
SNOOPY is it! It took 8 tries.
```

The game then ends.

- ii. If the user's guess is incorrect, the program prints "Sorry, that is incorrect." It then prompts for another guess.

- C. **Otherwise, if the user enters a single character**, the program records the guess in `guesses` and then checks to see if the letter is in the word. It should do this by calling the `check()` function and passing it the secret word and `guesses`:

```
result = check(word, guesses)
```

- i. The `check()` function must do the following:
 - a. Print one of the following based on the number of times the last guess in `guesses` shows up in the secret word:
 - a. **Multiple times:** "The word has *n* "*guess*"s." For example:

```
The word has 2 "O"s.
```
 - b. **Exactly one time:** "The word has one "*guess*"." For example:

```
The word has one "S".
```
 - c. **Zero times:** "Sorry. The word has no "*guess*"s." For example:

```
Sorry. The word has no "E"s.
```
 - b. It should then return a string representation of the word displaying asterisks for letters not yet guessed. For example, if the word is SNOOPY and O and Y have been guessed, it would return `**OO*Y`.
- ii. Back in the `main()` function, the value returned from the `check()` function should be compared to the secret word:

- a. **If the two values are the same**, the program prints “*GUESS* is it! It took *n* tries.” For example:

```
SNOOPY is it! It took 8 tries.
```

The game then ends.

- b. **If the two values are different**, the program prints the value returned from `check()` (e.g., `**00*Y`).

- D. **Otherwise**, the program prints “Invalid Entry” and prompts the user for another guess. We only reach this block if the user enters a multi-character string that is not the same length as the secret word.

Challenge

Create your own list of words in a separate file in the `flow-control/data` folder (or use the Harry Potter spells in the `words.txt` file that’s already there) and use the words from that file in the `get_word()` function.


```
-----Lines 1 through 8 Omitted-----
9.  def check(word, guesses):
10.     """Creates and returns string representation of word
11.     displaying asterisks for letters not yet guessed."""
12.     status = '' # Current status of guess
13.     last_guess = guesses[-1]
14.     matches = 0 # Number of occurrences of last_guess in word
15.
16.     for letter in word:
17.         status += letter if letter in guesses else '*'
18.
19.         if letter == last_guess:
20.             matches += 1
21.
22.     if matches > 1:
23.         print('The word has {} "{}"s.'.format(matches, last_guess))
24.     elif matches == 1:
25.         print('The word has one "{}".'.format(last_guess))
26.     else:
27.         print('Sorry. The word has no "{}"s.'.format(last_guess))
28.
29.     return status
30.
31.  def main():
32.     word = get_word() # The random word
33.     n = len(word) # The number of letters in the random word
34.     guesses = [] # The list of guesses made so far
35.     guessed = False
36.     print('The word contains {} letters.'.format(n))
37.
38.     while not guessed:
39.         guess = input('Guess a letter or a {}-letter word: '.format(n))
40.         guess = guess.upper()
41.         if guess in guesses:
42.             print('You already guessed "{}".'.format(guess))
43.         elif len(guess) == n: # Guessing whole word
44.             guesses.append(guess)
45.             if guess == word:
46.                 guessed = True
47.             else:
48.                 print('Sorry, that is incorrect.')
49.         elif len(guess) == 1: # Guessing letter
50.             guesses.append(guess)
51.             result = check(word, guesses)
```

```
52.         if result == word:
53.             guessed = True
54.         else:
55.             print(result)
56.     else: # guess had wrong number of characters
57.         print('Invalid entry.')
58.
59.     print('{} is it! It took {} tries.'.format(word, len(guesses)))
60.
61. main()
```

Challenge Solution: flow-control/Solutions/guess_word_challenge.py

```
1.  import random
2.
3.  def get_word():
4.      """Returns random word."""
5.      with open('../data/words.txt') as f:
6.          words = f.read().splitlines()
7.          return random.choice(words).upper()
      -----Lines 8 through 61 Omitted-----
```

❖ The else Clause of Loops

In Python, `for` and `while` loops have an optional `else` clause, which is executed after the loop has successfully completed iterating (i.e., without a `break`). The following demo shows how it works:

Demo 42: flow-control/Demos/loop_else.py

```
1. def main():
2.     print('Example 1: for loop')
3.     num = int(input('Enter a number: '))
4.     for i in range(5):
5.         if i == num:
6.             break
7.         print(i)
8.     else:
9.         print(f'Completed iterating without reaching {num}.')
10.
11.    print('\nExample 2: while loop')
12.    i = 0
13.    num = int(input('Enter a number: '))
14.    while i <= 5:
15.        if i == num:
16.            break
17.        print(i)
18.        i += 1
19.    else:
20.        print(f'Completed iterating without reaching {num}.')
21.
22.    main()
```

Code Explanation

The preceding code will render something like the following:

First Run

```
PS ...\\flow-control\\Demos> python loop_else.py
```

```
Example 1: for loop
```

```
Enter a number: 4
```

```
0
```

```
1
```

```
2
```

```
3
```

```
Example 2: while loop
```

```
Enter a number: 3
```

```
0
```

```
1
```

```
2
```

Second Run

```
PS ...\\flow-control\\Demos> python loop_else.py
```

Example 1: for loop

Enter a number: 7

0

1

2

3

4

Completed iterating without reaching 7.

Example 2: while loop

Enter a number: 6

0

1

2

3

4

5

Completed iterating without reaching 6.

The first run through, we entered numbers lower than 5, so the loops were broken out of before completing. The second run through, we entered numbers higher than 5, so the loops completed all iterations, which led to the `else` clause also being executed.

So, when would you use this? You might use it to check user-entered text for black-listed words. The following code shows a common way to do this in other programming languages:

```
sentence = input('Input a sentence: ')
found_bad_word = False
for word in sentence.split():
    if word in BLACK_LIST:
        found_bad_word = True
        break

if found_bad_word:
    print('You used a naughty word.')
else:
    print('Sentence passes cleanliness test.')
```

The code in the following file shows how you can accomplish the same thing in a cleaner way using the else clause:

Demo 43: flow-control/Demos/loop_else_use_case.py

```
1.  BLACK_LIST = [  
2.      'shitake',  
3.      'sugar',  
4.      'fudge',  
5.      'gosh',  
6.      'darn',  
7.      'dang',  
8.      'heck'  
9.  ]  
10.  
11. def main():  
12.     sentence = input('Input a sentence: ')  
13.  
14.     for word in sentence.split():  
15.         if word in BLACK_LIST:  
16.             print('You used a naughty word.')  
17.             break  
18.     else:  
19.         print('Sentence passes cleanliness test.')  
20.  
21.     main()
```

Code Explanation

Here are the results of running this file, first with a “clean” sentence and then with a “dirty” one:

```
PS ..\flow-control\Demos> python loop_else_use_case.py  
Input a sentence: I like cream in my coffee.  
Sentence passes cleanliness test.  
PS ..\flow-control\Demos> python loop_else_use_case.py  
Input a sentence: I like sugar in my coffee.  
You used a naughty word.
```

Exercise 24: for...else

 10 to 20 minutes

In this exercise, you will rewrite a function to use the `else` clause of a loop.

1. Open `flow-control/Exercises/states.py` in your editor and review the code. Then run the script to see how it works.

A. First, try entering only valid states:

```
PS ~\flow-control\Exercises> python states.py
Name as many state abbreviations do you know?
Separate them with spaces:
AZ CA NY
You named 3 states.
```

B. Now, try including an invalid state:

```
PS ~\flow-control\Exercises> python states.py
Name as many state abbreviations do you know?
Separate them with spaces:
AZ CA CC NY
CC is not a state.
```

2. Modify the `main()` function so that it uses the `for` loop's `else` clause.
3. Run the script again. It should work in exactly the same way.

Exercise Code: flow-control/Exercises/states.py

```
1.  def is_state(state):
2.      with open('../data/states.txt') as f:
3.          states = f.read().splitlines()
4.
5.          state_abbreviations = []
6.          for state_row in states:
7.              state_abbreviation = state_row.split('\t')[1]
8.              state_abbreviations.append(state_abbreviation)
9.
10.         return state in state_abbreviations
11.
12.  def main():
13.      print('Name as many state abbreviations do you know?')
14.      print('Separate them with spaces:')
15.      states = input('').split()
16.      bad_state = False
17.      for state in states:
18.          state = state.upper()
19.          if not is_state(state):
20.              print(f'{state} is not a state.')
21.              bad_state = True
22.              break
23.
24.      if not bad_state:
25.          print(f'You named {len(states)} states.')
26.
27.  main()
```

Solution: flow-control/Solutions/states.py

```
-----Lines 1 through 11 Omitted-----
12. def main():
13.     print('Name as many state abbreviations do you know?')
14.     print('Separate them with spaces:')
15.     states = input('').split()
16.     for state in states:
17.         state = state.upper()
18.         if not is_state(state):
19.             print(f'{state} is not a state.')
20.             break
21.     else:
22.         print(f'You named {len(states)} states.')
23.
24. main()
```

The enumerate() Function

It is common in other programming languages to write code like this:

Demo 44: flow-control/Demos/without_enum.py

```
1. i = 1
2. for item in ['a', 'b', 'c']:
3.     print(i, item, sep='. ')
4.     i += 1
```

Code Explanation

This code will output an enumerated list:

```
1. a
2. b
3. c
```

The more Pythonic way of accomplishing the same thing is to use the `enumerate()` function, like this:

Demo 45: flow-control/Demos/with_enum.py

```
1. for i, item in enumerate(['a', 'b', 'c'], 1):
2.     print(i, item, sep='. ')
```

This saves us from creating a new variable to hold the count.

Note that `enumerate()` takes two arguments:

1. The iterable to enumerate.
2. The number at which to start counting (defaults to 0).

It returns an iterable of two-element tuples of the format `(count, value)`.

Here is a more practical example, in which we output a list of all the state names in `states.txt`:

Demo 46: flow-control/Demos/state_list.py

```
1. def main():
2.     with open('../data/states.txt') as f:
3.         states = f.read().splitlines()
4.
5.         for i, state in enumerate(states, 1):
6.             state_name = state.split('\t')[0]
7.             print(f'{i}. {state_name}')
8.
9.     main()
```

Generators

Generators are special iterators that can only be iterated through one time. Generators are created with special generator functions. Before looking at one, let's first consider how a standard iterator (e.g., a list) works:

Demo 47: flow-control/Demos/list_loop.py

```
1.  import random
2.
3.  def get_rand_nums(low, high, num):
4.      numbers = []
5.      for number in range(num):
6.          numbers.append(random.randint(low, high))
7.      return numbers
8.
9.  numbers = get_rand_nums(1, 100, 5)
10.
11. print('First time through:')
12. for num in numbers:
13.     print(num)
14.
15. print('Second time through:')
16. for num in numbers:
17.     print(num)
```

Code Explanation

This code will output something like:

```
First time through:
13
63
50
10
16
Second time through:
13
63
50
10
16
```

There is nothing new in this code. The `get_rand_nums()` function returns a list of `num` random numbers between `low` and `high`. We assign that list to `numbers` and then loop through it twice. We can loop through it any number of times.

Now, consider this code, which uses a generator:

Demo 48: flow-control/Demos/generator_loop.py

```
1. import random
2.
3. def get_rand_nums(low, high, num):
4.     for number in range(num):
5.         yield random.randint(low, high)
6.
7. numbers = get_rand_nums(1, 100, 5)
8.
9. print('First time through:')
10. for num in numbers:
11.     print(num)
12.
13. print('Second time through:')
14. for num in numbers:
15.     print(num)
```

Code Explanation

This code will output something like:

```
First time through:
69
32
65
76
87
Second time through:
```

Here are the two functions again:

List Loop

```
def get_rand_nums(low, high, num):
    numbers = []
    for number in range(num):
        numbers.append(random.randint(low, high))
    return numbers
```

Generator

```
def get_rand_nums(low, high, num):  
    for number in range(num):  
        yield random.randint(low, high)
```

The two functions work in essentially the same way, but the second function is much cleaner as there is no need for the local `numbers` variable. Rather than creating a full list and then returning it, the generator function yields each result one by one as it works its way through the `for` loop. The only functional difference, as the example illustrates, is that you can only iterate through the generator one time. However, if you want to iterate through multiple times, you can simply call the generator function again as shown in the next example:

Demo 49: flow-control/Demos/generator_loop2.py

```
1.  import random  
2.  
3.  def get_rand_nums(low, high, num):  
4.      for number in range(num):  
5.          yield random.randint(low, high)  
6.  
7.  print('First time through:')  
8.  for num in get_rand_nums(1, 100, 5):  
9.      print(num)  
10.  
11. print('Second time through:')  
12. for num in get_rand_nums(1, 100, 5):  
13.     print(num)
```

Code Explanation

This will return something like:


```
First time through:
```

```
56
```

```
19
```

```
99
```

```
33
```

```
8
```

```
Second time through:
```

```
36
```

```
88
```

```
76
```

```
54
```

```
80
```

As the function is returning a sequence of random numbers, you will get different numbers each time you call the generator function.

If you want a sequence of random numbers that you can count on being the same each time you iterate through it, then you should create a function that returns a list.

❖ When to Use Generators

Use generators whenever you want to be able to get the next item in a sequence without creating the whole sequence ahead of time. Some examples:

Infinite Sequences

Demo 50: flow-control/Demos/odd_numbers.py

```
1.  def odd_numbers():
2.      i = -1
3.      while True:
4.          i += 2
5.          yield i
6.
7.  def main():
8.      for i in odd_numbers():
9.          print(i)
10.         if input('Enter for next or q to quit: ') == 'q':
11.             print('Goodbye!')
12.             break
13.
14.  main()
```

Code Explanation

It's impossible to store the infinite number of odd numbers in a list. This generator function just yields the next one every time it is called:

```
PS ..\flow-control\Demos> python odd_numbers.py
```

```
1
Enter for next or q to quit:
3
Enter for next or q to quit:
5
Enter for next or q to quit: q
Goodbye!
```

The Fibonacci Sequence

The Fibonacci sequence is a sequence in mathematics in which each number is the sum of the two preceding numbers.

Demo 51: flow-control/Demos/fibonacci.py

```
1.  def fibonacci():
2.      a, b = 0, 1
3.      while True:
4.          yield a
5.          a, b = b, a + b
6.
7.  def main():
8.      for i in fibonacci():
9.          print(i)
10.         if input('Enter for next or q to quit: ') == 'q':
11.             print('Goodbye!')
12.             break
13.
14.  main()
```

Code Explanation

See https://en.wikipedia.org/wiki/Fibonacci_number for information on the Fibonacci Sequence.

A Sequence of Unknown Length

In Bingo, the caller keeps picking out numbers until someone wins. It's impossible to know ahead of time how many numbers must be drawn.

Demo 52: flow-control/Demos/bingo.py

```
1.  import random
2.
3.  def bingo():
4.      balls = {
5.          'B': list(range(1, 16)),
6.          'I': list(range(16, 31)),
7.          'N': list(range(31, 46)),
8.          'G': list(range(46, 61)),
9.          'O': list(range(61, 76))
10.     }
11.
12.     while balls:
13.         letter = random.choice(list(balls.keys()))
14.         number = random.choice(balls[letter])
15.         balls[letter].remove(number) # Remove number from letter list
16.         if not balls[letter]: # Letter has no more numbers
17.             print('Removing', letter)
18.             del balls[letter] # Delete letter from dictionary
19.         yield (letter, number)
20.
21. def main():
22.     for ball in bingo():
23.         print(ball)
24.         if input('Enter for next ball or q to quit') == 'q':
25.             print('Goodbye!')
26.             break
27.     else:
28.         print('All out of balls.')
29.
30. main()
```

Code Explanation

This generator function keeps spitting out Bingo balls until it runs out of balls.

Here's a Bingo card, in case you want to try your luck:

B I N G O				
14	26	37	51	66
15	27	42	57	74
8	28	*	47	64
4	19	39	59	72
3	17	45	50	71

❖ The next() Function

To be an iterator, an object must have a special `__next__()` method (that's two underscores before and two underscores after "next"), which returns the next item of the iterator. Python's built-in `next(iter)` function calls the `__next__()` method of `iter`. The following example shows how to use the `next()` function with a generator to play Rock, Paper, Scissors (RoShamBo) against your computer.

Demo 53: flow-control/Demos/roshambo.py

```
1.  import random
2.
3.  def roshambo(weapons):
4.      while True:
5.          yield random.choice(weapons)
6.
7.  def play(weapons, choice, python_weapons):
8.      your_weapon = weapons[int(choice)-1]
9.      python_weapon = next(python_weapons)
10.
11.     if your_weapon == python_weapon:
12.         print('Tie: You both chose', your_weapon)
13.     elif ((your_weapon == 'Scissors' and python_weapon == 'Paper')
14.           or (your_weapon == 'Paper' and python_weapon == 'Rock')
15.           or (your_weapon == 'Rock' and python_weapon == 'Scissors')):
16.         print('You win:', your_weapon, 'beats', python_weapon)
17.     else:
18.         print('You lose:', python_weapon, 'beats', your_weapon)
19.     print('-----')
20.
21.
22.  def make_choice():
23.      choice = input("""Choose your weapon:
24.  1: Rock
25.  2: Paper
26.  3: Scissors
27.  q: Quit
28.  """)
29.      return choice
30.
31.  def main():
32.      weapons = ['Rock', 'Paper', 'Scissors']
33.      python_weapons = roshambo(weapons)
34.      choice = make_choice()
35.      while choice in ['1', '2', '3']:
36.          play(weapons, choice, python_weapons)
37.          choice = make_choice()
38.      print('Goodbye!')
39.
40.  main()
```

Code Explanation

One benefit of assigning a generator to `python_weapons` instead of a list is that you don't need to know how many iterations there will be. The generator just spits out a new result each time its `__next__()` method is called.

Here is a possible output of the game:

```
PS ..\flow-control\Demos> python roshambo.py
Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
1
You win: Rock beats Scissors
-----
Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
2
You lose: Scissors beats Paper
-----
Choose your weapon:
1: Rock
2: Paper
3: Scissors
q: Quit
q
Goodbye!
```

Note that the `roshambo()` generator is overly simplistic. On line 9, we could have just used `random.choice(weapons)` to get the value of `python_weapon`. But imagine that the generator did something more than that. For example, you could give it weapon preferences like this:

```
def roshambo(weapons):
    while True:
        num = random.random()
        if num < .5:
            yield weapons[0]
        elif num < .8:
            yield weapons[1]
        else:
            yield weapons[2]
```

List Comprehensions

List comprehensions are used to create new lists from existing sequences by taking a subset of that sequence and/or modifying its members. To understand the purpose, let's first look at creating a new list from a list using a for loop:

```
squares = []
for i in range(10):
    squares.append(i*i)

print(squares) # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

This same thing can be done much more tersely with a list comprehension. The syntax is:

```
new_list = [modify(item) for item in items]
```

Where `modify(item)` represents any change made to `item`.

Using a list comprehension, the code above could be rewritten like this:

```
squares = [i*i for i in range(10)]
```

Here's another example, in which we use a list comprehension to create a list of people's initials from a list of their names:

Demo 54: flow-control/Demos/list_from_list.py

```
1. def initials(name):
2.     fullname = name.split(' ')
3.     inits = (fullname[0][0], fullname[1][0])
4.     return inits
5.
6. def main():
7.     names = ['Graham Chapman', 'John Cleese', 'Eric Idle',
8.             'Terry Gilliam', 'Terry Jones', 'Michael Palin']
9.     inits = [initials(name) for name in names]
10.    for i in inits:
11.        print(f'{i[0]}.{i[1]}.')
12.
13.    main()
```

Code Explanation

The list comprehensions creates a new list from names by passing each of its elements to the `initials()` function. This will return:

```
G.C.
J.C.
E.I.
T.G.
T.J.
M.P.
```

Remember our `add_nums()` function from the **Iterables** lesson:

Demo 55: iterables/Demos/add_nums.py

```
1. def add_nums(num, *nums):
2.     total = sum(nums, num)
3.     print(f"The sum of {nums} and {num} is {total}.")
4.
5. def main():
6.     add_nums(1, 2)
7.     add_nums(1, 2, 3, 4, 5)
8.     add_nums(11, 12, 13, 14)
9.     add_nums(101, 201, 301)
10.
11. main()
```

When you call that function like this:

```
add_nums(1, 2, 3, 4, 5)
```

The result is:

```
The sum of (2, 3, 4, 5) and 1 is 15.
```

That's not too pretty. We can improve it using a list comprehension, like this:

Demo 56: flow-control/Demos/add_nums.py

```
1. def add_nums(num, *nums):
2.     total = sum(nums, num)
3.
4.     nums_joined = ', '.join([str(n) for n in nums])
5.     print(f"The sum of {nums_joined} and {num} is {total}.")
6.
7. def main():
8.     add_nums(1, 2)
9.     add_nums(1, 2, 3, 4, 5)
10.    add_nums(11, 12, 13, 14)
11.    add_nums(101, 201, 301)
12.
13. main()
```

Code Explanation

This uses a list comprehension to convert the list of numbers into a list of strings and then joins that list on ', ' in one step. The output will be:

```
The sum of 2 and 1 is 3.  
The sum of 2, 3, 4, 5 and 1 is 15.  
The sum of 12, 13, 14 and 11 is 50.  
The sum of 201, 301 and 101 is 603.
```

Much prettier, right?

❖ Creating Sublists with List Comprehensions

List comprehensions have an optional `if` clause that can be used to create a sublist from a list:

```
new_list = [item for item in items if some_condition]
```

Assume that you want to get all the three-letter words from a list of words. First, let's do it without a list comprehension:

```
three_letter_words = []  
for word in words:  
    if len(word) == 3:  
        three_letter_words.append(word)
```

In the following file, we do the same thing with a list comprehension:

Demo 57: flow-control/Demos/sublist_from_list.py

```
1. def main():  
2.     words = ['Woodstock', 'Gary', 'Tucker', 'Gopher', 'Spike', 'Ed',  
3.             'Faline', 'Willy', 'Rex', 'Rhino', 'Roo', 'Littlefoot',  
4.             'Bagheera', 'Remy', 'Pongo', 'Kaa', 'Rudolph', 'Banzai',  
5.             'Courage', 'Nemo', 'Nala', 'Alvin', 'Sebastian', 'Iago']  
6.     three_letter_words = [w for w in words if len(w) == 3]  
7.     print(three_letter_words)  
8.  
9.     main()
```

Code Explanation

This code combines an `if` condition within a `for` loop into a single line. It will return:

```
['Rex', 'Roo', 'Kaa']
```

Here's another example. Consider the following file:

Demo 58: flow-control/Demos/states_with_spaces.py

```
1.  def main():
2.      with open('../data/states.txt') as f:
3.          lines = f.read().splitlines()
4.
5.          states_with_spaces = []
6.
7.          for line in lines:
8.              # Split the line into a 2-item list containing the
9.              #   state name and abbreviation
10.             state = line.split('\t')
11.
12.             # If the state name has a space, add it to states_with_spaces
13.             if ' ' in state[0]:
14.                 states_with_spaces.append(state[0])
15.
16.             # Print the states_with_spaces list
17.             for i, state_name in enumerate(states_with_spaces, 1):
18.                 print(f'{i}. {state_name}')
19.
20.  main()
```

Code Explanation

This reads in the content of the `states.txt` file and then splits it into a list of lines. It then loops through that list to create a new list of just the states that have spaces in their names. It will output:

1. New Hampshire
2. New Jersey
3. New Mexico
4. New York
5. North Carolina
6. North Dakota
7. Rhode Island
8. South Carolina
9. South Dakota
10. West Virginia

You could replace the for loop with a couple of list comprehensions, like this:

Demo 59: flow-control/Demos/states_with_spaces2.py

```
-----Lines 1 through 4 Omitted-----
5.     states = [line.split('\t')[0] for line in lines]
6.     states_with_spaces = [state for state in states if ' ' in state]
-----Lines 7 through 12 Omitted-----
```

Code Explanation

1. The first list comprehension gets all the state names from the lines in the file.
2. The second list comprehension then uses that list to get just the state names that have spaces in them.

This could also be done in a single step like this:

Demo 60: flow-control/Demos/states_with_spaces3.py

```
-----Lines 1 through 4 Omitted-----
5.     states_with_spaces = [
6.         line.split('\t')[0] # Get state
7.         for line in lines # For each line
8.         if ' ' in line.split('\t')[0] # Where there is a space in state
9.     ]
-----Lines 10 through 15 Omitted-----
```

Code Explanation

While that's a bit harder to read, the code is more efficient, because it doesn't have to loop through lines and then loop again through states. It just does all the work the first time through the loop.

Conclusion

In this lesson, you have learned to write `if-elif-else` conditions and to loop through sequences. You also learned about the `enumerate()` function, generators, and list comprehensions.

LESSON 8

Exception Handling

Topics Covered

- ☑ Handling exceptions in Python.

Once married, it would be infinitely easier to ask her father's forgiveness, than to beg his permission beforehand.

– *A Professional Rider, Mrs. Edward Kennard*

Introduction

By this time, you've probably run into some exceptions (errors) in your Python scripts. In this lesson, we will learn how to anticipate and handle exceptions gracefully.

Exception Basics

You may remember from school that you cannot divide by zero. Two or three people can share a pizza. One person can eat a whole pizza alone. But zero people cannot eat a pizza. The pizza will never get eaten. In Python, if you try to divide by zero, you will get a `ZeroDivisionError` exception:

```
>>> 1/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

In some cases, you may be fine with allowing the Python interpreter to report these exceptions as it finds them. But in other cases, you will want to anticipate and catch these exceptions and handle them in some special way. You do this in Python with `try/except` blocks. Here's a very simple example:

Demo 61: exception-handling/Demos/simple.py

```
1.  try:
2.      1/0
3.  except:
4.      print('You cannot divide by zero!')
```

Multiple except Clauses

You can have multiple except clauses with each clause specifying the type of exception to catch:

Demo 62: exception-handling/Demos/specific.py

```
1.  def divide():
2.      try:
3.          numerator = int(input('Enter a numerator: '))
4.          denominator = int(input('Enter a denominator: '))
5.          result = numerator / denominator
6.          print(numerator, 'over', denominator, 'equals', result)
7.      except ValueError:
8.          print('Integers only please. Try again.')
9.          divide()
10.     except ZeroDivisionError:
11.         print('You cannot divide by zero! Try again.')
12.         divide()
13.     except KeyboardInterrupt:
14.         print('Quitter!')
15.     except:
16.         print('I have no idea what went wrong!')
17.
18. def main():
19.     divide()
20.
21. main()
```

Code Explanation

1. If the user enters anything other than an integer for the numerator or denominator, the `int()` function will fail and a `ValueError` exception will be thrown.
2. If the user enters `0` for the denominator, a `ZeroDivisionError` exception will be thrown.
3. If the user quits the program by pressing **CTRL+C**, a `KeyboardInterrupt` exception will be thrown.

4. The last `except` clause, which is optional, serves as a wildcard to catch any unspecified exception types.

Run the program and try entering different values to see how it works.

```
PS ..\exception-handling\Demos> python specific.py
Integers only please. Try again.
Enter a numerator: 10
Enter a denominator: 0
You cannot divide by zero! Try again.
Enter a numerator: 10
Enter a denominator: 2
10 over 2 equals 5.0
PS ..\exception-handling\Demos> python specific.py
Enter a numerator: 5
Enter a denominator: Quitter!
```

At the end, the user pressed **Ctrl+C** to end the program.

Wildcard except Clauses

Including a wildcard `except` clause can be dangerous because it can hide a real error. If you do use it, you may want to print a friendly error message and then reraise the error, like this:

Demo 63: exception-handling/Demos/reraise.py

```
1.  try:
2.      1/0
3.  except:
4.      print('Something really bad just happened! Oh no, oh no, oh no!')
5.      raise
```

Code Explanation

This first prints the friendly error (if you want it to be friendly) and then outputs the interpreter's traceback:

```
Something really bad just happened! Oh no, oh no, oh no!  
Traceback (most recent call last):  
File ".\reraise.py", line 2, in <module>  
1/0  
ZeroDivisionError: division by zero
```

Getting Information on Exceptions

To access the exception object, assign it to a variable using the `as` keyword, like this:

Demo 64: `exception-handling/Demos/details.py`

```
1.  try:  
2.      1/0  
3.  except Exception as e:  
4.      print(type(e)) # prints exception type  
5.      print(e) # prints friendly message
```


Code Explanation

This will output the following:

```
<class 'ZeroDivisionError'>  
division by zero
```

Note that `e` is an arbitrary variable name. You can name the variable whatever you like.

Exercise 25: Raising Exceptions

 15 to 25 minutes

In this exercise, you will intentionally try to create different types of exceptions using the following template:

```
try:
    # code that creates exception
except Exception as e:
    print(type(e))
    print(e, '\n')
```

1. Open `exception-handling/Exercises/exception_details.py` in your editor.
2. Finish the `try/except` blocks to raise the following types of errors:
 - A. `ZeroDivisionError` – This one is done for you.
 - B. `ValueError`
 - C. `NameError`
 - D. `FileNotFoundError`
 - E. `ModuleNotFoundError`
 - F. `TypeError`
 - G. `AttributeError`
 - H. `StopIteration`
 - I. `KeyError`
3. Use the documentation at <https://docs.python.org/3/library/exceptions.html#base-classes> as necessary.

```
1.  # ZeroDivisionError
2.  try:
3.      1/0
4.  except Exception as e:
5.      print(type(e))
6.      print(e, '\n')
7.
8.  # ValueError
9.  try:
10.     int('a')
11. except Exception as e:
12.     print(type(e))
13.     print(e, '\n')
14.
15. # NameError
16. try:
17.     print(foo)
18. except Exception as e:
19.     print(type(e))
20.     print(e, '\n')
21.
22. # FileNotFoundError
23. try:
24.     open('non-existing-file.txt', 'r')
25. except Exception as e:
26.     print(type(e))
27.     print(e, '\n')
28.
29. # ImportError
30. try:
31.     import non_existing_module
32. except Exception as e:
33.     print(type(e))
34.     print(e, '\n')
35.
36. # TypeError
37. try:
38.     nums = [1, 2] # Lists are iterables, not iterators
39.     next(nums)
40. except Exception as e:
41.     print(type(e))
42.     print(e, '\n')
43.
44. # AttributeError
```

```
45. try:
46.     greeting = 'Hello'
47.     greeting.print() # strings don't have a print() method
48. except Exception as e:
49.     print(type(e))
50.     print(e, '\n')
51.
52. # StopIteration
53. try:
54.     nums = [1, 2]
55.     iter_nums = iter(nums)
56.     print(next(iter_nums))
57.     print(next(iter_nums))
58.     print(next(iter_nums))
59. except Exception as e:
60.     print(type(e))
61.     print(e, '\n')
62.
63. # KeyError
64. try:
65.     grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
66.     print(grades['Science'])
67. except Exception as e:
68.     print(type(e))
69.     print(e, '\n')
```

Code Explanation

The preceding code will render the following:

```
<class 'ZeroDivisionError'>
division by zero

<class 'ValueError'>
invalid literal for int() with base 10: 'a'

<class 'NameError'>
name 'foo' is not defined

<class 'FileNotFoundError'>
[Errno 2] No such file or directory: 'non-existing-file.txt'

<class 'ModuleNotFoundError'>
No module named 'non_existing_module'

<class 'TypeError'>
'list' object is not an iterator

<class 'AttributeError'>
'str' object has no attribute 'print'

1
2
<class 'StopIteration'>

<class 'KeyError'>
'Science'
```

The else Clause

You should limit your try clause to the code that might cause an exception. After the final except clause, you can include an else clause that contains code that only runs if no exception was raised in the try clause. Take a look at the following example:

Demo 65: exception-handling/Demos/else.py

```
1.  def divide():
2.      try:
3.          numerator = int(input('Enter a numerator: '))
4.          denominator = int(input('Enter a denominator: '))
5.          result = numerator / denominator
6.      except ValueError:
7.          print('Integers only please. Try again.')
8.          divide()
9.      except ZeroDivisionError:
10.         print('You cannot divide by zero! Try again.')
11.         divide()
12.     except KeyboardInterrupt:
13.         print('Quitter!')
14.         raise
15.     except:
16.         print('I have no idea what went wrong!')
17.     else:
18.         print(numerator, 'over', denominator, 'equals', result)
19.
20. def main():
21.     divide()
22.
23. main()
```

Code Explanation

As we're fairly confident that the final `print()` function call will not raise a `ValueError` or a `ZeroDivisionError`, it makes sense to include it within the `else` clause.

The finally Clause

The `finally` clause is for doing any necessary cleanup (e.g., closing connections, releasing resources, etc.). You would generally only use it in a nested `try/except` block. Here is a pseudo-example:

```

try:
    resource = Resource() # Some generic resource
    resource.open() # Open resource

    # If the resource fails to open, execution will
    # jump to the outer except block, skipping the
    # whole try block below
    try:
        # Do something with resource
        resource.do_something()
    except:
        # Re-raise error to outer try block
        raise
    else:
        do_something_safe()
    finally:
        # Close resource. This will happen even if an
        # exception occurred in the inner except block above
        resource.close()
except Exception as e:
    # resource failed to open, so we don't need to close it
    print('Something bad happened:', e)
else:
    do_more_cool_stuff()

```

The crux of this code is that the resource should only be closed (and *always* be closed) if it was successfully opened, whether or not some exception occurred after it was opened. If the resource fails to open, then we should not try to close it, so `resource.close()` cannot go in a hypothetical `finally` block in the outer `try`. Note that this is an advanced scenario, so don't worry too much if you don't fully understand it. Generally, you won't have to use the `finally` block in exception handling.

Using Exceptions for Flow Control

In Python, it's not uncommon to use exception handling as a method of flow control. For example, rather than using a condition to check to make sure everything is in order before proceeding, your code can simply try to proceed and then respond appropriately if an exception is raised. Compare the following `square_num()` function, which uses exception handling, with the `cube_num()` function, which uses an `if` condition to accomplish the same thing:

Demo 66: exception-handling/Demos/try_else.py

```
1. def square_num():
2.     try:
3.         num = int(input('Input Integer: '))
4.     except ValueError:
5.         print('That is not an integer.')
6.     else:
7.         print(num, 'squared is', num**2)
8.
9. def cube_num():
10.    num = input('Input Number: ')
11.    if num.isdigit():
12.        print(num, 'cubed is', int(num)**3)
13.    else:
14.        print('That is not an integer.')
```

Code Explanation

The difference between using if/else and try/except for flow control is the difference between asking for permission and asking for forgiveness.

Exercise 26: Running Sum

 10 to 20 minutes

In this exercise, you will change a function that uses an if-else construct to use a try-except construct instead.

1. Open `exception-handling/Exercises/running_sum.py` in your editor and study the code. The code repeatedly prompts the user for a number, adding each valid entry to a total. If the user enters “q”, the program quits. If they enter any other non-integer value, the function prompts them for integers only. Run the script to see how it works.
2. Now, replace the if-else block with a try-except block. The script should continue to work in exactly the same way.

Exercise Code: `exception-handling/Exercises/running_sum.py`

```
1. def main():
2.     total = 0
3.     while True:
4.         num = input('Enter a number (q to quit): ')
5.
6.         if num.lower() == 'q':
7.             print('Goodbye!')
8.             break
9.
10.        if not num.isdigit():
11.            print('Integers only please. Try again.')
12.        else:
13.            total += int(num)
14.            print('The current total is:', total)
15.
16.    main()
```



```
1. def main():
2.     total = 0
3.     while True:
4.         try:
5.             num = input('Enter a number: ')
6.             if num.lower() == 'q':
7.                 print('Goodbye!')
8.                 break
9.             num = int(num) # This might cause an error
10.        except ValueError:
11.            print('Integers only please. Try again.')
12.        else:
13.            total += num
14.            print('The current total is:', total)
15.
16. main()
```

Raising Your Own Exceptions

Sometimes it can be useful to raise your own exceptions, which you do with the `raise` statement. To illustrate, consider the following function, which creates a bi-directional dictionary from a dictionary:

```
def bidict(d):
    d2 = d.copy()
    for k,v in d.items():
        d2[v] = k
    return d2
```

If you pass `{'hola': 'hi', 'adios': 'bye'}` to this function, it will return this dictionary:

```
{'hola': 'hi', 'adios': 'bye', 'hi': 'hola', 'bye': 'adios'}
```

You can use this new dictionary to look up values in either direction. But what if you pass `bidict()` a dictionary that has two keys with the same values, like this:

```
{'hola': 'hi', 'adios': 'bye', 'ciao': 'bye'}
```

It won't error, but it won't return the mapping you want, because the `bye` key can only have one value:

```
{'hola': 'hi', 'adios': 'bye', 'ciao': 'bye', 'hi': 'hola', 'bye': 'adios'}
```

The value of the `bye` key is `'adios'`, but there is no key with the value of `'ciao'`.

Consider the loop in the `bidict()` function. Each time it assigns a value to a key. But if that key already exists, it overwrites the existing value.

One way to handle this is to raise an exception, like this:

```
def bidict(d):
    d2 = d.copy()
    for k,v in d.items():
        if v in d2.keys():
            raise KeyError('Cannot create bidirectional dict ' +
                           'with duplicate keys.')
    d2[v] = k
    return d2
```

Now, when a programmer tries to pass a dict that won't work with the function, it will raise an exception. Here is the full code:

Demo 67: exception-handling/Demos/bidict_with_exception.py

```
1.  def bidict(d):
2.      d2 = d.copy()
3.      for k, v in d.items():
4.          if v in d2.keys():
5.              raise KeyError('Cannot create bidirectional dict ' +
6.                             'with duplicate keys.')
7.          d2[v] = k
8.      return d2
9.
10. translator = bidict({'hola': 'hi', 'adios': 'bye'})
11. print(translator['hola'])
12. print(translator['hi'])
13.
14. translator2 = bidict({'hola': 'hi', 'adios': 'bye', 'ciao': 'bye'})
```

Code Explanation

Run this and you will see that the call creating `translator` works, but the call creating `translator2` does not:

```
hi
hola
Traceback (most recent call last):
File ".\bidict_with_exception.py", line 14, in <module>
translator2 = bidict({'hola':'hi', 'adios':'bye', 'ciao':'bye'})
File ".\bidict_with_exception.py", line 5, in bidict
raise KeyError('Cannot create bidirectional dict ' +
KeyError: 'Cannot create bidirectional dict with duplicate keys.'
```

Conclusion

In this lesson, you have learned to handle Python exceptions. For further study, see <https://docs.python.org/3/tutorial/errors.html>

LESSON 9

Python Dates and Times

Topics Covered

- ☑ The `time` module.
- ☑ The `datetime` module.

It was the best of times, it was the worst of times,... it was the epoch of belief, it was the epoch of incredulity...

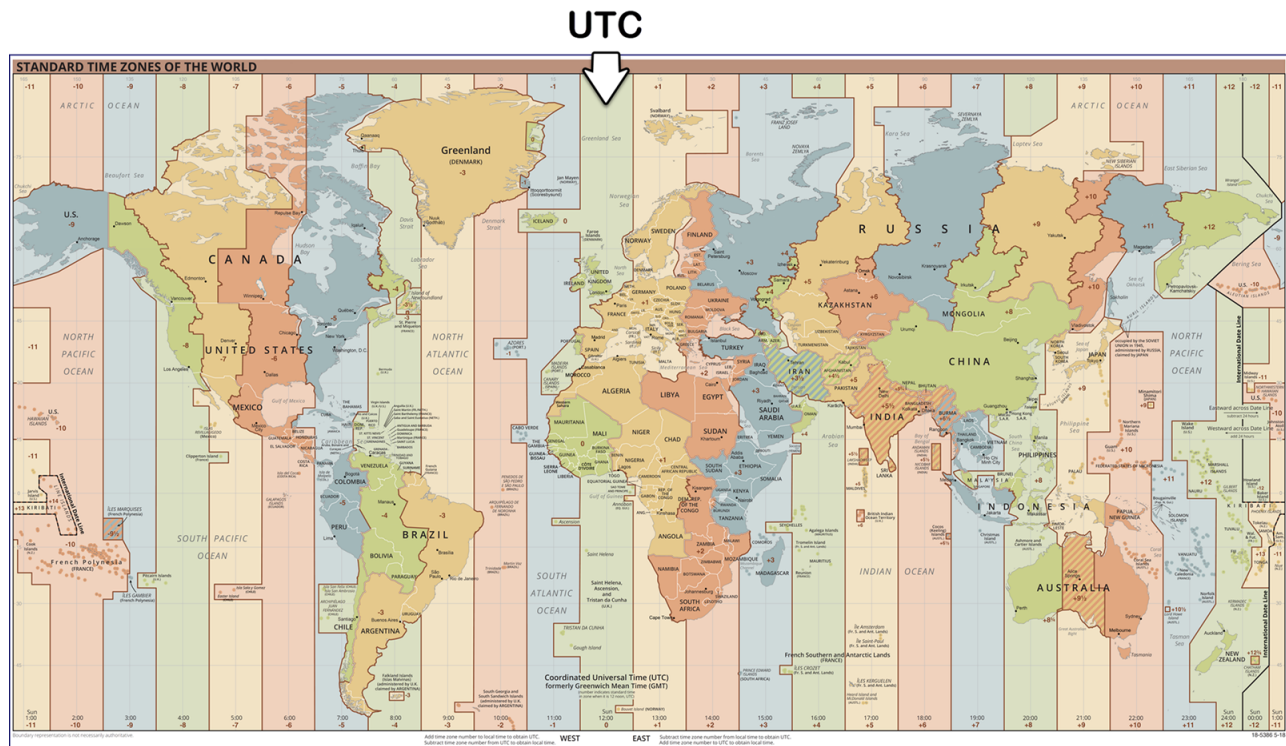
– *A Tale of Two Cities*, Charles Dickens

Introduction

Working with dates and times can be tricky in programming. Python has some great built-in modules for helping with this. You'll learn about these modules in this lesson.

Understanding Time

Before we get into the Python code, it's important to understand a little about how computer languages, including Python, understand time. In particular, time is not the same across geography. The time in Moscow is different from the time in New York City. These differences wouldn't cause too much of a problem if they were consistent, but they are not, in large part because Daylight Saving Time (DST) is practiced in some parts of the world, but not in others, and it goes into effect on different dates and times. Even that wouldn't be too bad if there were some scientific way of determining where and when times were changed. But, alas, there is not. The decision to practice Daylight Saving Time is a political one, not a scientific one. Because of this, we cannot rely on local times to make exact calculations. **Coordinated Universal Time** (UTC) to the rescue. UTC, the successor to GMT (Greenwich Mean Time), is the standard by which we measure time. The following image shows world time zones as offsets:



World Time Zones on May 7, 2019 at 12:48³⁰

Whether or not you need to be concerned with UTC time and time offsets will depend on the data with which you are working and the type of problem that you are trying to solve.

❖ The Epoch

The **epoch** is the moment that a computer or computer language considers time to have started. Python considers the epoch to be January 1, 1970 at midnight (1970-01-01 00:00:00).³¹ Times before the epoch are expressed internally as negative numbers.

❖ Python and Time

The most useful built-in modules for working with dates and times in Python are `time` and `datetime`.

30. The https://commons.wikimedia.org/wiki/File:World_Time_Zones_Map.png image is used under the terms of Public Domain (https://en.wikipedia.org/wiki/Copyright_status_of_works_by_the_federal_government_of_the_United_States).

31. The time of the epoch varies across computing systems, but in Python it is pretty much guaranteed to be January 1, 1970. For a discussion on this, see <https://grokbase.com/t/python/python-dev/086gxjdb5a/epoch-and-platform>.

The time Module³².

The `time` module is useful for comparing moments in time, especially for testing the performance of your code. It can also be used for accessing and manipulating dates and times.

❖ Clocks

Python's `time` module includes these different types of clocks:

1. `time.monotonic()`
2. `time.perf_counter()`
3. `time.process_time()`
4. `time.time()`

Absolute Time

Of the clocks, only `time.time()` measures the *absolute time*: the number of seconds since the epoch, and can be converted to an actual time of day.

Demo 68: date-time/Demos/what_time_is_it.py

```
1.  import time
2.
3.  seconds_since_epoch = time.time()
4.  minutes_since_epoch = seconds_since_epoch / 60
5.  hours_since_epoch = minutes_since_epoch / 60
6.  days_since_epoch = hours_since_epoch / 24
7.  years_since_epoch = days_since_epoch / 365.25
8.
9.  print("""s: {:,}
10. m: {:,}
11. h: {:,}
12. d: {:,}
13. y: {:,}""".format(seconds_since_epoch,
14.                    minutes_since_epoch,
15.                    hours_since_epoch,
16.                    days_since_epoch,
17.                    years_since_epoch, sep='\n'))
```

^{32.} Documentation on the `time` module: <https://docs.python.org/3/library/time.html>

Code Explanation

The preceding code will render the following (on March 06, 2020 at 08:28:21):

```
s: 1,583,501,301.986899  
m: 26,391,688.366448317  
h: 439,861.4727741386  
d: 18,327.56136558911  
y: 50.17812831099003
```

Relative Time

The other clocks return relative times from an indeterminate start time. They are useful in determining differences between moments of time. The most useful of these will be `time.perf_counter()` as it provides the most precise results. It is often used to measure how quickly a piece of code runs.

To illustrate how you would use this, assume that you wanted to create a list of random integers between 1 and 100 and that you needed the list as a string.

The most straightforward way of doing this is to create a loop and concatenate a new random number onto your string with each iteration.

But string concatenation is much less efficient than appending to a list, so it's actually faster to create a list, append a random number with each iteration, and then when the loop is complete join the list into a string.

And for a slightly faster solution, use a list comprehension. Here's the code:

Demo 69: date-time/Demos/compare_times.py

```
1.  import time
2.  import random
3.
4.  iterations = int(input('Number of iterations: '))
5.
6.  # Concatenating strings
7.  start_time = time.perf_counter()
8.  numbers = ''
9.  for i in range(iterations):
10.     num = random.randint(1, 100)
11.     numbers += ',' + str(num)
12. end_time = time.perf_counter()
13. td1 = end_time - start_time
14.
15. # Appending to a list
16. start_time = time.perf_counter()
17. numbers = []
18. for i in range(iterations):
19.     num = random.randint(1, 100)
20.     numbers.append(str(num))
21. numbers = ','.join(numbers)
22. end_time = time.perf_counter()
23. td2 = end_time - start_time
24.
25. # Using a list comprehension
26. start_time = time.perf_counter()
27. numbers = [str(random.randint(1, 100)) for i in range(1, iterations)]
28. numbers = ','.join(numbers)
29. end_time = time.perf_counter()
30. td3 = end_time - start_time
31.
32. print(f"""Number of numbers: {iterations:},
33. Time Delta 1: {td1}
34. Time Delta 2: {td2}
35. Time Delta 3: {td3}
36. td1 is {round(td1/td3, 2)}x slower than td3.\n""")
```

Code Explanation

And here are the results using different values for integers:

```
PS ...date-time\Demos> python compare_times.py
Number of iterations: 10000
Number of numbers: 10,000
Time Delta 1: 0.011690399999999999
Time Delta 2: 0.009983500000000145
Time Delta 3: 0.00927950000000033
td1 is 1.26x slower than td3.
```

```
PS ...date-time\Demos> python compare_times.py
Number of iterations: 100000
Number of numbers: 1,000,000
Time Delta 1: 1.8037353
Time Delta 2: 1.0743994
Time Delta 3: 0.9399708000000002
td1 is 1.92x slower than td3.
```

```
PS ...date-time\Demos> python compare_times.py
Number of iterations: 1000000
Number of numbers: 10,000,000
Time Delta 1: 60.20499050000001
Time Delta 2: 11.402855500000001
Time Delta 3: 9.909772900000007
td1 is 6.08x slower than td3.
```

Notice that the speed differential increases markedly when dealing with large amounts of data.

The timeit Module

Python includes a `timeit`³³ module specifically for testing performance.

Time Structures

In addition to expressing time in seconds since the epoch, as `time.time()` does, time can be expressed as a special `time.struct_time` object, which is a type of tuple. The `time.struct_time` object for the epoch looks like this:

33. <https://docs.python.org/3/library/timeit.html>

```
time.struct_time(tm_year=1970,
                 tm_mon=1, # 1 = January
                 tm_mday=1,
                 tm_hour=0,
                 tm_min=0,
                 tm_sec=0,
                 tm_wday=3, # 3 = Thursday (0 = Monday)
                 tm_yday=1, # 1 = First day of year
                 tm_isdst=0) # 0 = No daylight savings time
```

The same time expressed as a string looks like this:

```
'Thu Jan  1 00:00:00 1970'
```

Methods that create `time.struct_time` objects include `time.gmtime()` and `time.localtime()`.

`time.gmtime([secs])`

`time.gmtime([secs])` converts a time expressed in seconds since the epoch to a `struct_time` in *Coordinated Universal Time (UTC)*. If the `secs` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch. The value of `tm_isdst` will always be 0, meaning no daylight savings time.

Epoch as struct_time

```
>>> epoch = time.gmtime(0)
>>> epoch
time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1,
                 tm_hour=0, tm_min=0, tm_sec=0,
                 tm_wday=3, tm_yday=1, tm_isdst=0)
```

Current Time as struct_time

```
>>> now = time.gmtime()
>>> now
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                 tm_hour=20, tm_min=11, tm_sec=52,
                 tm_wday=4, tm_yday=66, tm_isdst=0)
```

Yesterday as struct_time

```
>>> day_in_seconds = 60*60*24
>>> yesterday = time.gmtime(time.time() - day_in_seconds)
>>> yesterday
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=5,
                  tm_hour=20, tm_min=12, tm_sec=2,
                  tm_wday=3, tm_yday=65, tm_isdst=0)
```

Tomorrow as struct_time

```
>>> day_in_seconds = 60*60*24
>>> tomorrow = time.gmtime(time.time() + day_in_seconds)
>>> tomorrow
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=7,
                  tm_hour=20, tm_min=12, tm_sec=5,
                  tm_wday=5, tm_yday=67, tm_isdst=0)
```

`time.localtime([secs])`

`time.localtime([secs])` is similar to `time.gmtime([secs])`, but it converts a time expressed in seconds since the epoch to a `struct_time` *in the local time*. As with `time.gmtime([secs])`, if the `secs` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch. The value of `tm_isdst` will be 0 or 1 depending on whether daylight savings time applies.

Epoch as Local struct_time

```
>>> epoch_offset = time.localtime(0) # "offset," because it's local time
>>> epoch_offset
time.struct_time(tm_year=1969, tm_mon=12, tm_mday=31,
                  tm_hour=19, tm_min=0, tm_sec=0,
                  tm_wday=2, tm_yday=365, tm_isdst=0)
```

Current Time as Local struct_time

```
>>> now = time.localtime()
>>> now
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=16, tm_min=20, tm_sec=36,
                  tm_wday=4, tm_yday=66, tm_isdst=0)
```

Yesterday as Local struct_time

```
>>> day_in_seconds = 60*60*24
>>> yesterday = time.localtime(time.time() - day_in_seconds)
>>> yesterday
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=16, tm_min=20, tm_sec=36,
                  tm_wday=4, tm_yday=66, tm_isdst=0)
```

Tomorrow as Local struct_time

```
>>> tomorrow = time.localtime(time.time() + day_in_seconds)
>>> tomorrow
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=7,
                  tm_hour=16, tm_min=20, tm_sec=36,
                  tm_wday=5, tm_yday=67, tm_isdst=0)
```

To find out if daylight savings time currently applies in your timezone, run this:

```
time.localtime().tm_isdst
```

time.mktime()

The inverse of `time.localtime()` is `time.mktime()`, which takes a `struct_time` and returns the number of seconds since the epoch:

```
>>> lt = time.localtime()
>>> lt
time.struct_time(tm_year=2020, tm_mon=3, tm_mday=6,
                  tm_hour=14, tm_min=33, tm_sec=32,
                  tm_wday=4, tm_yday=66, tm_isdst=0)

>>> seconds_since_epoch = time.mktime(lt)
>>> seconds_since_epoch
1583523234.0
```

Times as Strings

When comparing or doing any sort of calculations with dates and times, it is necessary to treat them as objects or numbers; however, when outputting dates in reports, it is more useful to see their human-readable string representations.

Methods that create times as strings include :

`time.asctime([t])`

`time.asctime([t])` converts a time expressed as a `struct_time` to a date represented as a string. If the `struct_time` argument is omitted, it defaults to `time.localtime()`, which returns the current local time.

String Representation of Epoch

```
>>> time.asctime(time.gmtime(0))
'Thu Jan  1 00:00:00 1970'
```

String Representation of Current Local Time

```
>>> time.asctime()
'Fri Mar  6 09:41:24 2020'
```

`time.ctime([secs])`

`time.ctime([secs])` is like `time.asctime([t])`, but it takes a time expressed in seconds since the epoch instead of a `struct_time`. It returns a date in local time represented as a string. If the `secs` `struct_time` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch.

```
>>> str_epoch_offset = time.ctime(0) # "offset," because it's local time
>>> str_epoch_offset
'Wed Dec 31 19:00:00 1969'

>>> str_now = time.ctime()
>>> str_now
'Fri Mar  6 09:57:17 2020'
```


Time and Formatted Strings

```
time.strftime(format[, t])
```

The `time.strftime(format[, t])` takes a `struct_time` object and returns a formatted string. The “f” in “strftime” is for “format.”

```
time.strptime(string[, format])
```

The `time.strptime(string[, format])` takes a string representing a time and returns a `struct_time` object. The “p” in “strptime” is for parse.”

Formatting Directives

Some of the more common formatting directives³⁴ are shown in the following list:

- `%a` – Abbreviated weekday name.
- `%A` – Full weekday name.
- `%b` – Abbreviated month name.
- `%B` – Full month name.
- `%d` – Day of the month as a decimal number. Possible values: 01 - 31.
- `%H` – Hour (24-hour clock) as a decimal number. Possible values: 00 - 23.
- `%I` – Hour (12-hour clock) as a decimal number. Possible values: 01 - 12.
- `%j` – Day of the year as a decimal number. Possible values: 001 - 366.
- `%m` – Month as a decimal number. Possible values: 01 - 12.
- `%M` – Minute as a decimal number. Possible values: 00 - 59.
- `%p` – Locale’s equivalent of either AM or PM.
- `%S` – Second as a decimal number. Possible values: 00 - 59.
- `%w` – Weekday as a decimal number. Possible values: 0 (Sunday) – 6.
- `%x` – Locale’s appropriate date representation.

³⁴. Documentation on date and time formatting directives is available at https://docs.python.org/3/library/date_time.html?highlight=strptime#strftime-and-strptime-format-codes.

- `%X` – Locale’s appropriate time representation.
- `%c` – Locale’s appropriate date and time representation.
- `%y` – Year without century as a decimal number. Possible values: 00 - 99.
- `%Y` – Year with century as a decimal number.
- `%Z` – Time zone name (no characters if no time zone exists).

The following demo provides some examples of `time.strftime()` and `time.strptime()`.

Demo 70: date-time/Demos/formatting_times.py

```
1. import time
2.
3. epoch = time.gmtime(0)
4. print(time.strftime('%c', epoch)) # 01/01/70 00:00:00
5. print(time.strftime('%x', epoch)) # 01/01/70
6. print(time.strftime('%X', epoch)) # 00:00:00
7. print(time.strftime('%A, %B %d, %Y, %I:%M %p', epoch))
8. # Thursday, January 01, 1970, 12:00 AM
9.
10. independence_day = time.strptime('07/04/1776', '%m/%d/%Y')
11. print(independence_day)
```

Code Explanation

This will output:

```
Thu Jan  1 00:00:00 1970
01/01/70
00:00:00
Thursday, January 01, 1970, 12:00 AM
time.struct_time(tm_year=1776, tm_mon=7, tm_mday=4,
                  tm_hour=0, tm_min=0, tm_sec=0,
                  tm_wday=3, tm_yday=186, tm_isdst=-1)
```

Pausing Execution with `time.sleep()`

The `time.sleep(secs)` method suspends execution for a given number of seconds. Run the following demo to see how it works:

Demo 71: date-time/Demos/sleep.py

```
1. import time
2.
3. for i in range(10, 0, -1):
4.     print(i)
5.     time.sleep(.5)
6.
7. print('Blast off!')
```

Code Explanation

This will output the following, printing a new line every half second:

```
10
9
8
7
6
5
4
3
2
1
Blast off!
```

`time.sleep()` can be used to create a digital clock:

Demo 72: date-time/Demos/clock.py

```
1. import time
2.
3. def start_clock():
4.     print("Starting Clock")
5.     try:
6.         while True:
7.             localtime = time.localtime()
8.             result = time.strftime("%I:%M:%S %p", localtime)
9.             print(result)
10.            time.sleep(1)
11.        except KeyboardInterrupt:
12.            print("Stopping Clock")
13.
14. start_clock()
```

Code Explanation

This will output something like the following, printing a new time every second until the user presses **Ctrl+C**:

```
Starting Clock
10:45:02 AM
10:45:03 AM
10:45:04 AM
10:45:05 AM
10:45:06 AM
Stopping Clock
```

The datetime Module³⁵

The datetime module includes the following types:

1. `datetime.date` – a date with year, month, and day attributes.
2. `datetime.time` – a time with hour, minute, second, microsecond, and tzinfo attributes.
3. `datetime.datetime` – a combination of `datetime.date` and `datetime.time`.

³⁵. Documentation on the datetime module: <https://docs.python.org/3/library/datetime.html>.

4. `datetime.timedelta` – a duration expressing the difference between instances of two `datetime.date`, `datetime.time`, or `datetime.datetime` objects.

❖ `datetime.date` Objects

There are a number of `datetime` methods for creating `datetime.date` objects:

1. `datetime.date(year, month, day)` – Creates a local date.

```
>>> datetime.date(1776, 7, 4)
datetime.date(1776, 7, 4)
```

2. `datetime.date.today()` – Returns the current local date.

```
>>> datetime.date.today()
datetime.date(2020, 1, 31)
```

3. `datetime.date.fromtimestamp(secs)` – Returns the local date from `secs` seconds since the epoch:

```
>>> week_in_seconds = 24*60*60*7
>>> datetime.date.fromtimestamp(week_in_seconds) # One week after epoch
datetime.date(1970, 1, 7)
```

`datetime.date` Attributes

A `datetime.date` instance includes the following attributes:

- `year`
- `month`
- `day`

```
>>> i_day = datetime.date(1776, 7, 4)
>>> i_day.year, i_day.month, i_day.day
(1776, 7, 4)
```

`datetime.date` Methods

A `datetime.date` instance includes the following methods:

- `replace()` – Returns a new `datetime.date` instance based on `date` with the given replacements.

```
>>> i_day = datetime.date(1776, 7, 4)
>>> i_day.replace(year=1826)
datetime.date(1826, 7, 4)
```

- `timetuple()` – Returns a `struct_time`.

```
>>> i_day.timetuple()
time.struct_time(tm_year=1776, tm_mon=7, tm_mday=4,
                 tm_hour=0, tm_min=0, tm_sec=0,
                 tm_wday=3, tm_yday=186, tm_isdst=-1)
```

- `weekday()` – Returns an integer representing the day of the week.

```
>>> i_day.weekday()
3
```

- `ctime()` – Returns a formatted date string. Similar to `time.ctime()`.

```
>>> i_day.ctime()
'Thu Jul  4 00:00:00 1776'
```

- `strftime()` – Returns a formatted date string. Similar to `time.strftime(format[, t])` with the same formatting directives.

```
>>> i_day.strftime('%A, %B %d, %Y, %I:%M %p')
'Thursday, July 04, 1776, 12:00 AM'
```

❖ `datetime.time` Objects

`datetime.time` objects are created with the `datetime.time()` method, which takes the following arguments:

- `hour` – defaults to 0.
- `minute` – defaults to 0.
- `second` – defaults to 0.
- `microsecond` – defaults to 0.
- `tzinfo` – defaults to `None`, which makes the `datetime` object “naive,” meaning that it is unaware of timezones. It is common to set `tzinfo` to UTC time like this `tzinfo=datetime.timezone.utc`.

`datetime.time` Attributes

A `datetime.time` instance includes the following attributes:

- `hour`
- `minute`
- `second`
- `microsecond`

```
>>> t = datetime.time(hour=14, minute=13, second=12,
                      microsecond=11, tzinfo=datetime.timezone.utc)
>>> t.hour, t.minute, t.second, t.microsecond, t.tzinfo
(14, 13, 12, 11, datetime.timezone.utc)
```

`datetime.time` Methods

A `datetime.time` instance includes the following methods:

- `replace()` – Returns a new `datetime.time` instance based on `time` with the given replacements.

```
>>> t = datetime.time(hour=14, minute=13, second=12,
                      microsecond=11, tzinfo=datetime.timezone.utc)
>>> t.replace(hour=4)
datetime.time(4, 13, 12, 11, tzinfo=datetime.timezone.utc)
```

- `strftime()` – Returns a formatted date string. Similar to `time.strftime(format[, t])` with the same formatting directives.

```
>>> t.strftime('%I:%M %p')  
'02:13 PM'
```

`datetime.datetime` Objects

A `datetime.datetime` object is a combination of a `datetime.date` object and a `datetime.time` object. There are a number of `datetime` methods for creating `datetime.datetime` objects:

1. `datetime(year, month, day, hour, minute, second, microsecond, tzinfo)` – creates new `datetime`
2. `datetime.today()` – Returns the current local date and time.
3. `datetime.now()` – like `datetime.today()` but allows for time zone to be set.
4. `datetime.utcnow()` – Returns the current UTC date and time.
5. `datetime.fromtimestamp(timestamp)` – Returns the local date and time corresponding to `timestamp`.
6. `datetime.utcfromtimestamp(timestamp)` – Returns the UTC date and time corresponding to `timestamp`.
7. `datetime.combine(datetime.date, datetime.time)` – Combines a `datetime.date` object and `datetime.time` object into a single `datetime.datetime` object.
8. `datetime.strptime(date_string, format)` – Takes a string representing a date and time and returns a `datetime.datetime` object.

`datetime.datetime` Attributes

A `datetime.datetime` instance includes the following attributes:

- `year`
- `month`
- `day`
- `hour`

- minute
- second
- microsecond

```
>>> moon_landing = datetime.datetime(year=1969, month=7, day=21,
                                     hour=2, minute=56, second=15,
                                     tzinfo=datetime.timezone.utc)
>>> moon_landing.year, moon_landing.month, moon_landing.day
(1969, 7, 21)

>>> moon_landing.hour, moon_landing.minute, moon_landing.second
(2, 56, 15)

>>> moon_landing.microsecond, moon_landing.tzinfo
(0, datetime.timezone.utc)
```

datetime.datetime Methods

A `datetime.datetime` instance includes the following methods:

- `datetime.replace(year, month, day, hour, minute, second, microsecond)` – Returns a new `datetime.datetime` instance based on `datetime` with the given replacements.

```
>>> moon_landing.replace(year=2019)
datetime.datetime(2019, 7, 21, 2, 56, 15,
                  tzinfo=datetime.timezone.utc)
```

- `datetime.date()` – Returns a `datetime.date` object with same year, month, and day.

```
>>> moon_landing.date()
datetime.date(1969, 7, 21)
```

- `datetime.time()` – Returns a `datetime.time` object with same hour, minute, second, and microsecond.

```
>>> moon_landing.time()
datetime.time(2, 56, 15)
```

- `datetime.timetuple()` – Returns a `struct_time` representing the local time.

```
>>> moon_landing.timetuple()
time.struct_time(tm_year=1969, tm_mon=7, tm_mday=21,
                 tm_hour=2, tm_min=56, tm_sec=15,
                 tm_wday=0, tm_yday=202, tm_isdst=-1)
```

- `datetime.utctimetuple()` – Returns a `struct_time` representing the UTC time.

```
>>> moon_landing.utctimetuple()
time.struct_time(tm_year=1969, tm_mon=7, tm_mday=21,
                 tm_hour=2, tm_min=56, tm_sec=15,
                 tm_wday=0, tm_yday=202, tm_isdst=0)
```

- `datetime.timestamp()` – Returns a timestamp corresponding to the `datetime`.

```
>>> moon_landing.timestamp()
-14159025.0
```

- `datetime.weekday()` – Returns an integer representing the day of the week.

```
>>> moon_landing.weekday()
0
```

- `datetime.ctime()` – Returns a formatted date string. Similar to `time.ctime()`.

```
>>> moon_landing.ctime()
'Mon Jul 21 02:56:15 1969'
```

- `datetime.strftime(format)` – Returns a formatted date string. Similar to `time.strftime(format[, t])` with the same formatting directives.

```
>>> moon_landing.strftime('%A, %B %d, %Y, %I:%M %p')
'Monday, July 21, 1969, 02:56 AM'
```



Exercise 27: What Color Pants Should I Wear?



15 to 30 minutes

1. Create a new file called `pant_color.py` in the `date-time/Exercises` folder.
2. Write an `is_summer()` function that takes one argument: a `datetime.datetime` object that defaults to the current time. The function should return `True` if the date is between June 20 and September 22 of the year, and `False` otherwise. You will need to construct dates marking the start of summer and the start of fall. To do so, you should make use of the year of the passed-in `datetime.datetime` object. For example, if the passed in date is in 2025, you must check if the date is between June 20, 2025 and September 22, 2025.
3. In the `main()` function, make a call to `is_summer()` and then print “You should wear white pants.” if it is summer or “You should wear black pants.” if it isn’t summer.
4. There are many ways to do this. Our solution makes use of the ternary operator (see page 165) in the `main()` function.

Solution: date-time/Solutions/pant_color.py

```
1.  from datetime import datetime
2.
3.  def is_summer(the_date=datetime.now()):
4.      # Get the year of the passed-in datetime.datetime object
5.      year = the_date.year
6.
7.      # Create datetime.datetime objects for starts of seasons
8.      summer_start = datetime(year, 6, 20)
9.      fall_start = datetime(year, 9, 22)
10.
11.     # Return true if passed-in date is between starts of seasons
12.     return (summer_start < the_date < fall_start)
13.
14.  def main():
15.      # Use ternary operator to assign pant color
16.      pant_color = 'white' if is_summer() else 'black'
17.      print(f'You should wear {pant_color} pants.')
18.
19.  main()
```

datetime.timedelta Objects

A `datetime.timedelta` object expresses a duration – the time between two date, time, or `datetime` objects.

`datetime.timedelta` objects can be...

- Added ($t1 + t2$). **Result:** a new `datetime.timedelta` object.
- Subtracted ($t1 - t2$). **Result:** a new `datetime.timedelta` object.
- Divided ($t1 / t2$). **Result:** a float.
- Multiplied by an integer or float ($t1 * 2$). **Result:** a new `datetime.timedelta` object.
- Divided by an integer or float ($t1 / 2$). **Result:** a new `datetime.timedelta` object.

datetime.timedelta Attributes

A `datetime.timedelta` instance includes the following attributes:

- `days`

- seconds
- microseconds

```
>>> now = datetime.datetime.today()
>>> starttime = now.replace(hour=8, minute=30, second=0, microsecond=0)
>>> endtime = now.replace(hour=8, minute=48, second=23, microsecond=0)
>>> racetime = endtime - starttime
>>> racetime
datetime.timedelta(seconds=1103)
>>> racetime.days, racetime.seconds, racetime.microseconds
(0, 1103, 0)
```

`datetime.timedelta.total_seconds()` Method

A `datetime.timedelta` instance includes only one method: `total_seconds()`, which returns the total number of seconds in the duration (with microsecond accuracy).

```
>>> racetime.total_seconds()
1103.0
```

The Time Delta Between Dates

To determine the time delta between two dates, simply subtract one from the other:

```
>>> start = datetime.datetime(year=1861, month=4, day=12)
>>> end = datetime.datetime(year=1865, month=4, day=9)
>>> delta = end - start
>>> delta.days
1458
```

Exercise 28: Report on Departure Times

 45 to 90 minutes

In this exercise, you will create a small report on departure times from July, 1980. All the data is in a text file (date-time/data/departure-data.txt). Some of the data is shown below:

Exercise Code: date-time/data/departure-data.txt

```
1.    *Scheduled Actual
2.    07/01/1980 2:40 AM 07/01/1980 4:00 AM
3.    07/01/1980 3:00 AM 07/01/1980 3:00 AM
4.    07/01/1980 4:40 AM 07/01/1980 4:40 AM
5.    07/01/1980 5:30 AM 07/01/1980 5:30 AM
6.    07/01/1980 6:00 AM 07/01/1980 6:01 AM
      -----Lines 7 through 1969 Omitted-----
1970. 07/25/1980 7:00 PM 07/25/1980 7:09 PM
1971. 07/25/1980 7:01 PM 07/25/1980 7:01 PM
1972. 07/25/1980 7:15 PM
1973. 07/25/1980 7:53 PM 07/25/1980 7:53 PM
1974. 07/25/1980 8:00 PM 07/25/1980 8:00 PM
      -----Lines 1975 through 2478 Omitted-----
2479. 07/31/1980 9:10 PM 07/31/1980 9:10 PM
2480. 07/31/1980 10:05 PM 07/31/1980 10:05 PM
2481. 07/31/1980 10:45 PM 07/31/1980 10:45 PM
2482. 07/31/1980 11:15 PM 07/31/1980 11:15 PM
```

Things to note:

1. The first line is a header.
2. Each subsequent line has a tab separating a planned departure date and an actual departure date.
3. Some lines have a planned date, but no actual departure date. That means the trip was cancelled.

The file you will be working on has been started already:

Exercise Code: date-time/Exercises/departure_report.py

```
1.  from datetime import datetime
2.
3.  def get_departures():
4.      departures = []
5.      with open('../data/departure-data.txt') as f:
6.          for line in f.read().splitlines():
7.              departure = get_departure(line)
8.              if departure:
9.                  departures.append(departure)
10.     return departures
11.
12. def get_departure(line):
13.     """Return a tuple containing two  datetime objects."""
14.
15.     # If the line begins with an asterisk (*), return None
16.
17.     # Get the planned and actual departures as strings by
18.     # splitting the line on a tab character into a list
19.     # Assign the first item in the list to planned and the
20.     # second item to actual
21.
22.     # Convert the planned departure time to a datetime and
23.     # assign the result to date_planned
24.
25.     # For those lines that have an actual departure time,
26.     # convert the actual departure time to a datetime and
27.     # assign the result to date_actual.
28.     # For lines that don't have an actual departure date, assign
29.     # None to date_actual.
30.
31.     # Return a tuple with date_planned and date_actual.
32.     return (date_planned, date_actual)
33.
34. def left_ontime(departure):
35.     planned = departure[0]
36.     actual = departure[1]
37.     if not actual:
38.         return False
39.     return actual == planned
40.
41. # Write the following four functions. They should
42. # all return a boolean value
43. def left_early(departure):
44.     pass
```

```

45.
46. def left_late(departure):
47.     pass
48.
49. def left_next_day(departure):
50.     pass
51.
52. def did_not_run(departure):
53.     pass
54.
55. def main():
56.     departures = get_departures()
57.     ontime_departures = [d for d in departures if left_ontime(d)]
58.     early_departures = [d for d in departures if left_early(d)]
59.     late_departures = [d for d in departures if left_late(d)]
60.     next_day_departures = [d for d in departures if left_next_day(d)]
61.     cancelled_trips = [d for d in departures if did_not_run(d)]
62.
63.     print(f"""Total Departures: {len(departures)}
64. Ontime Departures: {len(ontime_departures)}
65. Early Departures: {len(early_departures)}
66. Late Departures: {len(late_departures)}
67. Next Day Departures: {len(next_day_departures)}
68. Cancelled Trips: {len(cancelled_trips)}""")
69.
70. main()

```

The `main()` and `get_departures()` functions are complete. Your task is to complete the `get_departure()` function.

1. Open `date-time/Exercises/departure_report.py` in your editor.
2. Review the `main()` and `get_departures()` functions.
3. Your first job is to write the `get_departure()` function:
 - A. The first line is a header and begins with an asterisk (*). Return `None` for that line.
 - B. Each passed-in line is formatted as follows:

```
07/12/1980 7:53 PM\t07/12/1980 7:57 PM
```


The string before the `\t` represents the planned departure time. The string after the `\t` represents the actual departure time. Not all records have an actual departure time, indicating that the trip was cancelled. Those lines are formatted like this:

```
07/12/1980 7:53 PM\t
```

- C. Split the line on the tab into a two-element list.
 - D. Convert the planned departure time to a `datetime` and assign the result to `date_planned`.
 - E. For those lines that have an actual departure time, convert the actual departure time to a `datetime` and assign the result to `date_actual`. For lines that don't have an actual departure date, assign `None` to `date_actual`.
 - F. Return a tuple with `date_planned` and `date_actual`. This line is already written.
4. Study the `main()` function. See how it makes use of the `left_ontime()`, `left_early()`, `left_late()`, `left_next_day()`, and `did_not_run()` functions. The `left_ontime()` function is already written. Write the other four functions. They should all return a boolean value.

```
-----Lines 1 through 12 Omitted-----
13. def get_departure(line):
14.     """Return a tuple containing two datetime objects."""
15.
16.     if line[0] == '*':
17.         return None
18.
19.     departure = line.split('\t')
20.     planned = departure[0]
21.     actual = departure[1]
22.
23.     date_planned = datetime.strptime(planned, '%m/%d/%Y %I:%M %p')
24.
25.     if actual:
26.         date_actual = datetime.strptime(actual, '%m/%d/%Y %I:%M %p')
27.     else: # Date doesn't exist
28.         date_actual = None
29.
30.     return (date_planned, date_actual)
31.
32. def left_ontime(departure):
33.     """Return True if left ontime. False, otherwise."""
34.     planned = departure[0]
35.     actual = departure[1]
36.     if not actual:
37.         return False
38.     return actual == planned
39.
40. def left_early(departure):
41.     """Return True if left early. False, otherwise."""
42.     planned = departure[0]
43.     actual = departure[1]
44.     if not actual:
45.         return False
46.     if actual < planned:
47.         print('Early:', departure)
48.     return actual < planned
49.
50. def left_late(departure):
51.     """Return True if left late. False, otherwise."""
52.     planned = departure[0]
53.     actual = departure[1]
54.     if not actual:
55.         return False
```

```
56.         return actual > planned
57.
58. def left_next_day(departure):
59.     """Return True if departed next day. False, otherwise."""
60.     planned = departure[0]
61.     actual = departure[1]
62.     if not actual:
63.         return False
64.     return actual.day > planned.day
65.
66. def did_not_run(departure):
67.     """Return True if did not depart. False, otherwise."""
68.     return not departure[1]
        -----Lines 69 through 85 Omitted-----
```

Code Explanation

The result of running this file is shown below:

```
Total Departures: 2481
Ontime Departures: 1207
Early Departures: 0
Late Departures: 1260
Next Day Departures: 2
Cancelled Trips: 14
```

Conclusion

In this lesson, you have learned to work with the `time` and `datetime` modules.

LESSON 10

File Processing

Topics Covered

- ✓ Reading files.
- ✓ Creating and writing to files.
- ✓ Working with directories.
- ✓ Working with the `os` and `os.path` modules.

He made haste to sit down in his easy chair and opened the book. He tried to read, but he could not revive the very vivid interest he had felt before in Egyptian hieroglyphics.

— *Anna Karenina*, Leo Tolstoy

Introduction

Python allows you to access and modify files and directories on the operating system. Among other things, you can:

1. Open new or existing files and store them in *file object* variables.
2. Read file contents, all at once or line by line.
3. Append to file contents.
4. Overwrite file contents.
5. List directory contents.
6. Rename files and directories.

Opening Files

The built-in `open()` function will attempt to open a file at a given path and return a corresponding file object, which can be read and, if opened with the right permissions, written to. This is the most basic syntax for opening a file:

```
open(path_to_file, file_mode)
```

`path_to_file` can either be a relative or an absolute path. File modes are described in the following list:

1. `"r"` – Open for reading (default). Returns `FileNotFoundError` if the file doesn't exist.
2. `"w"` – Open for writing. If the file exists, existing content is erased. If the file doesn't exist, a new file is created.
3. `"x"` – Create and open for writing. Returns `FileExistsError` if the file already exists.
4. `"a"` – Open for appending to end of content. If the file doesn't exist, a new file is created.
5. `"a+"` – Open for appending to end of content and reading. If the file doesn't exist, a new file is created.
6. `"r+"` – Open for reading and writing.
7. `"w+"` – Open for writing and reading. If the file exists, existing content is erased. If the file doesn't exist, a new file is created.
8. `"b"` – Opens in binary mode.
9. `"t"` – Opens in text mode. This is the default.

```
open('poem.txt') # Open for reading
open('poem.txt', 'w') # Open for writing
open('poem.txt', 'r+') # Open for reading and writing
open('logo.png', 'r+b') # Open in binary mode for reading and writing
```

❖ Methods of File Objects

Reading Files

Once you have a file object, you can read the file contents using any of the following methods:

- `f.read(size)` – reads `size` bytes of the file. If `size` is omitted, the entire file is read.
- `f.readline()` – reads a single line up to and including the newline character (`\n`).
- `f.readlines()` – reads the file into a list split after the newline characters. Each line will end with a newline character.

- `f.read().splitlines()` – reads the file into a string and then splits on the newline characters. Lines will not end with newline characters.
- `list(f)` – does the same thing as `f.readlines()`.

Closing Files

You should always be sure to close files when you are done with them. This can be done with the `f.close()` method like this:

```
f = open('my_files/zen_of_python.txt')
# Do stuff with file
f.close()
```

While this works, it creates a potential problem. What if an exception occurs between the time the file is opened and closed? We will be left with a dangling reference to the file holding onto system resources and potentially blocking other applications from accessing the file.

As we have seen (see page 27), a better practice is to use the `with` keyword when working with files, like this:

```
with open(file) as f:
    do_stuff()
```

Using this structure, we don't have to explicitly close the file with `f.close()`. File objects have a special built-in `__exit__()` method that closes the file and is always called at the end of a `with` block even if code within the `with` block raises an exception. Often, you will read a file into a variable within a `with` block and then use the variable throughout the script, as shown in the following demo:

Demo 73: file-processing/Demos/with.py

```
1.  with open('my_files/zen_of_python.txt') as f:
2.      poem = f.read()
3.
4.  # The file is now closed, but we saved its content in the poem variable
5.  for i, line in enumerate(poem.splitlines(), 1):
6.      print(f"{i}. {line}")
```

Code Explanation

Run this file by opening the terminal at `file-processing/Demos/` and running:

```
python with.py
```

This will output:

```
1. The Zen of Python
2. Tim Peters
3. https://www.python.org/dev/peps/pep-0020/
4.
5. Beautiful is better than ugly.
6. Explicit is better than implicit.
7. Simple is better than complex.
8. Complex is better than complicated.
9. Flat is better than nested.
10. Sparse is better than dense.
11. Readability counts.
12. Special cases aren't special enough to break the rules.
13. Although practicality beats purity.
14. Errors should never pass silently.
15. Unless explicitly silenced.
16. In the face of ambiguity, refuse the temptation to guess.
17. There should be one-- and preferably only one --obvious way to do it.
18. Although that way may not be obvious at first unless you're Dutch.
19. Now is better than never.
20. Although never is often better than *right* now.
21. If the implementation is hard to explain, it's a bad idea.
22. If the implementation is easy to explain, it may be a good idea.
23. Namespaces are one honking great idea -- let's do more of those!
```




Exercise 29: Finding Text in a File

🕒 15 to 25 minutes

In this exercise, you will create a tool for searching through a file. This is the starting code:

Exercise Code: `file-processing/Exercises/word_search.py`

```
1. def search(word, text):
2.     """Return tuple holding line num and line text (or None)."""
3.     pass
4.
5. def main():
6.     # Open my_files/zen_of_python.txt and
7.     # create a list from its contents
8.     # Save the list as zop
9.
10.    word = input('Enter search word: ')
11.    result = search(word, zop)
12.    if result:
13.        print(word, ' first appears on line ',
14.              result[0], ': ', result[1], sep='')
15.    else:
16.        print(word, 'was not found.')
17.
18. main()
```

1. Open `file-processing/Exercises/word_search.py` in your editor.
2. In the `main()` function, open `my_files/zen_of_python.txt` and create a list from its contents. Save the list as `zop`.
3. The rest of the `main()` function is already written. Read through the code to make sure you understand what it is doing.
4. Replace `pass` in the `search()` function with code that returns a two-element tuple containing the line number and line text in which the passed-in word is found. Return `None` if the word is not found. You may find it useful to review the `enumerate()` function (see page 196).
5. Run your solution by opening the terminal at `file-processing/Exercises` and running:

```
python word_search.py
```

Challenge

Modify the code so that it prints all the lines in which the word is found, like this:

```
Enter search word: than
than appears on line 5: Beautiful is better than ugly.
than appears on line 6: Explicit is better than implicit.
than appears on line 7: Simple is better than complex.
than appears on line 8: Complex is better than complicated.
than appears on line 9: Flat is better than nested.
than appears on line 10: Sparse is better than dense.
than appears on line 19: Now is better than never.
than appears on line 20: Although never is often better than *right* now.
```


Solution: file-processing/Solutions/word_search.py

```
1.  def search(word, text):
2.      """Return tuple holding line num and line text."""
3.      for line in enumerate(text, 1):
4.          if line[1].find(word) >= 0:
5.              return line
6.      return None
7.
8.  def main():
9.      with open('my_files/zen_of_python.txt') as f:
10.         zop = f.readlines()
11.
12.         word = input('Enter search word: ')
13.         result = search(word, zop)
14.         if result:
15.             print(word, ' first appears on line ',
16.                   result[0], ': ', result[1], sep='')
17.         else:
18.             print(word, 'was not found.')
19.
20.  main()
```

Challenge Solution: file-processing/Solutions/word_search_challenge.py

```
1. def search(word, text):
2.     """Return tuple holding line num and line text."""
3.     results = []
4.     for line in enumerate(text, 1):
5.         if line[1].find(word) >= 0:
6.             results.append(line)
7.     return results
8.
9. def main():
10.     with open('my_files/zen_of_python.txt') as f:
11.         zop = f.readlines()
12.
13.     word = input('Enter search word: ')
14.     results = search(word, zop)
15.     if not results:
16.         print(word, 'was not found.')
17.
18.     for result in results:
19.         print(word, ' appears on line ', result[0],
20.               ': ', result[1], sep='', end='')
21.
22. main()
```

Writing to Files

Files opened with a mode that permits writing can be written to using the `write()` method as shown in the following file:

Demo 74: file-processing/Demos/simple_write.py

```
1. with open('my_files/yoda.txt', 'w') as f:
2.     f.write('Powerful you have become.')
```

Code Explanation

1. Before running this file, look in the `file-processing/Demos/my_files` folder. It should **not** contain a `yoda.txt` file. If it does, delete it.
2. Now, run the following from `file-processing/Demos` at the terminal:

```
python simple_write.py
```

3. Now, look again in the `file-processing/Demos/my_files` folder. It should now contain a `yoda.txt` file. Open it in a text editor to see its contents.
-

Exercise 30: Writing to Files

 5 to 10 minutes

In this exercise, you will learn how modes affect reading and writing files. This exercise starts out like the previous demo.

1. Open `file-processing/Exercises/simple_write.py` in your editor.
2. Before running this file, look in the `file-processing/Exercises/my_files` folder. It should **not** contain a `yoda.txt` file. If it does, delete it.
3. Now, run the file from `file-processing/Exercises` at the terminal:

```
python simple_write.py
```

4. Now, look again in the `file-processing/Exercises/my_files` folder. The `yoda.txt` file should be there. Open it in a text editor to see its contents.
5. In `simple_write.py`, change the string passed to the `write()` method to “Pass on what you have learned” or whatever you like.
6. Run the file again. It should completely overwrite the file with the new string.
7. Now, change the mode from “w” to “a” and run it again. This time, it should append to the file.
8. Try adding a call to `print(f.read())` on the line after the call to `f.write()` and run the file again. You should get an error. Review the modes at the beginning of this lesson (see page 262) to see if you can figure out why and how to fix it.

Challenge

The `seek()` method is used to change the cursor position in the `file` object. Its most common purpose is to return to the beginning of the file by passing it `0`. If you were able to fix the issue causing the error when you added the call to `f.read()`, you may have been surprised to find that the content you wrote to the file didn’t get read. This is because, after writing, the cursor is at the end of the file. Use the `seek()` method to fix this so that the contents of the file are printed out.

Solution: file-processing/Solutions/simple_write.py

```
1. with open('my_files/yoda.txt', 'a+') as f:
2.     f.write('Powerful you have become.')
3.     print(f.read())
```

Code Explanation

By changing the mode to “a+”, the file is opened for appending and reading. However, when a file is opened for appending, the cursor is placed at the end of the file, and when you read from the file, it begins where the cursor is, which is why `print(f.read())` doesn’t output anything.

Challenge Solution: file-processing/Solutions/simple_write_challenge.py

```
1. with open('my_files/yoda.txt', 'a+') as f:
2.     f.write('Powerful you have become.')
3.     f.seek(0)
4.     print(f.read())
```

Code Explanation

Placing the cursor at the beginning of the file with `f.seek(0)` makes it possible to read the entire contents of the file.

Exercise 31: List Creator

 30 to 45 minutes

In this exercise, you will create a module that allows you to maintain a list of items in a file.

1. Open `file-processing/Exercises/list_creator.py` in your editor.
2. The `main()` and `show_instructions()` functions are already written. Your job is to write the following functions, which are documented in the file:
 - A. `add_item(item)`
 - B. `remove_item(item)` – This one is a little tricky. You will need to open the file twice, once for reading and once for writing. You may also find the `splitlines()` method (see page 132) of a string object useful.
 - C. `delete_list()`
 - D. `print_list()`
3. Now, run the file from `file-processing/Exercises` at the terminal:

```
PS ..\file-processing\Exercises> python list_creator.py
```

This is the starting code:

Exercise Code: file-processing/Exercises/list_creator.py

```
1. def add_item(item):
2.     """Appends item (after stripping leading and trailing
3.     whitespace) to list.txt followed by newline character
4.
5.     Keyword arguments:
6.     item -- the item to append"""
7.     pass
8.
9. def remove_item(item):
10.    """Removes first instance of item from list.txt
11.    If item is not found in list.txt, alerts user.
12.
13.    Keyword arguments:
14.    item -- the item to remove"""
15.    pass
16.
17. def delete_list():
18.    """Deletes the entire contents of the list by opening
19.    list.txt for writing."""
20.    pass
21.
22. def print_list():
23.    """Prints list"""
24.    pass
25.
26. def show_instructions():
27.    """Prints instructions"""
28.    print("""OPTIONS:
29.    P
30.        -- Print List
31.    +abc
32.        -- Add 'abc' to list
33.    -abc
34.        -- Remove 'abc' from list
35.    --all
36.        -- Delete entire list
37.    Q
38.        -- Quit\n""")
39.
40. def main():
41.    show_instructions()
42.
43.    while True:
44.        choice = input('>> ')
```

```
45.
46.     if choice.lower() == 'q':
47.         print('Goodbye!')
48.         break
49.
50.     if choice.lower() == 'p':
51.         print_list()
52.     elif choice.lower() == '--all':
53.         delete_list()
54.     elif choice and choice[0] == '+':
55.         add_item(choice[1:])
56.     elif choice and choice[0] == '-':
57.         remove_item(choice[1:])
58.     else:
59.         print("I didn't understand.")
60.         show_instructions()
61.
62. if __name__ == '__main__':
63.     main()
```

Challenge

1. The `delete_list()` function currently deletes the list without warning. Give the user a chance to confirm before deleting the list.
2. In the `print_list()` function, use the `enumerate()` function to print the items as a numbered list.

Solution: file-processing/Solutions/list_creator.py

```
1.  def add_item(item):
2.      """Appends item (after stripping leading and trailing
3.      whitespace) to list.txt followed by newline character
4.
5.      Keyword arguments:
6.      item -- the item to append"""
7.      with open('list.txt', 'a') as f:
8.          f.write(item.strip() + '\n')
9.
10. def remove_item(item):
11.     """Removes first instance of item from list.txt
12.     If item is not found in list.txt, alerts user.
13.
14.     Keyword arguments:
15.     item -- the item to remove"""
16.     item_found = False
17.     with open('list.txt', 'r') as f:
18.         items = f.read().splitlines()
19.         if item in items:
20.             items.remove(item)
21.             item_found = True
22.         else:
23.             print('"' + item + '" not found in list.')
24.
25.     if item_found:
26.         with open('list.txt', 'w') as f:
27.             f.write('\n'.join(items) + '\n')
28.
29. def delete_list():
30.     """Deletes the entire contents of the list by opening
31.     list.txt for writing."""
32.     with open('list.txt', 'w') as f:
33.         print('The list has been deleted.')
34.
35. def print_list():
36.     """Prints list"""
37.     with open('list.txt', 'r') as f:
38.         print(f.read())
39.
40. -----Lines 39 through 78 Omitted-----
```

Challenge Solution: file-processing/Solutions/list_creator_challenge.py

```
-----Lines 1 through 28 Omitted-----
29. def delete_list():
30.     """After confirming user really wants to delete list,
31.     deletes the entire contents of the list by opening
32.     list.txt for writing."""
33.     confirm = input('Are you sure you want to delete the list? y/n ')
34.     if confirm.lower() == 'y':
35.         with open('list.txt', 'w') as f:
36.             print('The list has been deleted.')
37.     else:
38.         print('OK. Whew! That was a close one.')
39.
40. def print_list():
41.     """Prints list"""
42.     with open('list.txt', 'r') as f:
43.         for i, item in enumerate(f, 1):
44.             print(i, '. ' + item, sep='', end='')
-----Lines 45 through 83 Omitted-----
```

The os Module

The `os` module is a built-in Python module for interacting with the operating system. In this section, we cover some of its most useful methods.

`os.getcwd()` and `os.chdir(path)`

`os.getcwd()` returns the current working directory as a string.

`os.chdir(path)` changes the current directory to `path`.

The following code shows how to get the current working directory with `os.getcwd()`, then change it using `os.chdir()`, and then get it again to prove that it changed:

```
>>> import os
>>> os.getcwd()
'C:\\Webucator\\Python\\file-processing\\Demos'

>>> os.chdir('..')
>>> os.getcwd()
'C:\\Webucator\\Python\\file-processing'
```

Notice the double backslash. That's because the string is escaped. Remember that the backslash is used to escape characters (see page 70). For a backslash to be just a backslash, it needs to be escaped by putting another backslash in front of it.

`os.listdir(path)`

Returns a list of files and subdirectories in `path`, which defaults to the current directory.

Demo 75: file-processing/Demos/directory_list.py

```
1. import os
2. dir_contents = os.listdir("my_files")
3. for item in dir_contents:
4.     print(item)
```

Code Explanation

This prints out a list of files and directories in the `my_files` directory:

```
PS ..\file-processing\Demos> python directory_list.py
a b c.txt
d e f.txt
foo.txt
g h i.txt
logo.png
zen_of_python.txt
```

`os.mkdir(dirname)` and `os.makedirs(path)`

`os.mkdir(dirname)` creates a directory named `dirname`. It errors if the directory already exists.

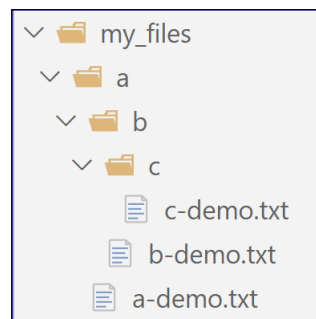
`os.makedirs(path)` is like `mkdir()`, but it creates all the necessary directories along the path.

Demo 76: file-processing/Demos/make_dirs.py

```
1. import os
2.
3. os.mkdir("my_files/a")
4. os.makedirs("my_files/a/b/c")
5.
6. with open('my_files/a/a-demo.txt', 'w') as f:
7.     f.write('Dummy text')
8. with open('my_files/a/b/b-demo.txt', 'w') as f:
9.     f.write('Dummy text')
```

Code Explanation

This creates directory `a` using `mkdir()` and then makes directories `b` and `c` using `makedirs()`. It then writes text files to the `a` and `b` directories. Run `make_dirs.py` from the `file-processing/Demos` directory and then look to see that the directories were created:



`os.rmdir(path)` and `os.removedirs(path)`

`os.rmdir(path)` deletes the directory at `path`. It errors if the directory is not empty or doesn't exist.

`os.removedirs(path)` is like `rmdir()`, but removes all empty directories in `path`, starting with the *leaf* (the last directory in the path). It will error if the leaf directory is not empty.

1. Open the Python terminal at `file-processing/Demos`.

2. Run `import os`.
3. Run `os.rmdir('my_files/a/b/c')` to remove directory `c`.
4. Look in your file system to see that the directory was removed.
5. Run `os.removedirs('my_files/a/b')` to try to remove both the `a` and `b` directories. You will get an error because directory `b` is not empty.

`os.remove(path)` and `os.unlink(path)`

`os.remove(path)` and `os.unlink(path)` are identical. They delete the file at `path`, and they error if `path` doesn't point to a file.

1. Open the Python terminal at `file-processing/Demos` if it's not already open.
2. Run `import os`.
3. Run `os.listdir('my_files/a/b')` to show that directory `b` contains a file:

```
>>> import os
>>> os.listdir('my_files/a/b')
['b-demo.txt']
```

4. Run `os.remove('my_files/a/b/b-demo.txt')` to remove the file in directory `b`, and then run `os.listdir('my_files/a/b')` again to show that directory `b` is now empty:

```
>>> os.remove('my_files/a/b/b-demo.txt')
>>> os.listdir('my_files/a/b')
[]
```

5. Now, remove the file from directory `a`:

```
>>> os.remove('my_files/a/a-demo.txt')
```


- Now that both directories `a` and `b` are empty, you can run `os.removedirs('my_files/a/b')` to remove both directories, and then use `os.listdir('my_files')` to check that they have been removed:

```
>>> os.remove('my_files/a/a-demo.txt')
>>> os.removedirs('my_files/a/b')
>>> os.listdir('my_files')
['a b c.txt', 'd e f.txt', 'foo.txt', 'g h i.txt', 'logo.png',
'zen_of_python.txt']
```

- Close the Python terminal.

`os.rename(source, destination)` and `os.rename(oldpath, newpath)`

`os.rename(source, destination)` renames a file or a directory, changing the name from `source` to `destination`.

`os.rename(oldpath, newpath)` is like `rename(source, destination)`, but it starts by creating all necessary directories in `newpath` and ends by removing all empty directories in `oldpath`.

Demo 77: file-processing/Demos/renames.py

```
1. import os
2.
3. foo = 'my_files/foo.txt'
4. bar = 'my_new_files/bar.txt'
5.
6. def foo2bar():
7.     os.rename(foo, bar)
8.     print('Renamed', foo, 'to', bar)
9.
10. def bar2foo():
11.     os.rename(bar, foo)
12.     print('Renamed', bar, 'to', foo)
13.
14. foo2bar()
```

Code Explanation

- Before running this file, look in the `file-processing/Demos/my_files` folder. It should contain a file named `foo.txt`.

2. Look in the `file-processing/Demos` folder. It should **not** contain a `my_new_files` folder. If it does, delete it.
3. Now, run `renames.py` from the `file-processing/Demos` directory:

```
PS ~\file-processing\Demos> python renames.py
Renamed my_files/foo.txt to my_new_files/bar.txt
```

4. Look again in the `file-processing/Demos` folder. A `my_new_files` folder should have appeared. Open it to find a `bar.txt` file, which is the `foo.txt` renamed and moved.
5. Look again in the `file-processing/Demos/my_files` folder. The `foo.txt` file should be gone. The `my_files` folder remains, because it has other things in it.
6. Open `file-processing/Demos/renames.py` in your editor. Change it so that it calls `bar2foo()` instead of `foo2bar()`. Run the file again. The results:
 - A. `file-processing/Demos/my_files/foo.txt` returns.
 - B. `file-processing/Demos/my_new_files/bar.txt` gets deleted.
 - C. `file-processing/Demos/my_new_files` also gets deleted as the only file it contained was `bar.txt`.

Walking a Directory

`os.walk(top, topdown=True, onerror=None, followlinks=False)`

The `os.walk()` method returns a generator by walking the directory tree and yielding a three-element tuple for each directory containing:

- `dirpath` – The path to the directory as a string.
- `dirnames` – A list of subdirectories in `dirpath`.
- `filenames` – A list of files in `dirpath`.

`os.walk()` Parameters

- `top` – The top-level directory for the walk.

- **topdown** – By default, `os.walk()` starts with **top** and works its way down; however, if **topdown** is `False`, it will work its way from bottom to top.
- **onerror** – By default, errors are ignored. If you want to handle errors in some way, set **onerror** to a function. When an error occurs, that function will be called and passed an `OSError` instance.
- **followlinks** – By default, `os.walk()` ignores symbolic links (files that link to other directories). Set **followlinks** to `True` to include those linked directories. Be careful: you could end up with an infinite loop!

Let's take a look at an example. Review the following code to see if you can understand what it does:

Demo 78: file-processing/Demos/walk.py

```
1.  import os
2.
3.  def spaces2dashes():
4.      for dirpath, dirnames, filenames in os.walk('my_files'):
5.          for fname in filenames:
6.              if ' ' in fname:
7.                  oldname = dirpath + '/' + fname
8.                  newname = dirpath + '/' + fname.replace(' ', '-')
9.                  print("Replacing", oldname, "with", newname)
10.                 os.rename(oldname, newname)
11.
12.  def dashes2spaces():
13.      for dirpath, dirnames, filenames in os.walk('my_files'):
14.          for fname in filenames:
15.              if '-' in fname:
16.                  oldname = dirpath + '/' + fname
17.                  newname = dirpath + '/' + fname.replace('-', ' ')
18.                  print("Replacing", oldname, "with", newname)
19.                  os.rename(oldname, newname)
20.
21.  spaces2dashes()
```

Code Explanation

1. Before running this file, look in the `file-processing/Demos/my_files` folder. It should contain some files with spaces in their names (e.g., `a b c.txt`). That's ugly!
2. Now, run `file-processing/Demos/walk.py`.

3. Now, look again in the `file-processing/Demos/my_files` folder. The spaces in those file names should have been replaced with dashes.
 4. If you like, edit the file so that it runs `dashes2spaces()` in place of `spaces2dashes()` and run it again to put the spaces back in the file names.
-

The `os.path` Module

The `os.path` module is used for performing functions on path names. In this section, we cover some of its most useful methods:

`os.path.abspath(path)`

`os.path.abspath(path)` returns the absolute path of `path`.

```
>>> os.path.abspath('.')
'C:\\Webucator\\Python\\file-processing\\Demos'
```

`os.path.basename(path)`

`os.path.basename(path)` returns the basename of `path`.

```
>>> os.path.basename('my_files/logo.png')
'logo.png'
```

`os.path.dirname(path)`

`os.path.dirname(path)` returns the path to the directory containing the leaf element of `path`.

```
>>> os.path.dirname('my_files/logo.png')
'my_files'
```

`os.path.exists(path)`

`os.path.exists(path)` returns `True` if `path` exists.

```
>>> os.path.exists('my_files/logo.png')
True
```

`os.path.getatime(path)`, `os.path.getmtime(path)`, and `os.path.getctime(path)`

`os.path.getatime(path)` returns the time path was last **a**ccessed in number of seconds from the epoch.

`os.path.getmtime(path)` returns the time path was last **m**odified in number of seconds from the epoch.

`os.path.getctime(path)` returns the time path was **c**reated in number of seconds from the epoch.

We can use the `datetime` module to convert and format these as readable dates and times:

```
>>> last_accessed = os.path.getatime('my_files/logo.png')
>>> last_modified = os.path.getmtime('my_files/logo.png')
>>> date_created = os.path.getctime('my_files/logo.png')
>>> last_accessed, last_modified, date_created
(1582050514.8227544, 1581523336.466663, 1580745606.866663)
>>> from datetime import datetime
>>> datetime.fromtimestamp(last_accessed).strftime('%c')
'Tue Feb 18 13:28:34 2020'
>>> datetime.fromtimestamp(last_modified).strftime('%c')
'Wed Feb 12 11:02:16 2020'
>>> datetime.fromtimestamp(date_created).strftime('%c')
'Mon Feb  3 11:00:06 2020'
```

`os.path.getsize(path)`

`os.path.getsize(path)` returns the size of the file at path.

```
>>> os.path.getsize('my_files/logo.png')
26216
```

`os.path.isabs(path)`

`os.path.isabs(path)` returns True if path is an absolute path.

```
>>> os.path.isabs('my_files/logo.png')
False
```

`os.path.relpath(path, start)`

`os.path.relpath(path, start)` returns a relative path to `path` from `start`, which defaults to the current directory.

```
>>> os.path.relpath(r'C:\Webucator\Python\file-processing\Demos\my_files\logo.png')
'my_files\\logo.png'
```

If you are confused by the “r” prefix, review Raw Strings (see page 71).

`os.path.isfile(path)` and `os.path.isdir(path)`

`os.path.isfile(path)` returns True if `path` is a file.

```
>>> os.path.isfile('my_files/logo.png')
True
```

`os.path.isdir(path)` returns True if `path` is a directory.

```
>>> os.path.isdir('my_files/logo.png')
False
```

`os.path.join(path, *paths)`

`os.path.join(path, *paths)` returns a path by joining the passed-in paths intelligently. Note that different operating systems use different path structures. This will create the correct structure for the operating system it runs on.

```
>>> os.path.join('my_files', 'logo.png')
'my_files\\logo.png'
```

`os.path.split(path)`

Returns a 2-element tuple containing `os.path.dirname(path)` and `os.path.basename(path)`.

```
>>> os.path.split('my_files/logo.png')
('my_files', 'logo.png')
```

`os.path.splitext(path)`

Returns a 2-element tuple containing *the path minus the extension* and *the extension*.

```
>>> os.path.splitext('my_files/logo.png')
('my_files/logo', '.png')
```

A Better Way to Open Files

Let's take another look at `Demos/with.py`:

Demo 79: file-processing/Demos/with.py

```
1.  with open('my_files/zen_of_python.txt') as f:
2.      poem = f.read()
3.
4.  # The file is now closed, but we saved its content in the poem variable
5.  for i, line in enumerate(poem.splitlines(), 1):
6.      print(f"{i}. {line}")
```

We previously ran that file from the `file-processing/Demos` directory and it worked just fine, but try running it from a different directory:

1. Open `file-processing` in the Python terminal.

2. Run python Demos/with.py:

```
PS ..\Python\file-processing> python Demos/with.py
Traceback (most recent call last):
  File "Demos/with.py", line 1, in <module>
    with open('my_files/zen_of_python.txt') as f:
FileNotFoundError: [Errno 2] No such file or directory:
'my_files/zen_of_python.txt'
```

The issue is that Python is not looking for the file relative to where the Python file is; it is looking for it relative to where the Python script is being executed. One solution is to turn the relative path into an absolute path. To do that, we need the special `__file__` variable (that's two underscores on each side of "file").

`__file__` is a special variable that holds a relative path to the Python script from the present working directory. To see how it works, run `file-processing/Demos/special_file_variable.py`, which simply runs `print(__file__)`, from different directories:

```
PS ..\file-processing\Demos> python special_file_variable.py
special_file_variable.py

PS ..\file-processing\Demos> cd ..
PS ..\Python\file-processing> python Demos/special_file_variable.py
Demos/special_file_variable.py

PS ..\Python\file-processing> cd ..
PS ..\Webucator\Python> python file-processing/Demos/special_file_variable.py
file-processing/Demos/special_file_variable.py

PS ..\Webucator\Python> cd file-processing/Exercises
PS ..\file-processing\Exercises> python ../Demos/special_file_variable.py
../Demos/special_file_variable.py
```

We can take the absolute path to the folder holding the Python script and combine it with the parts of the relative path to the file we want to open to create an absolute path, like this:


```
relative_path = "my_files/zen_of_python.txt"

# Get absolute path to Python script
abs_path = os.path.abspath(__file__)

# Get absolute path to the directory that the script is in
folder = os.path.dirname(abs_path)

# Split the relative path on the forward slash
path_parts = relative_path.split('/')

# Join dir with path_parts to get an absolute path to the file to open.
new_path = os.path.join(folder, *path_parts)
```

The following script includes `print()` calls showing the values of the different variables:

Demo 80: file-processing/Demos/relpath_to_abspath.py

```
1.  import os
2.
3.  relative_path = "my_files/zen_of_python.txt"
4.  print("__file__:", __file__)
5.
6.  # Get absolute path to Python script
7.  abs_path = os.path.abspath(__file__)
8.  print("abs_path:", abs_path)
9.
10. # Get absolute path to the directory that the script is in
11. folder = os.path.dirname(abs_path)
12. print("dir:", folder)
13.
14. # Split the relative path on the forward slash
15. path_parts = relative_path.split("/")
16. print("path_parts:", path_parts)
17.
18. # Join dir with path_parts to get an absolute path to the file to open.
19. new_path = os.path.join(folder, *path_parts)
20. print("new_path:", new_path)
```

Code Explanation

This will output something like:

```
PS ..\file-processing\Demos> python relpath_to_abspath.py
__file__: relpath_to_abspath.py
abs_path: C:\Webucator\Python\file-processing\Demos\relpath_to_abspath.py
dir: C:\Webucator\Python\file-processing\Demos
path_parts: ['my_files', 'zen_of_python.txt']
new_path: C:\Webucator\Python\file-processing\
          Demos\my_files\zen_of_python.txt
```

For reuse purpose, we can create a function that takes a relative path to the file and returns an absolute path to the same file:

```
def file_path(relative_path):
    folder = os.path.dirname(os.path.abspath(__file__))
    path_parts = relative_path.split('/')
    new_path = os.path.join(folder, *path_parts)
    return new_path
```

Finally, we use that function in `file-processing/Demos/with2.py` so that we can run the script from anywhere:

Demo 81: file-processing/Demos/with2.py

```
1.  import os
2.  def file_path(relative_path):
3.      folder = os.path.dirname(os.path.abspath(__file__))
4.      path_parts = relative_path.split("/")
5.      new_path = os.path.join(folder, *path_parts)
6.      return new_path
7.
8.  path_to_file = file_path("my_files/zen_of_python.txt")
9.  with open(path_to_file) as f:
10.      poem = f.read()
11.
12.  for i, line in enumerate(poem.splitlines(), 1):
13.      print(f"{i}. {line}")
```

Exercise 32: Comparing Lists

 45 to 90 minutes

An article in the Atlantic³⁶ claims that parents give girls boys names, but won't give boys girls names. Journalists need data to make this kind of claim. They rely on data analysts, many of whom are Python programmers like you, to analyze the data.

In the `file-processing/data` directory, there are files that contain lists of the most popular names in 1880 and 2018:

1. `1880-boys.txt`
2. `1880-girls.txt`
3. `2018-boys.txt`
4. `2018-girls.txt`

The data in these files is from <https://www.ssa.gov/oact/babynames/>.

Using the data in these files, try to answer these questions:

1. What names that were exclusively boys names in 1880 became exclusively girls names in 2018?
2. What names that were exclusively girls names in 1880 became exclusively boys names in 2018?

Does what you found support the journalist's claim?

Spend as much time as you like on this. It is an opportunity to practice your new Python skills.

³⁶. <https://www.theatlantic.com/family/archive/2018/09/girls-names-for-baby-boys/569962/>

Solution: file-processing/Solutions/boys_and_girls.py

```
1.  import os
2.
3.  def file_path(relative_path):
4.      """Returns absolute path from relative path"""
5.      folder = os.path.dirname(os.path.abspath(__file__))
6.      path_parts = relative_path.split("/")
7.      new_path = os.path.join(folder, *path_parts)
8.      return new_path
9.
10. def file_to_list(path):
11.     """Returns content of file at path as list"""
12.     with open(file_path(path)) as f:
13.         lines = f.read().splitlines()
14.     return lines
15.
16. def subtract_lists(list1, list2):
17.     """Returns a list of all items in list1, but not in list2"""
18.     return [x for x in list1 if x not in list2]
19.
20. def dups(list1, list2, sort=True):
21.     """Returns a list containing items in both lists"""
22.     dup_list = []
23.     for item in list1:
24.         if item in list2:
25.             dup_list.append(item)
26.
27.     if sort:
28.         dup_list.sort()
29.
30.     return dup_list
31.
32. def list_to_file(path, the_list):
33.     """Creates/Overwrites file at path with content from the_list"""
34.     content = "\n".join(the_list)
35.
36.     with open(file_path(path), "w") as f:
37.         f.write(content)
38.
39.
40. def main():
41.     boys_2018_path = "../data/2018-boys.txt"
42.     girls_2018_path = "../data/2018-girls.txt"
43.     boys_1880_path = "../data/1880-boys.txt"
44.     girls_1880_path = "../data/1880-girls.txt"
```

```
45.
46.     boys_2018 = file_to_list(boys_2018_path)
47.     girls_2018 = file_to_list(girls_2018_path)
48.     boys_1880 = file_to_list(boys_1880_path)
49.     girls_1880 = file_to_list(girls_1880_path)
50.
51.     boys_only_2018 = subtract_lists(boys_2018, girls_2018)
52.     girls_only_2018 = subtract_lists(girls_2018, boys_2018)
53.     boys_only_1880 = subtract_lists(boys_1880, girls_1880)
54.     girls_only_1880 = subtract_lists(girls_1880, boys_1880)
55.
56.     boys_names_2_girls_names = dups(girls_only_2018, boys_only_1880)
57.     girls_names_2_boys_names = dups(boys_only_2018, girls_only_1880)
58.
59.     list_to_file("../data/girls-names-that-were-boys-names.txt",
60.                  boys_names_2_girls_names)
61.     list_to_file("../data/boys-names-that-were-girls-names.txt",
62.                  girls_names_2_boys_names)
63.
64.     main()
```

Code Explanation

There are different ways of going about this. We went through the following process:

1. Create a `file_to_list()` function that takes a file path, reads it, and returns a list. We use this function to create four lists from the four files.
2. Find all the exclusively boys names and all the exclusively girls names in 1880 and in 2018. We did this by creating a `subtract_lists()` function that gets all the items from one list that are not in another list.
3. Create a `dup()` function that takes two lists and returns a new list of all the items that are in both lists. We use this to compare exclusively girls names in 2018 with exclusively boys names in 1880. That tells us what names that used to be strictly boys names are now strictly girls names. We then do the same thing to compare exclusively boys names in 2018 with exclusively girls names in 1880.
4. We then write a `list_to_file()` function and use it to save the results in these new files:
 - A. `girls-names-that-were-boys-names.txt` – We found 16:
 - i. Addison
 - ii. Allison
 - iii. Ashley

- iv. Aubrey
- v. Bailey
- vi. Charley
- vii. Dana
- viii. Holly
- ix. Ivory
- x. Kelly
- xi. Lindsey
- xii. Madison
- xiii. Monroe
- xiv. Palmer
- xv. Shelby
- xvi. Sydney

B. `boys-names-that-were-girls-names.txt` – We didn't find any.
These results appear to support the journalist's claim.

Conclusion

In this lesson, you have learned to work with files and directories on the operating system.

LESSON 11

PEP8 and Pylint

Topics Covered

☑ PEP8.

☑ Pylint.

[H]is personality has suggested to me an entirely new manner in art, an entirely new mode of style. I see things differently, I think of them differently. I can now recreate life in a way that was hidden from me before.

– *The Picture of Dorian Gray*, Oscar Wilde

Introduction

PEP8³⁷ is the official style guide for Python code. Pylint is software that helps you follow the PEP8 guidelines and find potential problems with your code. In this lesson, you will learn about both.

PEP8

Here we provide a summary of PEP8's primary recommendations. We do not cover everything. We recommend that you read PEP8 yourself (see <https://www.python.org/dev/peps/pep-0008/>). Be aware that you may not understand everything as some of it deals with advanced Python functionality that we have not covered.

Maximum Line Length

Limit your lines to 79 characters whenever possible. For flowing lines (e.g., a long text string), limit lines to 72 characters.

37. <https://www.python.org/dev/peps/pep-0008/>

Indentation

Use 4 spaces (not tabs) per indentation level.

Continuation Lines

When you must wrap a line of code across lines to adhere to the maximum line length, you should either:

1. Align the wrapped elements vertically:

Good

```
moonwalk = datetime.datetime(year=1969, month=7, day=21,
                              hour=2, minute=56, second=15,
                              tzinfo=datetime.timezone.utc)
```

2. Use a hanging indent:

Good

```
moonwalk = datetime.datetime(
    year=1969, month=7, day=21,
    hour=2, minute=56, second=15,
    tzinfo=datetime.timezone.utc)
```

Note that, when using a hanging indent, no arguments should be on the first line:

Bad

```
moonwalk = datetime.datetime(year=1969, month=7, day=21,
                              hour=2, minute=56, second=15,
                              tzinfo=datetime.timezone.utc)
```

3. Long sequences that must wrap should be structured with the closing bracket aligned with the first line:

Good

```
fruit = [
    "apple", "orange", "banana", "pear",
    "lemon", "watermelon", "strawberry"
]
```


Or with the closing bracket aligned with the last line:

Good

```
fruit = [  
    "apple", "orange", "banana", "pear",  
    "lemon", "watermelon", "strawberry"  
]
```

Blank Lines

1. Separate function definitions with two blank lines:³⁸

Good

```
def do_this():  
    pass  
  
def do_that():  
    pass  
  
def do_this_other_thing():  
    pass  
  
# More Code
```

2. Blank lines can be used to separate logical sections within functions.

UTF-8 Encoding

Python files should be encoded using UTF-8. If you're using Visual Studio Code, the encoding is indicated in the bottom right:



Imports

1. All imports should be at the top of the file.

^{38.} In our class files, we use only a single blank line to separate functions to keep files shorter for printing purposes.

2. Import standard library imports before third-party imports.
3. Import third-party imports before local imports.
4. Import full packages before importing parts of packages:

Good

```
import math
from datetime import date, time
```

Bad

```
from datetime import date, time
import math
```

5. Imports should use separate lines:

Good

```
import math
import random
```

Bad

```
import math, random
```

Quotes

The only recommendations on single vs. double quotes are:

1. Use double quotes when string contains a single quote character (i.e., an apostrophe).
2. Use single quotes when string contains a double quote character.
3. Use double quote characters for triple-quoted strings.
4. Be consistent with yourself.

Whitespace in Expressions and Statements

Do not add whitespace...

1. Inside parentheses, brackets, or curly braces:

Good

```
if (a == b):
```

Bad

```
if ( a == b ):
```

Good

```
all([a, b, c, d])
```

Bad

```
all( [a, b, c, d] )
```

2. Before a comma:

Good

```
all([a, b, c, d])
```

Bad

```
all([a , b , c, d])
```

3. Before an open parentheses in a function call:

Good

```
print('Hello, world!')
```

Bad

```
print ('Hello, world!')
```

4. Around the equals (=) sign in keyword arguments:

Good

```
def dups(list1, list2, sort=True):
```

Bad

```
def dups(list1, list2, sort = True):
```

Surround assignment and comparison operators with single spaces:

Good

```
greeting = 'Hello'
```

Bad

```
greeting='Hello'
```

Good

```
a += 1
```

Bad

```
a+=1
```

Good

```
if (a == 1):
```

Bad

```
if (a==1):
```

Do not include extra whitespace at the end of a line or on blank lines.

Comments

Comments should use complete sentences and start with capital letters (unless they start with a keyword written in lowercase).

❖ Additional Content in PEP8

We've covered what we feel are the most important PEP8 recommendations, but there are more. Again, we recommend that you read PEP8 and return to it from time to time, especially when you're unsure of the appropriate way to style a piece of code.

Pylint

Linters are tools that review your code and report stylistic or programming errors. The most popular Python linter is Pylint. To install Pylint, run the following command in the terminal:

```
pip install pylint
```

You can then use Pylint to analyze a file by running:

```
pylint path_to_file
```

For example, open the terminal at `date-time/Exercises` and run:

```
pylint departure_report.py
```

To save the results in a file, use:

```
pylint path_to_file > filename
```

For example:

```
pylint departure_report.py > pylint.txt
```

When we run Pylint on `departure_report.py` in `date-time/Solutions`, we get this result:

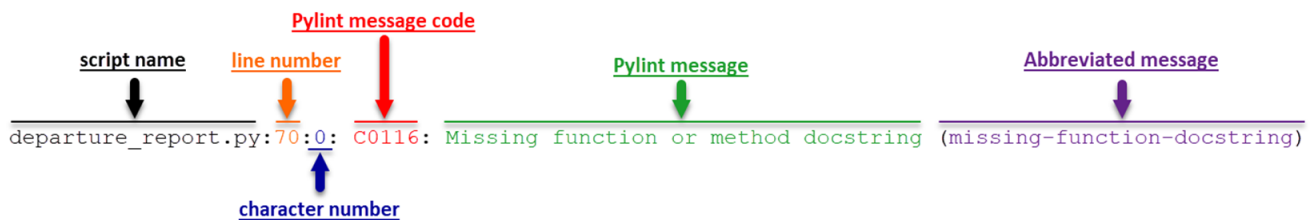
```

***** Module departure_report
departure_report.py:85:0: C0304: Final newline missing (missing-final-newline)
departure_report.py:1:0: C0114: Missing module docstring (missing-module-docstring)
departure_report.py:6:47: C0103: Variable name "f" doesn't conform to snake_case naming
style (invalid-name)
departure_report.py:70:0: C0116: Missing function or method docstring (missing-function-
docstring)

```

Your code has been rated at 9.30/10 (previous run: 8.25/10, +1.05)

For each line that Pylint deems problematic, it provides the script name, the line number, the character number, the Pylint message code, the Pylint message, and the abbreviated message, as shown in the following image:



The report ends with a rating based on a scale of 10.

It is important to note that Pylint has opinions that go beyond PEP8, and that not all Python developers agree with all of its opinions. For example, we don't follow the convention of adding a newline at the end of every Python script. In our view, this was a historically useful practice for technical reasons that are no longer relevant with today's programming tools. We also use certain one-letter variable names, such as `i` for integer, `f` for file, and `k` and `v` for key and value. However, we only use them in small blocks of code.

Missing Docstrings

According to PEP8, every module and function should be commented. While we agree with this, we have not followed the practice in these lessons, because the accompanying readings explain how the modules and functions work. In many cases, the extra documentation would distract from the specific functionality being introduced.

linter.py

We have included a `linter.py` script in the root folder of your class files. This will run Pylint on all the Python files in a directory and its subdirectories. Run the file from the directory it is in like this:

```
python linter.py path_to_folder output_file
```

For example:

```
python linter.py date-time/Exercises > pylint.txt
```

After running this, you will find a `pylint.txt` file in the root directory containing a Pylint report on every Python file in the `date-time/Exercises` directory. You can then make fixes and run it again to generate a new report.

The `pylintrc` file, which is also in the root of your class files, has the settings used by `linter.py`. We have added a few modifications to the default settings:

1. Ignore the following rules:
 - `duplicate-code`
 - `missing-final-newline`
 - `missing-function-docstring`
 - `missing-module-docstring`
 - `invalid-name`
2. Allow variables named `foo` and `bar`. By default, Pylint disallows those variables and several others.

Feel free to explore the `pylintrc` file to see what other modifications you can make.

Conclusion

In this lesson, you have learned about the PEP8 coding standards and how to use Pylint to analyze your code.