

Introduction to Python

(PYT112 version 1.0.0)

Copyright Information

© Copyright 2018 Webucator. All rights reserved.

Accompanying Class Files

This manual comes with accompanying class files, which your instructor or sales representative will point out to you. Most code samples and exercise and solution files found in the manual can also be found in the class files at the locations indicated at the top of the code listings.

Due to space limitations, the code listings sometimes have line wrapping, where no line wrapping occurs in the actual code sample. This is indicated in the manual using three greater than signs: >>> at the beginning of each wrapped line.

In other cases, the space limitations are such that we have inserted a forced line break in the middle of a word. When this occurs, we append the following symbol at the end of the line before the actual break: »»

Table of Contents

1. Python Basics.....	1
Running Python.....	1
Visual Studio Code.....	2
Python Interpreter in Interactive Mode.....	2
Commercial and Free Python IDEs.....	3
IDLE.....	3
Hello, world!.....	4
<i>Exercise 1: Hello World.</i>	5
Literals.....	7
Python Comments.....	7
Multi-line Comments.....	7
Data Types.....	9
<i>Exercise 2: Exploring Types.</i>	10
Variables.....	13
Variable Names.....	13
Variable Assignment.....	14
Simultaneous Assignment.....	15
<i>Exercise 3: A Simple Python Script.</i>	18
Constants.....	21
Deleting Variables.....	21
Writing a Python Module.....	21
The main() Function.....	21
print() Function.....	24
Named Arguments.....	25
Collecting User Input.....	26
<i>Exercise 4: Hello, You!</i>	28
Getting Help.....	31
2. Functions and Modules.....	35
Defining Functions.....	35
Variable Scope.....	37
Global Variables.....	38
Function Parameters.....	41
Using Parameter Names in Function Calls.....	43
<i>Exercise 5: A Function with Parameters.</i>	45
Default Values.....	51
<i>Exercise 6: Parameters with Default Values.</i>	52
Returning Values.....	55
Importing Modules.....	56
Module Search Path.....	57
pyc Files.....	57

Table of Contents

3. Math.....	59
Arithmetic Operators.....	59
Modulus and Floor Division.....	59
<i>Exercise 7: Floor and Modulus.....</i>	61
Assignment Operators.....	63
Precedence of Operations.....	63
Built-in Math Functions.....	64
int(x).....	64
eval(str).....	64
float(x).....	65
abs(x).....	65
min(args) and max(args).....	65
pow(x,y[,z]).....	66
round(number[, n]).....	66
sum(iter[, start]).....	67
The math Module.....	67
Additional math Functions.....	68
The random Module.....	69
Seeding.....	70
<i>Exercise 8: How Many Pizzas Do We Need?.....</i>	72
4. Python Strings.....	77
Quotation Marks and Special Characters.....	77
Escaping Characters.....	77
Triple Quotes.....	78
String Indexing.....	81
<i>Exercise 9: Indexing Strings.....</i>	83
Slicing Strings.....	85
<i>Exercise 10: Slicing Strings.....</i>	87
Concatenation and Repetition.....	89
Concatenation.....	89
Repetition.....	89
<i>Exercise 11: Repetition.....</i>	90
Combining Concatenation and Repetition.....	93
Common String Methods.....	93
String Formatting.....	97
Format Specification.....	98
Long Lines of Code.....	105
<i>Exercise 12: Playing with Formatting.....</i>	107
Formatted String Literals (f-strings).....	108
<i>Exercise 13: Getting Acquainted with f-strings.....</i>	109
Built-in String Functions.....	111
str(object).....	111
len(string).....	111
min() and max().....	111
<i>Exercise 14: Outputting Tab-delimited Text.....</i>	112

5. Iterables: Sequences, Dictionaries, and Sets.....	117
Definitions.....	117
Sequences.....	117
Lists.....	117
Deleting List Elements.....	118
Sequences and Random.....	119
<i>Exercise 15: Remove and Return Random Element.....</i>	120
Tuples.....	123
The Immutability of Tuples.....	125
Lists vs. Tuples.....	126
Ranges.....	127
Converting Sequences to Lists.....	128
Indexing and Slicing.....	129
<i>Exercise 16: Simple Rock, Paper, Scissors Game.....</i>	131
<i>Exercise 17: Slicing Sequences.....</i>	135
min(iter) and max(iter).....	137
sum(iter[, start]).....	137
Converting Sequences to Strings with str.join(seq).....	138
Splitting Strings into Lists.....	138
Unpacking Sequences.....	139
Dictionaries.....	139
The update() Method.....	140
The setdefault() Method.....	141
Dictionary View Objects.....	142
Deleting Dictionary Keys.....	143
The len() Function.....	144
<i>Exercise 18: Creating a Dictionary from User Input.....</i>	145
Sets.....	148
*args and **kwargs.....	148
Using *args.....	149
Using **kwargs.....	150

Table of Contents

6. Flow Control.....	151
Conditional Statements.....	151
The is and is not Operators.....	154
all() and any().....	156
Ternary Operator.....	156
Loops in Python.....	157
while Loops.....	157
for Loops.....	159
<i>Exercise 19: All True and Any True.....</i>	163
break and continue.....	165
<i>Exercise 20: Word Guessing Game.....</i>	166
The else Clause.....	174
<i>Exercise 21: Find the Needle.....</i>	178
The enumerate() Function.....	183
Generators.....	184
Generator Use Case: Randomly Moving Object.....	187
The next() Function.....	188
<i>Exercise 22: Rolling Dice.....</i>	191
List Comprehensions.....	193
7. Virtual Environments.....	195
Virtual Environment.....	195
Creating a Virtual Environment.....	195
Activating and Deactivating a Virtual Environment.....	196
Deleting a Virtual Environment.....	197
<i>Exercise 23: Working with a Virtual Environment.....</i>	198
8. Regular Expressions.....	201
Regular Expression Syntax.....	201
Try it.....	201
Backreferences.....	205
Python's Handling of Regular Expressions.....	206
9. Unicode and Encoding.....	213
Bits and Bytes.....	213
Hexadecimal Numbers.....	214
<i>Exercise 24: Converting Numbers between Number Systems.....</i>	215
hex(), bin(), ord(), chr(), and int().....	219
Encoding.....	219
Encoding Text.....	219
Encoding and Decoding Files in Python.....	220
Converting a File from cp1252 to UTF-8.....	222
<i>Exercise 25: Finding Confusables.....</i>	224

10. File Processing.....	229
Opening Files.....	229
Methods of File Objects.....	230
<i>Exercise 26: Finding Text in a File</i>	234
<i>Exercise 27: Writing to Files</i>	239
<i>Exercise 28: List Creator</i>	241
The os and os.path Modules.....	246
os.....	246
os.path.....	249
11. Exception Handling.....	253
Wildcard except Clauses.....	254
Getting Information on Exceptions.....	255
<i>Exercise 29: Raising Exceptions</i>	256
The else Clause.....	261
The finally Clause.....	262
Using Exceptions for Flow Control.....	262
<i>Exercise 30: Running Sum</i>	263
Raising Your Own Exceptions.....	265
Exception Hierarchy.....	266
12. Python Dates and Times.....	269
The time Module.....	271
Clocks.....	271
Time Structures.....	276
Times as Strings.....	278
Time and Formatted Strings.....	278
time.sleep(secs).....	281
The datetime Module.....	281
datetime.date objects.....	282
datetime.time objects.....	283
datetime.datetime Objects.....	284
datetime.timedelta objects.....	286
<i>Exercise 31: Report on Amtrak Departure Times</i>	288
13. Running Python Scripts from the Command Line.....	295
sys.argv.....	295
A More Useful Example.....	295
sys.path.....	297

1. Python Basics

In this lesson, you will learn...

1. How Python works.
2. About Python's place in the world of programming languages.
3. About the difference between Python 3.x and Python 2.x.
4. About variables and Python's data types.
5. To create simple modules.
6. To get help on Python.

Python, which first appeared in 1991, is one of the most [popular programming languages¹](#) in use. 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 friendl .

1.1 Running Python

Python runs on Microsoft Windows, Mac OS X, Linux, and other Unix-like systems.

To run Python code, you will need the Python Interpreter. If you are using a Mac or Linux machine, you most likely already have a Python Interpreter installed.

If you are using Windows, you can download Python for free at <https://www.python.org/downloads>.

Checking Your Python Version

To see what version of Python you have, open a Terminal window or Command Prompt and type:

```
python -V
```

1. See <http://pvl.github.io/PYPL.html>.

Except when specifically pointed out, the class files in this course will run on Python 3.3 or later. Differences between Python 2 and Python 3 are pointed out throughout the course.

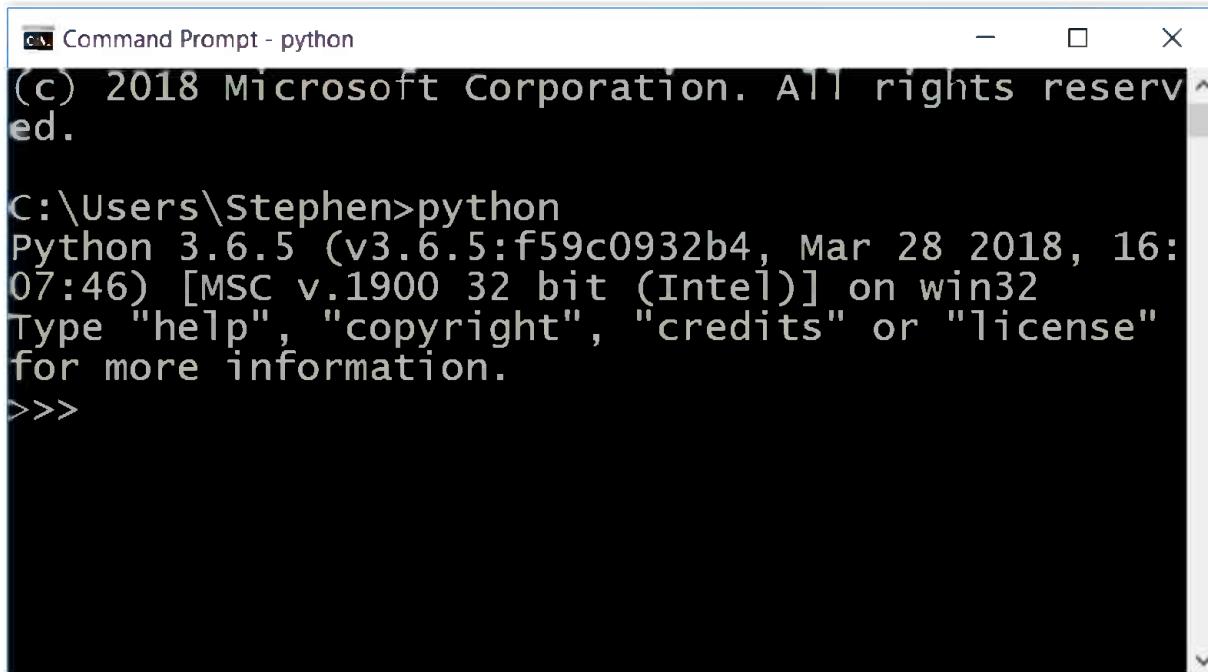
You can program Python in many different environments:

Visual Studio Code

With the Python extension installed, **Visual Studio Code (VS Code)** is a convenient Python development platform. You can download the Python extension for VS Code at <https://marketplace.visualstudio.com/items?itemName=ms-python.python>.

Python Interpreter in Interactive Mode

Open the Python Interpreter by typing "python" (without the quotes) at a Terminal window or Command Prompt:



```
Command Prompt - python
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Stephen>python
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license"
for more information.

>>>
```

Commercial and Free Python IDEs

There are many commercial and free Python IDEs available for download. A list of IDEs is maintained on the [Python wiki](https://wiki.python.org/moin/IntegratedDevelopmentEnvironments)².

IDLE

IDLE stands for Integrated DeveLopment Environment. It comes bundled with Python. To start IDLE, type "idle" (without the quotes) at a Terminal window or Command Prompt and then press **Enter**.

2. See <https://wiki.python.org/moin/IntegratedDevelopmentEnvironments>.

1.2 Hello, world!

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:

Code Sample

`python-basics/Demos/hello_world.py`

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

Code Explanation

To run this code:

1. Open a Terminal window or Command Prompt at your [ClassFiles](#) folder.
2. Run the file

```
python /python-basics/Demos/hello_world.py
```

3. The "Hello, world!" message will be displayed in your Terminal window.

The output should look like this:

```
Hello, world!
```

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

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 your file using the Python interpreter just as you did earlier for the demonstration.
5. The output should look like the following:

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

Exercise 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 Terminal window.

Python 2 Difference

In Python 2, `print` is a statement, not a function. So, instead of writing:

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

In Python 2, you would leave out the parentheses and write:

```
print "Hello, world!"
```

1.3 Literals

Note that 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 just 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 literals. Just make sure that the open and close quotation marks match.

1.4 Python Comments

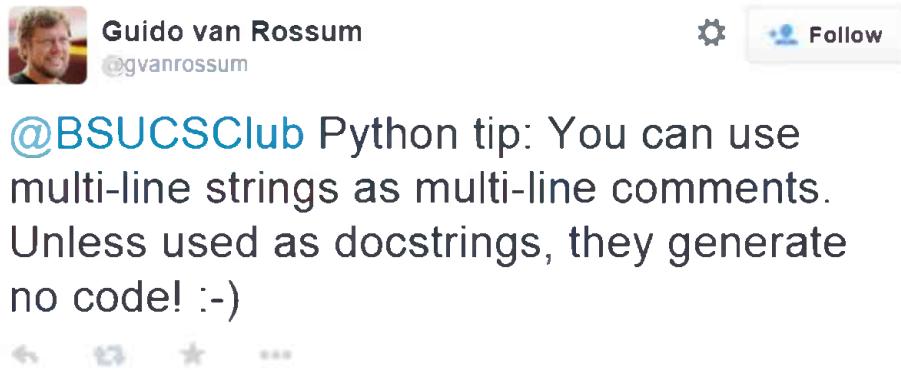
In the previous demo, you may have noticed this line of code:

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

That hash mark (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 tip as a workaround:



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

```
Syntax  
"""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:

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

Why Python 3.x?

Python 3 was released in 2008. It is, according to python.org, "the present and future of the language."³ Unless you are supporting or working with legacy Python code, you should learn Python 3.x. Once you know Python 3.x, it is quite easy to learn the differences in Python 2.x. Those differences are detailed in Mark Summerfield's [Moving from Python 2 to Python 3](#).

3. See <https://wiki.python.org/moin/Python2orPython3>.

4. See http://ptgmedia.pearsoncmg.com/imprint_downloads/informit/promotions/python/python2python3.pdf

1.5 Data Types

In Python programming, objects have different *data types*. The data type determines both what an object can be 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.

The table below shows the basic data types used in Python.

Common Built-in Data Types

Data Type	Abbreviation	Explanation
boolean	bool	A True or False value.
integer	int	A whole number.
float	float	A decimal.
string	str	A sequence of Unicode ⁵ characters.
list	list	An ordered sequence of values. Like an array in other languages.
tuple	tuple	A list of fixed length.
dictionary	dict	An unordered grouping of key-value pairs.
set	set	An unordered grouping of values.

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 a Terminal window or Command Prompt.
2. Start the interactive Python mode by typing `python` and then pressing **Enter**. You are now in the Python Shell.
3. Type `type(3)`. Press **Enter**.
4. Type `type(3.1)`. Press **Enter**.
5. Type `type("3")`. Press **Enter**.
6. Type `type("pizza")`. Press **Enter**.
7. Type `type(1==1)`. Press **Enter**.
8. Type `type((1, 2, 3))`. Press **Enter**.
9. Type `type([1, 2, 3])`. Press **Enter**.
10. Type `type({1, 2, 3})`. Press **Enter**.

Python Basics

Exercise 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(1==1)
<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 the data types above. We will cover them all in this course.

Class and Type

If you have experience with object-oriented programming, you may wonder at Python's use of the word "class" when outputting a data type. In Python, built-in types are the same as user-defined classes

1.6 Variables

Variables in Python are untyped and do not need to be declared. 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 `firstname` is different from `FirstName`. 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.

Underscores in Variable Names

Avoid beginning variable names with underscores until you have learned to write Python classes.

Keywords

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

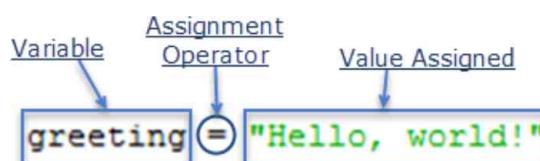
1. `and`
2. `as`
3. `assert`
4. `break`
5. `class`
6. `continue`
7. `def`
8. `del`
9. `elif`
10. `else`

```
11. except
12. False
13. finally
14. for
15. from
16. global
17. if
18. import
19. in
20. is
21. lambda
22. nonlocal
23. None
24. not
25. or
26. pass
27. raise
28. return
29. True
30. try
31. while
32. with
33. yield
```

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.

Code Sample

python-basics/Demos/hello_variables.py

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

Code Explanation

To run this code:

1. Open a Terminal window or Command Prompt.
2. Run the file by typing `python` followed by the file name (remember to include the path to the file, e.g., `python-basics/Demos/hello_variables.py`).

The output will be the same as it was in the previous demo (see page 4).

Simultaneous Assignment

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

Syntax

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

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

```
smart_1, cute_1, funny_1, quiet_1 = "John", "Paul", "Ringo", "George"
```

But simultaneous assignment is really useful in a scenario like the one shown in the following demo.

Python Basics

Code Sample

python-basics/Demos/simultaneous_assignment.py

```
1. #Switch b and a. Attempt 1. The Wrong Way.
2. a=5
3. b=10
4. a=b
5. b=a
6. print("ATTEMPT 1. The Wrong Way.")
7. print(a)
8. print(b)
9.

10. #Switch b and a. Attempt 2. A Way that Works.
11. a=5
12. b=10
13. temp=a
14. a=b
15. b=temp
16. print("ATTEMPT 2. A Way that Works.")
17. print(a)
18. print(b)
19.

20. #Switch b and a. Attempt 3. The Python Way.
21. a=5
22. b=10
23. a,b = b,a
24. print("ATTEMPT 3. The Python Way.")
25. print(a)
26. print(b)
```

Code Explanation

To run this code:

1. Run the file in a Terminal window or Command Prompt:

```
python python-basics/Demos/simultaneous_assignment.py
```

2. The output should look like this:

```
>>>
ATTEMPT 1. The Wrong Way.
10
10
ATTEMPT 2. A Way that Works.
10
5
ATTEMPT 3. The Python Way.
10
5
>>>
```

The code demonstrates that simultaneous assignment *really is simultaneous*. The value for a does not change before the value of b, so b gets the original value of a, and we can switch the values without using an interim step of creating a temporary (and otherwise useless) variable.

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 using the Python interpreter.

Exercise Solution

python-basics/Solutions/today.py

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

Code Explanation

The above code will render the following (depending on the day you used):

```
>>>
Monday
>>>
```

Constants

In programming, a constant is like a variable in that it is an identifier that holds 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" #RGB value of red
```

Deleting Variables

Variables can be deleted using the `del` statement, like this:

```
a = 1
print(a) #prints 1
del a
print(a) #NameError: name 'a' is not defined
```

1.7 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 and discuss indenting. A function is created using the `def` keyword like this:

Python Basics

Code Sample

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

To run this code:

1. Open a Terminal window or Command Prompt.
2. Run the file by typing `python` followed by the file name (remember to include the path to the file, e.g., `python-basics/Demos/indent_demo.py`).

The above code 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 it.

```

indent-demo.py - C:\Users\Nat\Documents\_Webucator\courseware\Python\ClassFiles\pyth...
File Edit Format Run Options Windows Help
def main():
    print("I am part of the function.")
    print("I am also part of the function.")
    print("Hey, me too!")
print("Sad not to be part of the function. I've been outdented.")

main()

```

A few things to note about functions:

1. Functions are created using the `def` keyword.
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 later), 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 outdented is not part of the function. You'll learn that indenting matters to Python in other areas as well.
6. Functions are called (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.⁶

Again, we're not going to delve into the whys and hows of modular programming right now, but there is a quick advantage we get from writing our code this way: we can now re-run the program from within the Python Shell. To illustrate, type `python` and press **Enter** in your Terminal window or Command Prompt to enter the Python Shell (interactive mode). Next, execute your script by typing:

6. 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

```
exec( open('python-basics/Demos/indent_demo.py').read() )
```

Don't worry if you don't understand this code yet. The important thing is that it runs a Python module from the Python Shell.

You will see the same output as you saw in the demonstration above. Now, if you type `main()` and press **Enter**, you will see the same output again! The reason for this is you have called the `main()` function is now in memory.

1.8 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 default, it will output them all separated by a space. 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:

Code Sample

python-basics/Demos/variable_and_string_output.py

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

Code Explanation

To run this code:

1. Run the Python interpreter passing this script as the first parameter as you have done in previous demos and exercises.

The above code will render the following:

```
>>>
Today is Monday .
>>>
```

You may have noticed the extra space before the period 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 trail the printed objects. The default is a newline character (denoted with "`\n`"). You can use an empty string (e.g., "") to specify that nothing should trail the printed objects.

The following demo shows how these two arguments can be used to get rid of the extra space we saw in the previous example.

Code Sample

`python-basics/Demos/variable_and_string_output_fixed_spacing.py`

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

Code Explanation

Run the code in a Terminal window or Command Prompt (by now, you should know how).

The above code will render the following:

```
>>>  
Today is Monday.  
>>>
```

Concatenation

Later in the course, we'll learn how to use concatenation to combine strings, so that we won't need two separate calls to `print()` to output a single line of text.

1.9 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; it returns the input from the user as a string. The demo below shows how to use it to prompt the user for the day of the week.

Code Sample

`python-basics/Demos/input.py`

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

Code Explanation

Run the code in a Terminal window or Command Prompt.

The above code will render the following:

```
>>>
What day is it? Friday
Wow! Today is Friday? Awesome!
>>>
```

Python 2 Difference

In Python 2, the `input()` function parses and evaluates the user input. Python 2 has a `raw_input()` function that behaves like `input()` in Python 3.

Python 2

```
>>> input('Math problem? ')
Math problem? 2+2
4

>>> raw_input('Math problem? ')
Math problem? 2+2
'2+2'
```

Python 3

```
>>> input('Math problem? ')
Math problem? 2+2
'2+2'
```

Exercise 4 Hello, You!

5 to 10 minutes

In this exercise, you will ...

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 his/her name, output a greeting.
4. Save your changes.
5. Test your solution.

Exercise Solution

[python-basics/Solutions/hello_you.py](#)

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

Code Explanation

The code should work something like this:

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

1.10 Getting Help

In addition to [the official Python documentation \(<https://docs.python.org>\)](https://docs.python.org) and the many other resources on the web for getting help on Python, you can get great help using the built-in `help()` function in the Python Shell.

You can use `help()` without passing any arguments to open interactive help. This will give you a help prompt at which you can enter keywords, module names, symbols, or topic names to get relevant help.

To see how the interactive help works, try this:

1. Open the Python Shell from a Terminal window or Command Prompt.
2. Type `help()` and press **Enter**.

Python Basics

3. Type keywords and press **Enter**. You should get a list of keywords in Python:

```
help> keywords

Here is a list of the Python keywords. Enter any keyword to get more help.

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

help>

4. Type any one of the keywords and press **Enter**. You should get documentation on that keyword.

To leave interactive help, type quit and press **Enter**.

You can also pass a value to the `help()` function when you call it. For example, to get help on the `print()` function, you can call `help(print)`:

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

print(*args, **kwargs)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep: string inserted between values, default a space.
    end: string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

The first indented line after the line that reads `print(...)` shows the **signature** of the function. This tells you that the `print()` function can take several arguments, most of which have default values and are therefore optional.

Some functions, like the `input()` function, return an object, but `print()` does not. When a function does return an object, the documentation will generally tell you what data type the returned object will be.

None

When a function does not explicitly return anything, Python returns the value `None`. This lets the calling code know not to expect anything. The following code illustrates this:

```
>>> result = print("Hello")
Hello
>>> print(result)
None
>>>
```

Sometimes the built-in help documentation can be a little daunting, but as you gain experience with Python, you will find it easier to understand

1.11 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, to write very simple Python functions and modules, and to use the built-in help documentation.

2. Functions and Modules

In this lesson, you will learn...

1. To define and call functions
2. To define parameters in and pass arguments to functions.
3. To set default values for parameters.
4. About variable scope.
5. To return values from functions.
6. To create and import modules.

2.1 Defining Function

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:

Syntax

```
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:

Code Sample

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

Functions and Modules

`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.

Code Sample

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 above 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 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 wouldn't necessarily have a `main()` function.

2.2 Variable Scope

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

Answer: It doesn't know the user's name.

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

Code Sample**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.

2.3 Global Variables

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

Code Sample

`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:

Functions and Modules

Code Sample

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 underscore⁷](#) (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).

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

2.4 Function Parameters

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

```
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:

Syntax

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

Here is our "Hello, You!" program using parameters:

Functions and Modules

Code Sample

`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. It allows us to move `your_name = input('What is your name?')` to the `main()` function so we can pass `your_name` into both `say...` functions.
2. It allows us to move the call to `insert_separator()` out of the `say_hello()` function as it really doesn't belong there.

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*.

```
parameters-vs-arguments.py - C:/Users/Nat/Documents/_Webucator/cp
File Edit Format Run Options Windows Help
def ask_something(something):
    print(something)

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:

Functions and Modules

```
def divide(numerator, denominator):
    return int(numerator / denominator)

divide(10,2) #returns 5
divide(numerator=10, denominator=2) #returns 5
divide(denominator=2, numerator=10) #returns 5
divide(10, denominator=2) #returns 5
```

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

Exercise 5 A Function with Parameters

15 to 25 minutes

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

1. Run `functions/Exercises/add_nums.py` using the `python` interpreter. The output should look like this:

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

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

***Challenge**

How well do you understand variable scope? Take a look at the following code and see if you can figure out what will be printed out

Functions and Modules

Code Sample

functions/Exercises/global_vs_local.py

```
1.     a,b,c = 1,2,3
2.
3.     def change_values(a,c):
4.         global b
5.         print(a,b,c) #what gets printed?
6.         a,b,c = 0,0,0
7.
8.     def main():
9.         global a
10.        a=-1
11.        change_values(c,a)
12.        print(a,b,c) #what gets printed?
13.
14.    main()
```

Functions and Modules

Exercise Solution

functions/Solutions/add_nums.py

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

Functions and Modules

Challenge Solution

```
3 2 -1  
-1 0 3
```

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:

Syntax

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

For example:

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

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

Exercise 6 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 = 50
101 + 201 + 301 = 603
```

Exercise Code

[functions/Exercises/add_nums_with_defaults.py](#)

```
1.     def add_nums(num1,num2,num3,num4,num5):
2.         sum = num1 + num2 + num3 + num4 + num5
3.         print(num1,'+', num2,'+', num3,'+', num4,'+', num5, ' = ', sum)
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()
```

Functions and Modules

Exercise Solution

functions/Solutions/add_nums_with_defaults.py

```
1.      def add_nums(num1,num2,num3=0,num4=0,num5=0):
2.          sum = num1 + num2 + num3 + num4 + num5
3.          print(num1,'+', num2,'+', num3,'+', num4,'+', num5, ' = ', sum)
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()
```

2.5 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 code below:

Code Sample

`functions/Demos/add_nums_with_return.py`

```

1.  def add_nums(num1, num2, num3=0, num4=0, num5=0):
2.      sum = num1 + num2 + num3 + num4 + num5
3.      return sum
4.
5.  def main():
6.      sum = add_nums(1,2)
7.      print(sum)
8.      sum = add_nums(sum,3)
9.      print(sum)
10.     sum = add_nums(sum,4)
11.     print(sum)
12.     sum = add_nums(sum,5)
13.     print(sum)
14.     sum = add_nums(sum,6)
15.     print(sum)
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 variable, which we also named `sum`, but could have named something different. Then we decide what to do with that result - we print it and then 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 calling function.

2.6 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:

Code Sample

`functions/Demos/import_example.py`

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

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.

When you import a module into another module, the code in the imported module will run. This might be undesirable. It's possible (and common) to wrap the call to the `main()` function (or any other code) in a condition so that it will only run if the module is executed directly (i.e., is not imported into another module). 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:

Syntax

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

For example:

Code Sample**functions/Demos/import_example2.py**

```

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

```

When you use this second 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:

Syntax

```

import add_nums_with_return as anwr

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

```

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 script doing the importing).
2. The library of standard modules.
3. The paths defined in `sys.path` (see page 297).

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.

2.7 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.

Functions and Modules

3. Math

In this lesson, you will learn...

1. To do basic math in Python.
2. To use the `math` module for additional math operations.
3. To use the `random` module to generate random numbers.

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.

3.1 Arithmetic Operators

The table below lists the arithmetic operators in Python. Most are familiar to us already. We'll explain the others below.

Arithmetic Operators

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

Here are the examples in Python:

Code Sample

Math/Demos/arithmetic_operators.py

```

1. print(5+2) #returns 7
2. print(5-2) #returns 3
3. print(5*2) #returns 10
4. print(5/2) #returns 2.5
5. print(5%2) #returns 1
6. print(5**2) #returns 25
7. print(5//2) #returns 2

```

Modulus and Floor Division

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

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

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

```
5 // 2 #returns 2  
11 // 3 #returns 3  
22 // 4 #returns 5  
22 // 3 #returns 7  
10934 // 324 #returns 33  
-5 // 2 #returns -3 (it's rounded down, meaning towards negative infinity)
```

Python 2 Difference

In Python 3, division of two integers will **always return a float** even when the result is a whole number. For example, 10 / 2 returns 5.0.

In Python 2, division of two integers will **always return an integer** even when the result is a fraction. For example, 5 / 2 returns 2.

See [PEP 238 -- Changing the Division Operator](https://www.python.org/dev/peps/pep-0238/) (<https://www.python.org/dev/peps/pep-0238/>) for an explanation of why this change was made.

Exercise 7 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 could understand (e.g., "5 divided by 2 equals 2 with a remainder of 1").

1. Open [Math/Exercises/floor_modulus_v](#) in your editor.
2. Write the `divide()` function.
3. Try passing the `divide()` function different values in the Python Shell as shown in the screenshot below:

```
>>>
5 divided by 2 equals 2 with a remainder of 1
>>> divide(9,3)
9 divided by 3 equals 3 with a remainder of 0
>>> divide(27,4)
27 divided by 4 equals 6 with a remainder of 3
>>> divide(36,5)
36 divided by 5 equals 7 with a remainder of 1
>>> |
```

Exercise Solution

Math/Solutions/floor_modulus.p

```
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.      numerator = 5
9.      denominator = 2
10.     divide(numerator,denominator)
11.
12. main()
```

3.2 Assignment Operators

The table below lists the assignment operators in Python.

Assignment Operators

Operator	Description	Example
=	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
%=	One step modulus and assignment	a%=2 same as a = a % 2
=	One step exponent and assignment	a=2 same as a = a**2
//=	One step floor division and assignment	a//=2 same as a = a // 2

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

Code Sample

Math/Demos/assignment_operators.py

```

1. a = 5
2. a+=2
3. print(a) #returns 7
4. a-=2
5. print(a) #returns 5
6. a*=2
7. print(a) #returns 10
8. print(type(a)) #returns <class 'int'>
9. a/=2
10. print(a) #returns 5.0
11. print(type(a)) #returns <class 'float'>
12. a%=2
13. print(a) #returns 1.0
14. a**=2
15. print(a) #returns 1.0
16. a//=2
17. print(a) #returns 0.0

```

Code Explanation

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

Precedence of Operations

The order of operations for the operators shown in the table above 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 $6 / 2 * 3$ is the same as $(6 / 2) * 3$.

3.3 Built-in 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) #object is integer already. Returns 5
int('5') #object is string. Returns 5
int(5.4) #object is float. Returns 5
int(5.9) #object is float. Returns 5
int(-5.9) #object is float. Returns -5
int('5.4') #object is string. Returns ValueError: invalid literal
```

eval(str)

The `eval()` function evaluates a string as a Python expression.

```
eval('5 - 3') #returns 2
eval("print('foo')") #returns foo
```

Although both `eval()` and `int()` can be used to convert strings representing integers to actual integers, `int()` has the advantage of being able to handle leading zeros. Consider the following:

```
>>> int('0100')
100
>>> eval('0100')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    eval('0100')
  File "<string>", line 1
    0100
          ^
SyntaxError: invalid token
```

As you can see, passing a string with a leading zero to `eval()` causes it to error, but `int()` handles it just fine

float(x)

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

```
float(5) #object is integer. Returns 5.0
float('5') #object is string. Returns 5.0
float(5.4) #object is already float. Returns 5.4
float('5.4') #object is string. Returns 5.4
float('-5.99') #object is string. Returns -5.99
float(-5.99) #object is float. Returns -5.99
```

abs(x)

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

```
abs(-5) #Returns 5
abs(5) #Returns 5
abs(-5.5) #Returns 5.5
abs(5.5) #Returns 5.5
```

min(args) and max(args)⁸

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.

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

The `min()` and `max()` functions can also take an *iterable* containing values to compare. We will cover this in the Iterables lesson (see page 137).

pow(x,y[,z])

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

```
pow(4, 2)
```

is the same as

```
4**2
```

If the `z` argument is present, `pow(x, y, z)` is functionally equivalent to (but purportedly⁹ more efficient than) `x**y % z`:

```
pow(4, 2, 3)
```

is the same as

```
4**2 % 3
```

-
8. The `min()` and `max()` functions can also compare strings (see page 111).
 9. According to the Python documentation (<https://docs.python.org/3/library/functions.html#pow>), `pow(x, y, z)` is computed more efficiently than `pow(x, y) % z`, but our tests don't show that to be the case. In fact, we found `x**y % z` to be the most efficient way to complete the operation. Note that this is academic as all the methods we tested for raising one number to the power of another and then doing a modulus operation are lightning fast. You should use whichever method suits your coding style the most.

round(number[, n])

`round(number [, n])` returns `number` as a number rounded to `n` digits after the decimal. If `n` is 0 (zero) then it rounds to the nearest whole number. If `n` is -1 then it rounds to the nearest 10 (e.g., `round(55, -1)` returns 50).

```
round(3.14) #Returns 3
round(-3.14) #Returns -3
round(3.14,1) #Returns 3.1
round(3.95,1) #Returns 4.0
round(1111,0) #Returns 1111
round(1111,-2) #Returns 1100
```

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 137) as well.

3.4 The math Module

Examples from this section are included in `math_module.py` in the `math/Demos` folder.

The `math` module is built in to Python and provides many useful math functions. We cover some of them here.¹⁰

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

```
import math
```

The following table shows common `math` module methods in Python.

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

Common Methods of the `math` Module

Method	Returns	Example
<code>math.ceil(x)</code>	<code>x</code> rounded up to the nearest whole number as an integer.	<code>math.ceil(5.4)</code> #Returns 6 <code>math.ceil(-5.4)</code> #Returns -5
<code>math.floor(x)</code>	<code>x</code> rounded down to the nearest whole number as an integer.	<code>math.floor(5.6)</code> #Returns 5 <code>math.floor(-5.6)</code> #Returns -6
<code>math.trunc(x)</code>	<code>x</code> with the fractional truncated, effectively rounding towards 0 to the nearest whole number as an integer.	<code>math.trunc(5.6)</code> #Returns 5 <code>math.trunc(-5.6)</code> #Returns -5
<code>math.fabs(x)</code>	The absolute value of float <code>x</code> . This is similar to the built-in <code>abs(x)</code> function except that <code>math.fabs(x)</code> always returns a float whereas <code>abs(x)</code> returns a number of the same data type as <code>x</code> .	<code>math.fabs(-5)</code> #returns 5.0 <code>abs(-5)</code> #returns 5
<code>math.factorial(x)</code>	The factorial of <code>x</code> . This is often written as <code>x!</code> , but not in Python!	<code>math.factorial(3)</code> #returns 6 <code>math.factorial(5)</code> #returns 120
<code>math.fmod(x, y)</code>	Works in the same way as the Python modulus operator (i.e., <code>x % y</code>); however, <code>math.fmod(x, y)</code> always returns a float and for some edge cases the Python modulus operator is imprecise. So, <code>math.fmod(x, y)</code> should be used with floats and <code>x % y</code> should be used with integers.	<code>math.fmod(5, 2)</code> #returns 1.0
<code>math.pow(x, y)</code>	<code>x</code> raised to the power <code>y</code> as a float	<code>math.pow(5, 2)</code> #returns 25.0
<code>math.sqrt(x)</code>	The square root of <code>x</code> as a float	<code>math.sqrt(25)</code> #returns 5.0

The `math` module also contains two constants: `math.pi` and `math.e`, for Pi and *e* as used in the [natural logarithm](#).¹¹

11. See http://en.wikipedia.org/wiki/Natural_logarithm.

Additional math Functions

The math library offers many additional functions, including:

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

3.5 The random Module¹²

Examples from this section are included in `random_module.py` in the `math/Demos` folder.

The random module is also built into Python.

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

```
import random
```

The following table shows common `random` module methods in Python.

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

Common Methods of the `random` Module

Method	Returns	Example
<code>random.random()</code>	Random float between 0 and 1.	<code>random.random()</code>
<code>random.randint(a,b)</code>	Random integer between (and including) a and b.	<code>random.randint(1,10)</code>
<code>random.randrange(b)</code>	Random integer between 0 and b-1.	<code>random.randrange(10)</code> #returns one of 0 through 9
<code>random.randrange(a,b)</code>	Random integer between (and including) a and b-1.	<code>random.randrange(1,10)</code> #returns one of 1 through 9
<code>random.randrange(a,b,step)</code>	Random integer at step between (and including) a and b-1.	<code>random.randrange(1,10,2)</code> #returns one of 1,3,5,7,9
<code>random.uniform(a,b)</code>	Random float between (and including) a and b (or b and a).	<code>random.uniform(1,10)</code>
<code>random.choice(seq)</code>	Returns random element in seq.	Sequences are covered later in the course. (see page 117)
<code>random.shuffle(seq)</code>	None. Shuffles seq in place.	Sequences are covered later in the course. (see page 117)

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.seed(1)
>>> random.randint(1,100)
18
>>> random.randint(1,100)
73
>>>
```

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

Exercise 8 How Many Pizzas Do We Need?

25 to 45 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. The screenshot below shows how the program should work:

```
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 leftover slices.
>>> main()
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 leftover slices.
>>> |
```


Exercise Solution**math/Solutions/pizza_slices.py**

```
1.     import math
2.
3.     def calculate_slices(people, slices_per_person):
4.         return slices_per_person * people
5.
6.     def calculate_pizzas(slices,slices_per_pie):
7.         return math.ceil(slices / slices_per_pie)
8.
9.     def calculate_slices_left(slices_per_pie, pizzas, slices_needed):
10.        total_slices = slices_per_pie * pizzas
11.        return total_slices - slices_needed
12.
13.    def main():
14.        people = int(input("How many people are eating? "))
15.        slices_per_person = float(input("How many slices per person? "))
16.        slices = calculate_slices(people, slices_per_person)
17.
18.        slices_per_pie = int(input("How many slices per pie? "))
19.        pizzas = calculate_pizzas(slices,slices_per_pie)
20.
21.        print('You need', pizzas, 'pizzas to feed', people, 'people.')
22.
23.        slices_left = calculate_slices_left(slices_per_pie,pizzas,slices)
24.        print('There will be', slices_left, 'leftover slices.')
25.
26.    main()
```

3.6 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.

4. Python Strings

In this lesson, you will learn...

1. To work with strings.
2. To escape special characters.
3. To work with multi-line strings.
4. To index and slice strings.
5. To use common string operators and methods.
6. To format strings.
7. To use built-in string functions.

According to the [Python documentation](#)¹³, "Strings are immutable sequences of Unicode code points." Less technically speaking, strings are sequences of characters¹⁴. The term *sequence* in Python refers to an ordered set. Other common sequence types are lists, tuples, and ranges, all of which will be covered later in this course.

4.1 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 output 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 output 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:

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

14. 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).

Python Strings

```
phrase = "The soldier said, \"You've got two empty halves of a co >>
conut and you're bangin' 'em together.\""  
#or  
phrase = 'The soldier said, "You\'ve got two empty halves of a co >>
conut and you\'re bangin\' \'em together."'
```

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: \\\"
```

Two other common special characters are the linefeed (\n) and horizontal tab (\t).

Escape Sequences

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

Triple Quotes

Triple quotes are used to create multi-line strings. You can use three double quotes¹⁵ as shown in the example below:

15. 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 Sample

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.     -----""")  
9.  
10.  
11.    print('''-----  
12.    LUMBERJACK SONG  
13.  
14.    I'm a lumberjack  
15.    And I'm O.K.  
16.    I sleep all night  
17.    And I work all day.  
18.    -----'''')
```

Python Strings

Code Explanation

The above 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.
-----
-----
LUMBERJACK SONG

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

Notice 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 linefeed without having that linefeed show up in your output. You can escape the actual linefeed with a backslash as shown below:

Code Sample

`strings/Demos/triple_quotes_linefeed_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 above 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.

4.2 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 strin

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 (th right-most) is at position -1. The table below 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 usin indexing.

Python Strings

Code Sample

strings/Demos/string_indexing.py

```
1.     phrase = "Monty Python"
2.
3.     first_letter = phrase[0]
4.     # [M]onty Python
5.     print(first_letter)
6.
7.     last_letter = phrase[-1]
8.     #Monty Pytho[n]
9.     print(last_letter)
10.
11.    fifth_letter = phrase[4]
12.    #Mont[y] Python
13.    print(fifth_letter)
14.
15.    third_letter_from_end = phrase[-3]
16.    #Monty Pyt[h]on
17.    print(third_letter_from_end)
```

Code Explanation

The expected output for each print statement is shown in square brackets in the preceding comment.

Exercise 9 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 (s)he 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 a screenshot of the program completed by the user:

```
>>>
Choose a phrase? Hello, World!
Your phrase is 'Hello, World!'
Which character would you like to see? [Enter number] 4
Character number 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 would you like to see? [Enter number] 4
Character number 4 is 'l'
>>>
```

Python Strings

Exercise 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 would you like to see? [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 would you like to see? [Enter
>>> number] ")) - 1
5.      print("Character number ", pos+1, " is '", phrase[pos], "'", sep="")
6.
7.  main()
```

4.3 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:

Syntax

```
substring = original_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.

Python Strings

Code Sample

strings/Demos/string_slicing.py

```
1.     phrase = "Monty Python"
2.
3.     first_5_letters = phrase[0:5]
4.     #[Monty] Python
5.     print(first_5_letters)
6.
7.     letters_2_thru_4 = phrase[1:4]
8.     #M[ont]y Python
9.     print(letters_2_thru_4)
10.
11.    letter_5_to_end = phrase[4:]
12.    #Mont[y Python]
13.    print(letter_5_to_end)
14.
15.    last_3_letters = phrase[-3:]
16.    #Monty Pyt[hon]
17.    print(last_3_letters)
18.
19.    first_3_letters = phrase[:3]
20.    # [Mon]ty Python
21.    print(first_3_letters)
22.
23.    three_letters_before_last = phrase[-4:-1]
24.    #Monty Py[tho]n
25.    print(three_letters_before_last)
26.
27.    copy_of_string = phrase[:]
28.    # [Monty Python]
29.    print(copy_of_string)
```

Code Explanation

The expected output for each print statement is shown in square brackets in the preceding comment.

Exercise 10 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 (s)he entered (e.g., "Your phrase is 'Hello, World!'").
 - C. Prompts the user for a start number.
 - D. Prompts the user for a 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 a screenshot of the program completed by the user:

```
>>>
Choose a phrase? Hello, World!
Your phrase is 'Hello, World!'
Which character would you like to start with? [Enter number] 4
Which character would you like 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!'
Which character would you like to start with? [Enter number] 4
Which character would you like to end with? [Enter number] 9
Your substring is 'lo, Wo'
>>> |
```

Python Strings

Exercise 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("Which character would you like to start with?
>>> [Enter number] "))
5.      pos2 = int(input("Which character would you like 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("Which character would you like to start with?
>>> [Enter number] ")) - 1
5.      pos2 = int(input("Which character would you like to end with? [Enter
>>> number] "))
6.      print("Your substring is '",phrase[pos1:pos2],"',sep="")
7.
8.  main()
```

4.4 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:

Code Sample

strings/Demos/concatenation.py

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

Code Explanation

The above 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.

Code Sample

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 above code will render the following:

```
>>>
neeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneeneene
neeneeneenee
>>> |
```

Exercise 11 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 into 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.

Python Strings

Exercise Solution

strings/Solutions/hello_you.py

```
1.  def say_hello(name):
2.      print('Hello, ', name, '!', sep=' ')
3.
4.  def insert_separator(char="-", repeat=30):
5.      print(char*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"
```

This will return "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"
```

This will return "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:

Code Sample

[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 above code will render the following:

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

4.5 Common String Methods

Examples from this section are included in the following three files in the [string/Demos](#) folder.

1. [methods_that_return_copy.py](#)

Python Strings

2. [methods_that_return_boolean.py](#)
3. [methods_that_return_index.py](#)

The following tables show common string methods in Python.

String Methods that Return Copy of String

Method	Returns	Example
<code>str.capitalize()</code>	A string with only the first letter capitalized.	"hELLo".capitalize() returns "Hello"
<code>str.lower()</code>	An all lowercase string.	"hELLo".lower() returns "hello"
<code>str.swapcase()</code>	A string with the case of each letter swapped.	"hELLo".swapcase() returns "Hello"
<code>str.title()</code>	A string with each word beginning with a capital letter followed by all lowercase letters.	"hello world".title() returns "Hello World"
<code>str.upper()</code>	An all uppercase string.	"hELLo".upper() returns "HELLO"
<code>str.replace(old,new[,count])</code>	A string with old replaced by new count times.	"mom".replace("m","b") returns "bob"
<code>str.strip([chars])</code>	A string with leading and trailing chars removed. chars defaults to whitespace.	" hello ".strip() returns "hello"
<code>str.lstrip([chars])</code>	A string with leading chars removed. chars defaults to whitespace.	"hello".lstrip("h") returns "ello"
<code>str.rstrip([chars])</code>	A string with trailing chars removed. chars defaults to whitespace.	"hello".rstrip("o") returns "hell"

Python Strings

String Methods that Return a Boolean value (i.e., True or False)

Method	Returns	Example
<code>str.isalnum()</code>	True if all characters are letters or numbers.	"Hello World!".isalnum() returns False
<code>str.isalpha()</code>	True if all characters are letters.	"Hello World!".isalpha() returns False
<code>str.islower()</code>	True if string is all lowercase.	"hello".islower() returns True
<code>str.isupper()</code>	True if string is all uppercase.	"HELLO".isupper() returns True
<code>str.istitle()</code>	True if string is title case.	"Hello World!".istitle() returns True
<code>str.isspace()</code>	True if string is made up of only whitespace.	" ".isspace() returns True
<code>str.isdigit()</code>	True if all characters are digits.	"5.5".isdigit() returns False
<code>str.isdecimal()</code>	True if all characters are decimal characters ¹⁶ .	"5.5".isdecimal() returns False
<code>str.isnumeric()</code>	True if all characters are numeric.	"5.5".isnumeric() returns False
<code>str.startswith(prefix[, start[, end]])</code>	True if string starts with prefix. Start looking at start index and end looking at end index.	"hello".startswith("h") returns True
<code>str.endswith(suffix[, start[, end]])</code>	True if string ends with suffix. Start looking at start index and end looking at end index.	"hello".endswith("o") returns True

Note that all the `is...` methods shown in the table above return `False` if the string has zero length.

16. The difference between `isdecimal()` and `isdigit()` is mostly academic. If you're really curious about it, run [strings/Demos/decimal_vs_digit.py](#) to get a list of unicode characters that are digits but not decimals.

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

Method	Returns	Example
<code>str.find(sub[, start[, end]])</code>	The lowest index where <code>sub</code> is found. Returns -1 if <code>sub</code> isn't found. Start looking at <code>start</code> index and end looking at <code>end</code> index.	"Hello World!".find("l") returns 2
<code>str.rfind(sub[, start[, end]])</code>	The highest index where <code>sub</code> is found. Returns -1 if <code>sub</code> isn't found. Start looking at <code>start</code> index and end looking at <code>end</code> index.	"Hello World!".rfind("l") returns 9
<code>str.index(sub[, start[, end]])</code>	Same as <code>find()</code> , but errors when <code>sub</code> is not found.	"Hello World!".index("l") returns 2
<code>str.rindex(sub[, start[, end]])</code>	Same as <code>rfind()</code> , but errors when <code>sub</code> is not found.	"Hello World!".rindex("l") returns 9

`str.count()`

Method	Returns	Example
<code>str.count(sub[, start[, end]])</code>	The number of times <code>sub</code> is found. Start looking at <code>start</code> index and end looking at <code>end</code> index.	"Hello World!".count("l") returns 3

4.6 String Formatting

Python includes powerful options for formatting strings. The most common way to format strings is to use the `format()` method combined with the [Format Specification Mini-Language](#)¹⁷.

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')
```

The above code will output 'Monty Python is an awesome movie!' The curly brackets are used to indicate a replacement field, which takes position arguments specified by index (as in the example above) or by name (as in the example below):

17. See <https://docs.python.org/3/library/string.html#format-specification-mini-language>

Python Strings

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

As of Python 3.1, 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.

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 specification¹⁸ is:

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

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

18. 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>.

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 scientist¹⁹, 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:

```
'On a scale of {0:d} to {1:d}, I give {2:s} a {3:d}'.format(1, 5,
    'Monty Python', 6)

#Simplify by removing field names (indexes).
'On a scale of {:d} to {:d}, I give {:s} a {:d}'.format(1, 5,
    'Monty Python', 6)

#Further simplify by removing default type specifiers.
'On a scale of {} to {}, I give {} a {}'.format(1, 5, 'Monty
    Python', 6)

#And with the field name and type specifier gone, we can
#remove the colon separator.
'On a scale of {} to {}, I give {} a {}'.format(1, 5, 'Monty
    Python', 6)
```

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.

19. 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>.

Python Strings

```
import math
#fixed point
'pi equals {:.f}'.format(math.pi)
#outputs 'pi equals 3.141593'

#no type specified
'pi equals {}'.format(math.pi)
#outputs '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:

```
'pi equals {:.2f}'.format(math.pi)
#outputs '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)
#outputs '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. Here are some simple examples:

```
#Example 1
'{:5}'.format('abc')
#outputs 'abc  '

#Example 2
'{:5}'.format(123)
#outputs ' 123'

#Example 3
'{:5.2f}'.format(123)
#outputs '123.00'
```

The width of the formatted string in all three cases is five.

The first example is formatting a string ('abc') of three characters. The extra two characters are called the padding. By default, for strings the padding comes after the string, so that the string is aligned to the left.

The second example is formatting a number. By default, it uses the decimal number format, which (perhaps counterintuitively) displays as an integer. By default, for numbers the padding comes before the number, so that the string is aligned to the right.

The third example is formatting a number, 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, there is no room for 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)
#outputs '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')
#outputs ' abc'

'{:<5}'.format(123)
#outputs '123  '

'{:^5}'.format(123)
#outputs ' 123  '

'{:+5}'.format(123)
#outputs '+ 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:

```
'{.:^10.2f}'.format(math.pi)
#Note the period after the colon
#outputs '...3.14...' instead of '    3.14    '
```

Percentage Type

As mentioned earlier, another option for type is percentage. Consider the following example:

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

The examples above are all shown in the following file

Python Strings

Code Sample

strings/Demos/formatting.py

```
1. import math
2.
3. #Fixed Point
4. print('pi equals {:.f}'.format(math.pi))
5. #outputs 'pi equals 3.141593'
6.
7. #No Type Specified
8. print('pi equals {}'.format(math.pi))
9. #outputs 'pi equals 3.141592653589793'
10.
11. #Precision
12. print('pi equals {:.2f}'.format(math.pi))
13. #outputs 'pi equals 3.14'
14.
15. #Separate by Thousands
16. print('{:, .2f}'.format(1000000))
17. #outputs '1,000,000.00'
18.
19. #WIDTH EXAMPLES
20. ##Example 1
21. print('{:5}'.format('abc'))
22. ##outputs 'abc  '
23.
24. ##Example 2
25. print('{:5}'.format(123))
26. ##outputs ' 123'
27.
28. ##Example 3
29. print('{:5.2f}'.format(123))
30. ##outputs '123.00'
31.
32. #Forcing sign to show
33. print('pi equals {:.+2f}'.format(math.pi))
34. #outputs 'pi equals +3.14'
35.
36. #ALIGNMENT EXAMPLES
37. print('{:>5}'.format('abc'))
38. ##outputs ' abc'
39.
```

```

40. print('{:<5}'.format(123))
41. ##outputs '123'
42.
43. print('{:^5}'.format(123))
44. ##outputs ' 123 '
45.
46. print('{:=+5}'.format(123))
47. ##outputs '+ 123'
48.
49. #Fill
50. '{.:^10.2f}'.format(math.pi)
51. #Note the period after the colon
52. #outputs '...3.14...' instead of ' 3.14   '
53.
54. #Percentage
55. questions = 25
56. correct = 18
57. grade = correct / questions
58. print(grade)
59. #outputs 0.72
60. print('{:.2%}'.format(grade))
61. #outputs 72.00%
62. print('{:.0%}'.format(grade))
63. #outputs 72%

```

Long Lines of Code

The [Python Style Guide²¹](#) 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, Python provides a way for breaking a statement across multiple lines by enclosing the lines in parentheses:

```

phrase = ('On a scale of {0:d} to {1:d},' +
          'I give {2:s} a {3:d}'.format(1, 5,
                                         'Monty Python', 6))

```

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

Python Strings

Code Sample

`strings/Demos/long_code_lines.py`

```
1.      #EXAMPLE 1
2.      phrase = ('On a scale of {0:d} to {1:d}, ' +
3.                  'I give {2:s} a {3:d}.').format(1, 5,
4.                                         'Monty Python', 6))
5.      print(phrase)
6.
7.      #EXAMPLE 2
8.      location = "ponds"
9.      items = "swords"
10.     beings = "masses"
11.     adjective = "farcical"
12.
13.     quote = ("Listen, strange women lyin' in {} " +
14.               "distributin' {} is no basis for a system of " +
15.               "government. Supreme executive power derives from " +
16.               "a mandate from the {}, not from some {} " +
17.               "aquatic ceremony.").format(location, items,
18.                                         beings, adjective)
19.
20.     print(quote)
```

Exercise 12 Playing with Formatting

10 to 20 minutes

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

Option 1

1. Open the Python Shell.
2. Enter '{ }'.format('') and press **Enter**.
3. Try formatting different values in different ways.

Option 2

1. Open [strings/Demos/formatter.py](#) in your editor and run the file
2. Try different format specifications with different numbers, like this:

```
>>>
Format to try: {:2f}
Number to format: math.pi
Result: 3.141593
Again? Press ENTER to try another format or 'q' to quit. q
>>> |
```

4.7 Formatted String Literals (f-strings)

Python 3.6 introduced "formatted string literals", or "f-strings". The f-string syntax merges the `format` function within the string itself. Therefore, the coding tends to be less verbose.

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

Code Sample

strings/Demos/f_strings.py

```
1. import math
2. user_name = input("What is your name? ")
3. # old way with concatenation:
4. greeting = "Hello, " + user_name + "!"
5. # or with string formatting:
6. greeting = "Hello, {}".format(user_name)
7. # new way with f-string:
8. greeting = f"Hello, {user_name}!"
9. print(greeting)
10. # format specification is also available:
11. pi_statement = f"pi is {math.pi:.4f}"
12. 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 a `f`.

Everything you learned earlier about formatting can be applied to the f-string because the same formatting function is called. In addition, string functions can be applied to the variables that you reference within the curly braces.

Exercise 13 Getting Acquainted with f-strings

10 to 20 minutes

In this exercise, you will become more familiar with f_strings.

1. You will write a Python script that uses an f_string to display the user name as it was typed in by the user and the same name in uppercase. Sample output is shown below:

```
Enter your name: Stephen
Hello, Stephen! Your name in uppercase is STEPHEN.
>>> |
```

Python Strings

Exercise Solution

`strings/Solutions/f_string_with_uppercase.py`

```
1.  def main():
2.      user_name = input("Enter your name: ")
3.      print(f"Hello, {user_name}! Your name in uppercase is {user_name.upper()}")
4.  main()
```

Note the use of the `upper` string function within the curly braces.

4.8 Built-in String Functions

str(object)

Converts object to a string.

```
str('foo') #object is string already. Returns 'foo'
str(999) #object is integer. Returns '999'
str(math.pi) #object is float. Returns '3.141592653589793'
```

len(string)

Returns the number of characters in the string²².

```
len('foo') #Returns 3
```

min() and max()²³

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

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

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

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

The `min()` and `max()` functions can also take an *iterable* containing values to compare. We will cover this in the Iterables lesson (see page 137).

22. As we will see later in the course, the `len()` function can also take objects other than strings.
 23. The `min()` and `max()` functions can also compare numbers (see page 65).

Exercise 14 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 input as tab-delimited text. The screenshot below shows the program after it has run:

```
>>>
Company: Peppermints
Revenue: 1200000
Expenses: 999002
Again? Press ENTER to add another row or Q to quit.
Company: Ni Knights
Revenue: 19
Expenses: 24
Again? Press ENTER to add another row or Q to quit.
Company: Round Knights
Revenue: 777383
Expenses: 777382
Again? Press ENTER to add another row or Q to quit. q
   Company      Revenue      Expenses      Profit
Peppermints    $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
>>> |
```

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 as money (e.g., `$1,200,000.00`) and right-aligned.
4. Save and run the file. Try entering data for at least three companies.

Python Strings

Exercise 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 += 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 = "{:<10}".format(c)
20.        r_str = "${:>10,.2f}".format(r)
21.        e_str = "${:>10,.2f}".format(e)
22.        p_str = "${:>10,.2f}".format(p)
23.
24.        new_row = f'{c_str}\t{r_str}\t{e_str}\t{p_str}\n'
25.
26.        _output += new_row
27.
28.        again = input("Again? Press ENTER to add a row or Q to quit. ")
29.        if again.lower() != 'q':
30.            add_row()
31.        else:
32.            print(_output)
33.
34.    def main():
35.        add_headers()
36.        add_row()
37.
38.    main()
```

4.9 Conclusion

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

Python Strings

5. Iterables: Sequences, Dictionaries, and Sets

In this lesson, you will learn...

1. About the different types of iterables available in Python.
2. To create, modify, and work with lists.
3. To create and work with tuples.
4. To create ranges.
5. To create, modify, and work with dictionaries.
6. To create sets and to use sets to remove duplicates in lists.
7. About the `*args` and `**kwargs` parameters.

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.

5.1 Definition

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* 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 can not be populated with objects that can be modified

5.2 Sequences

Sequences are iterables that can return members based on their position within the iterable.

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']
```

Common List Methods

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

Deleting List Elements

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

Code Sample**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, we learned about the `random` module (see page 69). We mentioned two methods that we were not yet ready to explore. Now we are:

Sequence Methods of the `random` Module

Method	Returns	Example
<code>random.choice(seq)</code>	Returns random element in seq.	<code>random.choice(['a', 'b', 'c'])</code> #returns 'a', 'b', or 'c'
<code>random.shuffle(seq)</code>	None. Shuffles seq in place.	<code>random.shuffle(['a', 'b', 'c'])</code> #shuffles list

Exercise 15 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 list `l`.
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 Solution

iterables/Solutions/remove_random.py

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

Tuples

Tuples are like lists, except that 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:

Code Sample

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 above code will render the following:

```
>>>
<class 'tuple'>
<class 'tuple'>
Traceback (most recent call last):
  File "D:\_Webucator\courseware\ClassFiles\iterables\Datas\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.`

Remember Constants

The MAGENTA variable above makes a good constant (see page 21). The values represent the amounts of red, green, and blue in the color magenta.

Creating a Single-element Tuple

To create an empty tuple, use an empty set of parentheses:

```
veggies_my_son_likes24 = ()
```

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

```
pitchers_with_500_wins = ('Cy Young',)
```

If you do not include the comma, you just get a string as illustrated below:

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

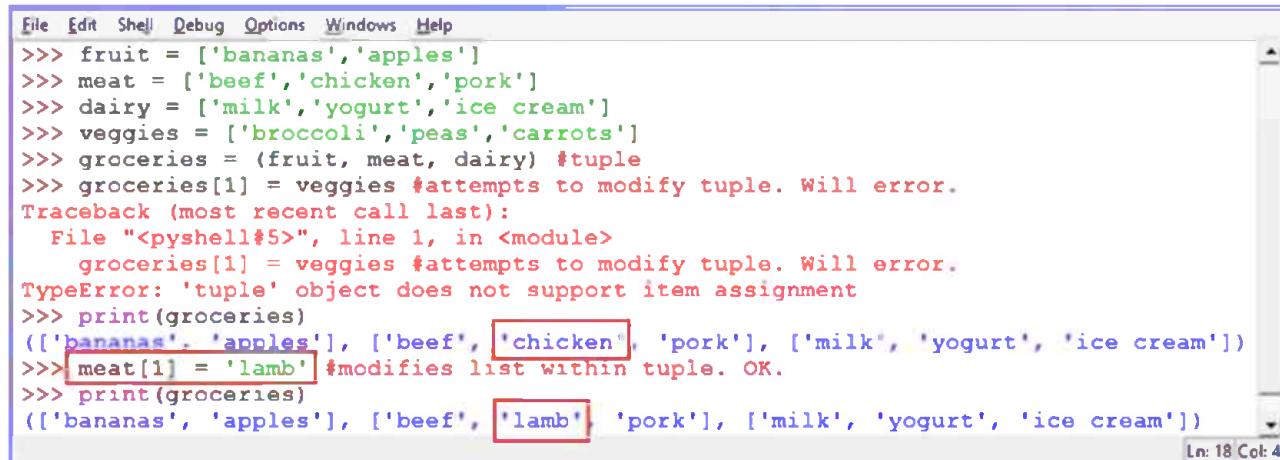
Common Tuple Methods

Method	Description
<code>mytup.count(x)</code>	Returns the number of times that <code>x</code> appears in <code>mytup</code> .
<code>mytup.copy()</code>	Returns a copy of <code>mytup</code> .
<code>mytup.index(x)</code>	Returns the index position of the first element in <code>mytup</code> whose value is <code>x</code> or a <code>ValueError</code> if no such element exists.

²⁴. My hope is that this should be a list and not a tuple. Get it? Tuples are immutable.

The Immutability of Tuples

Again, tuples are immutable. That means that you cannot add, remove, or change the elements within a tuple. This is not to say that the elements within a tuple are immutable. To illustrate, look at the following:



A screenshot of a Python shell window. The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The code entered is as follows:

```
File Edit Shell Debug Options Windows Help
>>> fruit = ['bananas', 'apples']
>>> meat = ['beef', 'chicken', 'pork']
>>> dairy = ['milk', 'yogurt', 'ice cream']
>>> veggies = ['broccoli', 'peas', 'carrots']
>>> groceries = (fruit, meat, dairy) #tuple
>>> groceries[1] = veggies #attempts to modify tuple. Will error.
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    groceries[1] = veggies #attempts to modify tuple. Will error.
TypeError: 'tuple' object does not support item assignment
>>> print(groceries)
([['bananas', 'apples'], ['beef', 'chicken', 'pork'], ['milk', 'yogurt', 'ice cream']])
>>> meat[1] = 'lamb' #modifies list within tuple. OK.
>>> print(groceries)
([['bananas', 'apples'], ['beef', 'lamb', 'pork'], ['milk', 'yogurt', 'ice cream']])
```

The line `meat[1] = 'lamb'` is highlighted with a red box. The output shows that the list within the tuple has been modified.

1. We create several lists: `fruit`, `meat`, `dairy`, and `veggies`.
2. We create a tuple, `groceries`, which contains three of our lists.
3. We attempt to change the second element of `groceries`, but that fails, because tuples are immutable.
4. We print `groceries` and see that the second element contains our `meat` list, whose second element is 'chicken'.
5. We assign 'lamb' to the second element of `meat`.
6. We print `groceries` again and see that the second element contains the modified `meat` list, whose second element is now 'lamb'.

So, again, while tuples themselves are immutable, the elements they contain can be modified

Lists vs. Tuples

Lists can be changed, but tuples are immutable. So, when to use a list and when to use a tuple? Two rules are clear:

1. Use a list if you think you might need to add, remove, or change a member at a certain index.
2. Use a tuple if you know that you never want to add, remove, or change a member at a certain index.

Beyond that, it's largely a matter of choice. However, lists are more often used to store like things. The order may be important, but the position of an element doesn't

indicate the meaning of the element. Tuples are often used when elements at different positions have different meanings. Looking again at the MAGENTA tuple:

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

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

Ranges

A range is an immutable sequence of numbers often used in for loops. Ranges are created using `range()`, which can take one, two, or three arguments:

Syntax

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

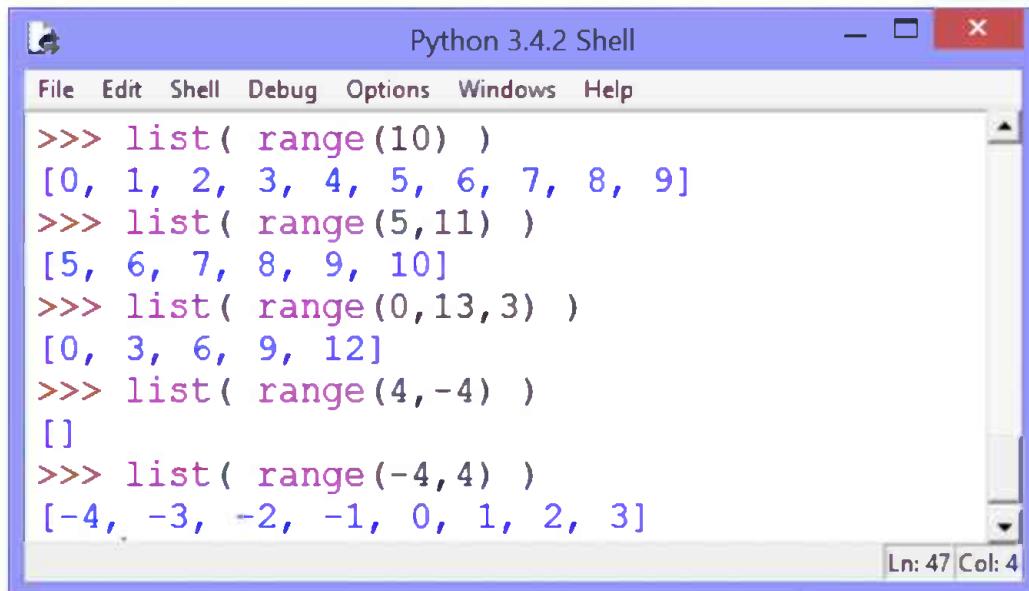
The examples below show the options for creating a range:

Iterables: Sequences, Dictionaries, and Sets

```
range(10) #creates a range starting at 0 and ending at 9
range(5,11) #creates a range starting at 5 and ending at 10
range(0,13,3) #creates a range starting at 0, ending at 12, in steps
    of 3
range(4,-4) #creates an empty range
range(4,-4,-1) #creates a 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 *end*."

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



A screenshot of the Python 3.4.2 Shell window. The menu bar includes File, Edit, Shell, Debug, Options, Windows, and Help. The main window displays the following code and its output:

```
>>> 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))
[]
>>> list(range(-4,4))
[-4, -3, -2, -1, 0, 1, 2, 3]
```

The status bar at the bottom right shows "Ln: 47 Col: 4".

We will revisit ranges when we discuss for loops. (see page 159)

Python 2 Difference

In Python 2, there is an `xrange()` function, which works exactly the same as Python 3's `range()` function.

Python 2 also has a `range()` function, but it returns a list rather than a `range` object. It is functionally similar to `xrange()`, but uses a tiny bit more memory.

Converting Sequences to Lists

Tuples and ranges (and other sequences) can be converted to lists using the `list()` function, like this:

```
list( ('a', 'b', 'c') ) #Returns ['a', 'b', 'c']
list( range(2, 5) ) #Returns [2, 3, 4]
```

Indexing and Slicing

In the Strings lesson, you learned how to find specific characters within a string using indexing and slicing (see page 81). All sequences can be indexed and sliced in this way.

Indexing

Again, indexing is the process of finding a specific element within a sequence 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

Let's look at an example.

Code Sample

iterables/Demos/indexing.py

```
1.     fruit = ['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
2.
3.     first_fruit = fruit[0]
4.     #apple
5.     print(first_fruit)
6.
7.     last_fruit = fruit[-1]
8.     #watermelon
9.     print(last_fruit)
10.
11.    fifth_fruit = fruit[4]
12.    #lemon
13.    print(fifth_fruit)
14.
15.    third_fruit_from_end = fruit[-3]
16.    #pear
17.    print(third_fruit_from_end)
```

Code Explanation

This demo uses a list, but the same can be done with any sequence type.

Exercise 16 Simple Rock, Paper, Scissors Game

15 to 20 minutes

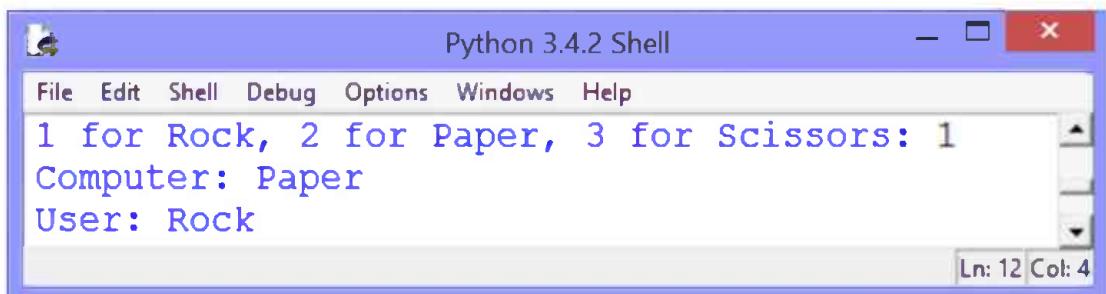
Exercise Code

[iterables/Exercises/roshambo.py](#)

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

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 screenshot below shows how the program should run:



Iterables: Sequences, Dictionaries, and Sets

Exercise 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:

Syntax

```
sub_sequence = original_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. So

`['a', 'b', 'c', 'd', 'e'] [:3]` would return `['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. So

`['a', 'b', 'c', 'd', 'e'] [3 :]` would return `['d', 'e']`.

The following demonstration shows how to get slices from a list:

Slicing Sequences

Code Sample

iterables/Demos/slicing.py

```
1.     fruit = ['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
2.
3.     first_5_fruit = fruit[0:5]
4.     #['apple', 'orange', 'banana', 'pear', 'lemon']
5.     print(first_5_fruit)
6.
7.     fruit_2_thru_4 = fruit[1:4]
8.     #['orange', 'banana', 'pear']
9.     print(fruit_2_thru_4)
10.
11.    fruit_5_to_end = fruit[4:]
12.    #['lemon', 'watermelon']
13.    print(fruit_5_to_end)
14.
15.    last_3_fruit = fruit[-3:]
16.    #['pear', 'lemon', 'watermelon']
17.    print(last_3_fruit)
18.
19.    first_3_fruit = fruit[:3]
20.    #['apple', 'orange', 'banana']
21.    print(first_3_fruit)
22.
23.    three_fruit_before_last = fruit[-4:-1]
24.    #['banana', 'pear', 'lemon']
25.    print(three_fruit_before_last)
26.
27.    copy_of_string = fruit[:]
28.    #['apple', 'orange', 'banana', 'pear', 'lemon', 'watermelon']
29.    print(copy_of_string)
```

Code Explanation

Again, this demo uses a list, but the same can be done with any sequence type.

Exercise 17 Slicing Sequences

10 to 20 minutes

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()
```

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. or 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. or 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 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]
9.
10.    def main():
11.        colors = ['red','blue','green','orange','purple']
12.        colors_split = split_list(colors)
13.        print(colors_split[0])
14.        print(colors_split[1])
15.
16.    main()
```

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.

Code Sample

`iterables/Demos/min_and_max.py`

```
1. def main():
2.     colors = ['red', 'blue', 'green', 'orange', 'purple']
3.     print( min(colors) )
4.
5.     ages = (27, 4, 15, 99, 33, 25)
6.     print( max(ages) )
7.
8. main()
```

Code Explanation

The above code will render the following:

```
>>>
blue
99
>>> |
```

Why blue? Because 'b' comes before 'r', 'g', 'o', and 'p' in the alphabet.

The `min()` and `max()` functions can also take a list of arguments to compare. This is covered in the Math (see page 65) and Strings (see page 111) 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 = [1,2,3,4,5]
sum(nums) #returns 15

nums = [1,2,3,4,5]
sum(nums, 10) #returns 25
```

Converting Sequences to Strings with `str.join(seq)`

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

```
colors = ['red','blue','green','orange']
','.join(colors) #returns 'red,blue,green,orange'
':'.join(colors) #returns 'red:blue:green:orange'
' '.join(colors) #returns 'red blue green orange'
```

Splitting Strings into Lists

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

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

`split()` takes an optional `sep` argument to indicate the separator. By default, it will separate on whitespace.

```
fruit = 'apple, banana, pear, melon'
list_of_fruit = fruit.split(',')
#returns ['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'
list_of_fruit = fruit.split(', ')
#returns ['apple', 'banana', 'pear', 'melon']
```

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

```
fruit = 'apple, banana, pear, melon'
list_of_fruit = fruit.split(',', 2)
#returns ['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; for example, line feeds (\n) and carriage returns (\r). For example:

```
fruit = '''apple
banana
pear
melon'''
list_of_fruit = fruit.splitlines()
#returns ['apple', 'banana', 'pear', 'melon']
```

5.3 Unpacking Sequences

We learned earlier about simultaneous assignment (see page 15), which allows you to assign values to multiple variables at once, like this:

```
smart_1, cute_1, quiet_1, funny_1 = "John", "Paul", "George", "Ringo"
```

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

```
beatles = ["John", "Paul", "George", "Ringo"]
smart_1, cute_1, quiet_1, funny_1 = beatles
print(cute_1) #Paul
```

5.4 Dictionaries

Dictionaries are mappings that use arbitrary keys to map to values. They are like associative arrays in other programming languages. Dictionaries are created by surrounding key/value pairs in curly brackets and separating them with commas, like this:

Iterables: Sequences, Dictionaries, and Sets

Syntax

```
dict = {key1:value, key2:value, key3:value,...}
dict[key2] = new_value #assign new value to existing key
dict[key4] = value #assign value to new key
print(dict[key1]) #print value of key
```

Code Sample

iterables/Demos/dict.py

```
1.     grades = {'English':97, 'Math':93, 'Global Studies':85, 'Art':74, 'Mu >>>
2.
3.     grades['Global Studies'] = 87 #assign new value to existing key
4.
5.     grades['Gym'] = 100 #assign value to new key
6.
7.     print(grades['Math']) #print value of key
```

Common Dictionary Methods

Method	Description
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.
mydict.pop(key [,default])	Removes and returns key if it is in mydict. Otherwise, it returns default if passed in or a KeyError if it is not.
mydict.popitem()	Removes and returns a random key/value pair as a tuple.
mydict.copy()	Returns a copy of mydict.
mydict.clear()	Removes all elements from mydict.

The update() Method

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:

Updating with a Dictionary

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
grades.update({'Math':97, 'Gym':93}) #changes Math, adds Gym
#{'Math': 97, 'Music': 86, 'Art': 74, 'Gym': 93, 'English': 97}
```

Updating with Keyword Arguments

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
grades.update(Math=97, Gym=93) #changes Math, adds Gym
#{'Math': 97, 'Music': 86, 'Art': 74, 'Gym': 93, 'English': 97}
```

Updating with an Iterable of Key/Value Pairs

```
grades = {'English':97, 'Math':93, 'Art':74, 'Music':86}
grades.update( [ ('Math',97), ('Gym',93) ] ) #changes Math, adds Gym
#{'Math': 97, 'Music': 86, 'Art': 74, 'Gym': 93, 'English': 97}
```

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 (e.g., `grades['Math'] = 97`).

The `setdefault()` Method

The `setdefault(key, default)` method is an oddly named method. It 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:

Code Sample**iterables/Demos/setdefault.py**

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

Code Explanation

The above code will render the following:

```
>>>
Art grade: 74
Gym grade: 97
>>> |
```

Dictionary View Objects

The following three methods return dictionary view objects:

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

Examine the following:

```
>>> grades = {'English':97, 'Math':93, 'Art':74}
>>> grades.keys()
dict_keys(['Art', 'English', 'Math'])
>>> grades.values()
dict_values([74, 97, 93])
>>> grades.items()
dict_items([('Art', 74), ('English', 97), ('Math', 93)])
```

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:

Code Sample**iterables/Demos/dict_views.py**

```

1.     grades = {'English':97, 'Math':93, 'Global Studies':85, 'Art':74, 'Mu >>>
2.             sic':86}
3.     gradepoints = grades.values()
4.     print('Grade points:', gradepoints)
5.
6.     grades['Art'] = 87
7.     print('Grade points:', gradepoints)

```

Code Explanation

The output:

```

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

```

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.

Deleting Dictionary Keys

The `del` statement can be used to delete a specific key from a dictionary, like this:

Code Sample**iterables/Demos/del_dict.py**

```

1.     grades = {'English':97, 'Math':93, 'Global Studies':85, 'Art':74, 'Mu >>>
2.             sic':86}
3.     del grades['Math'] #deletes Math key
4.     print(grades) #{'English':97, 'Global Studies':85, 'Art':74, 'Music':86}

```

Python 2 Difference

In Python 2, the `keys()`, `values()`, and `items()` methods of dictionaries return lists, not views.

Python 2.7 implements view objects, but they are retrieved with the `viewkeys()`, `viewvalues()`, and `viewitems()` methods, all of which have been discontinued in Python 3.

5.5 The `len()` Function

The `len()` function can be used to determine the number of characters in a string or 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, 'Art': 74, 'English': 97}) #5
```

Exercise 18 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 screenshot below shows how the program should run:

```
>>>
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.

Iterables: Sequences, Dictionaries, and Sets

Exercise 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.         newgrade = input('What is your new ' + subject + ' grade? ')
21.         grades[subject] = int(newgrade)
22.         average = avg(gradepoints)
23.
24.         print('Your new average is', average)
25.
26.     main()
```

5.6 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 brackets, just like dictionaries:

```
classes = {'English', 'Math', 'Global Studies', 'Art', 'Music'}
```

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:

Code Sample

iterables/Demos/remove_dups.py

```
1. def main():
2.     veggies = ['tomato', 'spinach', 'pepper', 'pea', 'tomato', 'pea']
3.     v_set = set(veggies) #turn list into set removing dups
4.     print(v_set) #{'pepper', 'tomato', 'spinach', 'pea'}
5.
6.     veggies = list(v_set) #turn set back into a list
7.     print(veggies) #['pepper', 'tomato', 'spinach', 'pea']
8.
9. main()
```

5.7 *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 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 in the course, 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:

Code Sample

`iterables/Demos/add_nums.py`

```
1.  def add_nums(num, *nums):  
2.      total = sum(nums, num)  
3.      str_nums = [str(i) for i in nums]  
4.      print(num, '+', ' + '.join(str_nums), '=', total)  
5.  
6.  def main():  
7.      add_nums(1,2)  
8.      add_nums(1,2,3,4,5)  
9.      add_nums(11,12,13,14)  
10.     add_nums(101,201,301)  
11.  
12.  main()
```

Code Explanation

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 137) to get the sum of `nums` and add it to `num`.
3. We then use a list comprehension to create a new list of strings from the list of integers so that we can join the passed-in numbers on a plus sign for printing.

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.

5.8 Conclusion

In this lesson, you have learned about lists, tuples, ranges, dictionaries, and sets. You also learned about the `*args` and `**kwargs` parameters. In the next lesson, you'll learn to search these iterables for values and to loop through them performing operations on each element they contain one by one.

6. Flow Control

In this lesson, you will learn...

1. To work with if conditions in Python.
2. To work with loops in Python.
3. To create generator functions.
4. To work with list comprehensions.

Generally, 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.

6.1 Conditional Statements

Conditional statements (`if`-`elif`-`else` conditions) allow programs to output different code based on specific conditions. The syntax is shown below:

```
Syntax
if statement
if some_conditions:
    do_this_1()
    do_this_2()
do_this_after()
```

Notice that the `do_this_1()` and `do_this_2()` functions 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`.

```
Syntax
if-else statement
if some_conditions:
    do_this_1()
    do_this_2()
else:
    do_this_3()
do_this_after()
```

```
Syntax
if-elif-else statement

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()
```

The two syntax blocks above show an `if-else` and an `if-elif-else` statement, which can have any number of `elif` blocks, but only zero or one `else` blocks.

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 ²⁵
<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

The following table shows Python's membership operators.

Membership Operators

Operator	Description	Example
<code>in</code>	True if value is found in sequence.	<code>'a' in ['a', 'b', 'c'] #True</code> <code>'a' in 'abc' #True</code>
<code>not in</code>	True if value is not found in sequence.	<code>'a' not in ['a', 'b', 'c'] #False</code> <code>'d' not in 'abc' #True</code>

The following example demonstrates an `if-elif-else` statement.

Code Sample

`if_w-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()

```

25. Python 2 also supports `<>` as a not equals operator, but `!=` is preferred.

Code Explanation

The file is relatively simple. You can see the different results by running it and entering different ages at the prompt.

Compound Conditions

More complex if statements often require that several conditions be checked. The table below shows *and* 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.

Code Sample

`fl w-control/Demos/if2.py`

```
1.  def main():
2.      age = int(input('How old are you? '))
3.
4.      if input('Are you a citizen? Y or N ').lower() == 'y':
5.          is_citizen = True
6.      else:
7.          is_citizen = False
8.
9.      if age >= 21 and is_citizen:
10.         print('You can vote and drink.')
11.     elif age >= 21:
12.         print('You can drink, but can\'t vote.')
13.     elif age >= 18 and is_citizen:
14.         print('You can vote, but can\'t drink.')
15.     else:
16.         print('You cannot vote or drink.')
17.
18. main()
```

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 example illustrates this:

Code Sample

f1 w-control/Demos/is_and_is_not.py

```

1. def main():
2.     a = [1,2,3,4]
3.     b = [1,2,3,4]
4.     c = a
5.     print('a == b:', a == b)
6.     print('a is b:', a is b)
7.     print('a == c:', a == c)
8.     print('a is c:', a is c)
9.
10.    a.append(5)
11.    print(c)
12.
13. main()

```

Code Explanation

The above code will render the following:

```

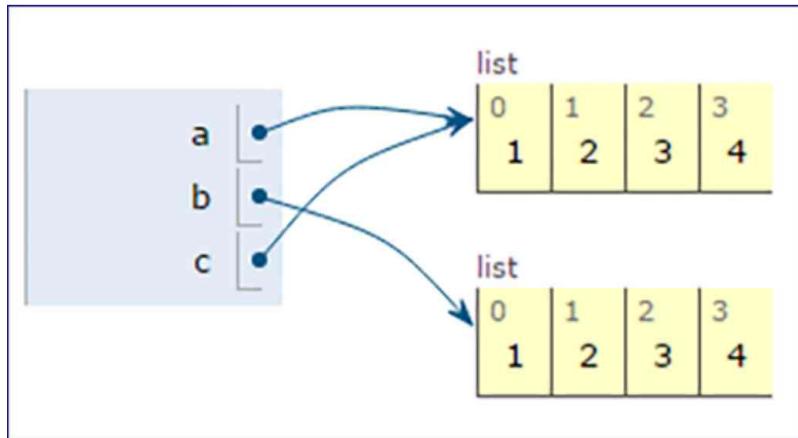
a == b: True
a is b: False
a == c: True
a is c: True
[1, 2, 3, 4, 5]

```

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. That's 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 of the program after line 4, after all the variables have been assigned, but before 5 was appended to a. As you can see, a and c point to the same object:



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()
```

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

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 game_type == 'home':
    shirt = 'white'
else:
    shirt = 'green'
```

That's very clear, but it takes four lines of code to assign a value to `game_type`.

The syntax for the ternary operator in Python is:

Syntax

```
[on_true] if [expression] else [on_false]
```

Here is how we could use this syntax to assign our shirt color:

```
shirt = 'white' if game_type == 'home' else 'green'
```

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`.

6.2 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. All of these examples are in [flow-control/Deimos/loops.py](#).

while Loops

`while` loops are used to execute a block of code repeatedly while one or more conditions are true.

```
while Loop

num=0
while num < 6:
    print(num)
    num += 1
```

The above code will return:

Flow Control

```
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 example below illustrates this:

Code Sample

fl w-control/Demos/while_input.py

```
1.  def is_valid_age(s):  
2.      if s.isdigit() and 1 <= int(s) <= 113:  
3.          return True  
4.      else:  
5.          return False  
6.  
7.  def main():  
8.      age = input('How old are you? ')  
9.      while not is_valid_age(age):  
10.         age = input('Please enter a real age as a number: ')  
11.  
12.     age = int(age)  
13.     if age >= 21:  
14.         print('You can vote and drink.')  
15.     elif age >= 18:  
16.         print('You can vote, but can\'t drink.')  
17.     else:  
18.         print('You cannot vote or drink.')  
19.  
20. main()
```

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. Spend a little time reviewing the code and then your instructor will walk you through it.

Code Sample**fl w-control/Demos/guess_the_number.py**

```

1.      import random
2.
3.      def is_valid_num(s):
4.          if s.isdigit() and 1 <= int(s) <= 100:
5.              return True
6.          else:
7.              return False
8.
9.      def main():
10.         number = random.randint(1,100)
11.         guessed_number = False
12.         guess = input("Guess a number between 1 and 100: ")
13.         num_guesses = 0
14.         while not guessed_number:
15.             if not is_valid_num(guess):
16.                 guess = input("I won't count that one. A number between 1
17. >>> and 100 please: ")
18.                 continue
19.             else:
20.                 num_guesses +=1
21.                 guess = int(guess)
22.
23.                 if guess < number:
24.                     guess = input("Too low. Guess again: ")
25.                 elif guess > number:
26.                     guess = input("Too high. Guess again: ")
27.                 else:
28.                     print("You got it in",num_guesses,"guesses!")
29.                     guessed_number = True
30.
31.         print("Thanks for playing.")
32.     main()

```

for Loops

A for loop is used to loop through an iterator (e.g., a range, list, tuple, dictionary, etc.).

Flow Control

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.

Looping through a Range with a Step

```
for num in range(1,11,2) :  
    print(num)
```

This loop will return:

```
1  
3  
5  
7  
9
```

Looping through a Range with a Negative Step

```
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 or Tuple

```
nums = [0,1,2,3,4,5] #list
for num in nums:
    print(num)

nums = (0,1,2,3,4,5) #tuple
for num in nums:
    print(num)
```

Both loops above will return:

```
0
1
2
3
4
5
```

Looping through a Dictionary

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

for course in grades:
    print(course)

for course in grades.keys():
    print(course)

for grade in grades.values():
    print(grade)

for course, grade in grades.items():
    print(course, ': ', grade, sep='')
```

The first two loops above will return the courses (the keys) in no particular order. A possible result is:

```
Music
Art
Math
English
```

The third loop will return the grades (the values) in no particular order. A possible result is:

Flow Control

```
86  
74  
93  
97
```

The fourth loop will return the courses (the keys) and grades (the values) in no particular order. A possible result is:

```
Music: 86  
English: 97  
Art: 74  
Math: 93
```

Exercise 19 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.

Exercise Code

`fl_w-control/Exercises/all_and_any.py`

```
1.  def all_true(iterable):
2.      #write function
3.
4.  def any_true(iterable):
5.      #write function
6.
7. def main():
8.     a = all_true( [1, 0, 1, 1, 1] )
9.     b = all_true( [1, 1, 1, 1, 1] )
10.    c = any_true( [0, 0, 0, 1, 1] )
11.    d = any_true( [0, 0 ,0 ,0, 0] )
12.
13.    print(a, b, c, d) #Should be: False True True False
14.
15. main()
```

1. Open `fl_w-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 Solution

fl_w-control/Solutions/all_and_any.py

```
1.      def all_true(iterable):
2.          for elem in iterable:
3.              if not elem:
4.                  return False
5.          return True
6.
7.      def any_true(iterable):
8.          for elem in iterable:
9.              if elem:
10.                  return True
11.          return False
12.
13.     def main():
14.         a = all_true( [1, 0, 1, 1, 1] )
15.         b = all_true( [1, 1, 1, 1, 1] )
16.         c = any_true( [0, 0, 0, 1, 1] )
17.         d = any_true( [0, 0 ,0 ,0, 0] )
18.
19.         print(a, b, c, d) #Should be: False True True False
20.
21.     main()
```

6.3 break and continue

To break out of a loop, insert a `break` statement:

```
for num in range(6):
    print(num)
    if num > 3:
        break
```

To jump to the next iteration of a loop without executing the remaining statements in the block, insert a `continue` statement:

Syntax

```
for num in range(6):
    if num==3:
        continue
    print(num)
```

Exercise 20 Word Guessing Game

45 to 60 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.

Exercise Code

fl w-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))
>>>

```

Flow Control

```
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 [fl_w-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.' (e.g., '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: ' (e.g., 'Guess a letter or a 6-letter word: ').
4. For each guess:
 - A. **If the user has already guessed that letter or word**, the program prints 'You already guessed "guess".' (e.g., '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 a sequence of 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.' (e.g., '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 a sequence of 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 the sequence of 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 the sequence of guesses shows up in the secret word:
 - a. **Multiple times**: 'The word has n "guess"s.' (e.g., 'The word has 2 "O"s.').
 - b. **Exactly one time**: 'The word has one "guess".' (e.g., 'The word has one "S".').
 - c. **Zero times**: 'Sorry. The word has no "guess"s.' (e.g., '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. 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." (e.g., '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., '**OO*Y').
 - D. **Otherwise**, the program prints 'Invalid Entry' and prompts the user for another guess.

The screenshot below shows a completed game:

```
>>>
The word contains 6 letters.
Guess a letter or a 6-letter word: A
Sorry. The word has no "A"s.
*****
Guess a letter or a 6-letter word: O
The word has 2 "O"s.
**OO**
Guess a letter or a 6-letter word: GOOFY
Invalid entry.
Guess a letter or a 6-letter word: GOOFFY
Sorry, that is incorrect.
Guess a letter or a 6-letter word: A
You already guessed "A".
Guess a letter or a 6-letter word: Y
The word has one "Y".
**OO*Y
Guess a letter or a 6-letter word: Snoopy
SNOOPY is it! It took 5 tries.
>>>
```


Exercise Solution

fl_w-control/Solutions/guess_word.py

```
-----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.         >>>
41.         guess = guess.upper()
42.         if guess in guesses:
43.             print('You already guessed "{}.'.format(guess))
44.         elif len(guess) == n: #Guessing whole word
45.             guesses.append(guess)
46.             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()
```

The `else` Clause

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 demo below shows how it works:

Code Sample**fl w-control/Demos/loop-else.py**

```
1.      def main():
2.          print('Example 1: for loop')
3.          for i in range(4):
4.              print(i)
5.          else:
6.              print('Completed iterating.')
7.
8.          print('\nExample 2: for loop with break')
9.          for i in range(4):
10.              if i == 2:
11.                  break
12.              print(i)
13.          else:
14.              print('Completed iterating.')
15.
16.          print('\nExample 3: while loop')
17.          i = 0
18.          while i <= 4:
19.              print(i)
20.              i += 1
21.          else:
22.              print('Completed iterating.')
23.
24.          print('\nExample 4: while loop with break')
25.          i = 0
26.          while i <= 4:
27.              if i == 2:
28.                  break
29.              print(i)
30.              i += 1
31.          else:
32.              print('Completed iterating.')
33.
34.  main()
```

Code Explanation

The above code will render the following:

```
>>>
Example 1: for loop
0
1
2
3
Completed iterating.

Example 2: for loop with break
0
1

Example 3: while loop
0
1
2
3
4
Completed iterating.

Example 4: while loop with break
0
1
>>> |
```

Notice that examples 1 and 3 end with "Completed iterating", which is output in the `else` clause. Examples 2 and 4 end with a `break` statement, so the code in the `else` block doesn't run.

So, when would you use this? Let's say we have a list of people and we want to know if anyone in the list is awesome, according to our magic (and completely imaginary) `is_awesome()` function. The code below shows a common way to do this in other programming languages:

```
found_awesomeness = False
for person in people:
    if is_awesome(person):
        found_awesomeness = True
        break

if not found_awesomeness:
    print('There is no awesomeness in this group.')
else:
    celebrate_awesomeness()
```

But in Python, we can use the `else` clause, like this:

```
for person in people:
    if is_awesome(person):
        celebrate_awesomeness()
        break
else:
    print('There is no awesomeness in this group.)
```

While using the `else` clause is not functionally better than the method shown in the first example, it is cleaner, and it saves us from having to create the `found_awesomeness` flag.

Exercise 21 Find the Needle

10 to 20 minutes

In this exercise, you will rewrite a function to use the `else` clause of a loop.

1. Open `fl_w-control/Exercises/find_needle.v` in your editor and review the code. Then run the script to see how it works.
2. Modify the `search()` 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**fl w-control/Exercises/find_needle.p**

```

1. import random
2.
3. def is_sharp(x):
4.     if x=='needle' or x=='Albert Einstein':
5.         return True
6.     else:
7.         return False
8.
9. def create_haystack():
10.    haystack = []
11.    haystack.append('needle')
12.    for i in range(100):
13.        haystack.append('piece of hay')
14.    return haystack
15.
16. def search_for_needle(haystack, n=5):
17.     searches = []
18.     for i in range(n):
19.         searches.append(random.choice(haystack))
20.     return searches
21.
22. def search(searches):
23.     needle_found = False
24.     for x in searches:
25.         if is_sharp(x):
26.             needle_found = True
27.             break
28.
29.     if(needle_found):
30.         print("Found needle!") #Albert wouldn't be in a haystack
31.     else:
32.         print("No luck! Searching for a needle in a haystack?")
33.
34. def main():
35.     haystack = create_haystack()
36.     searches = search_for_needle(haystack)
37.     search(searches)
38.     if input('Try again: (y/n) ').lower() == 'y':
39.         main()

```

Flow Control

```
40.     else:  
41.         print('Goodbye!')  
42.  
43. main()
```


Exercise Solution

fl_w-control/Solutions/find_needle.p

```
-----Lines 1 through 21 Omitted-----
22. def search(searches):
23.     for x in searches:
24.         if is_sharp(x):
25.             print("Found needle!") #Albert wouldn't be in a haystack
26.             break
27.         else:
28.             print("No luck! Searching for a needle in a haystack?")
-----Lines 29 through 39 Omitted-----
```

6.4 The enumerate() Function

It is common in other programming languages to write code like this:

Code Sample

fl w-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:

Code Sample

fl w-control/Demos/with_enum.py

```
1.     for item in enumerate(['a', 'b', 'c'], 1):
2.         print(item[0], item[1], 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)`.

6.5 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:

Code Sample

fl w-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 the following:

```
>>>
First time through:
45
25
81
39
68
Second time through:
45
25
81
39
68
>>> |
```

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:

Code Sample

fl w-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 the following:

```
>>>
First time through:
15
88
19
11
72
Second time through:
>>>
```

This time, the `get_rand_nums()` function returns a generator. The function works in essentially the same way but 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:

Code Sample

fl w-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

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.

Generator Use Case: Randomly Moving Object

Imagine you were creating a game, in which a meteor randomly moved around the screen. You could use a generator to change the meteor's x, y position every 0.2 seconds, like this:

Code Sample**fl w-control/Demos/jiggle.py**

```
1. import random
2. from time import sleep
3.
4. def jiggle():
5.     x=0
6.     y=0
7.     while True:
8.         x_change = random.randint(-1,1)
9.         y_change = random.randint(-1,1)
10.        x += x_change
11.        y += y_change
12.        yield (x,y)
13.
14. for i in jiggle():
15.     print(i)
16.     sleep(.2)
```

Code Explanation

Run this to see how `i` changes. This code uses the `time` module's `sleep()` method to create the 0.2 second delay.

The `next()` Function

To be an iterator, an object must have a special `__next__()` method, which returns the next item of the iterator. Python's built-in `next(iter)` function calls the `__next__()` method of `iter`. The example below shows how to use the `next()` function with a generator to play Rock, Paper, Scissors (RoShamBo) against your computer.

Code Sample**fl w-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') or
14.          (your_weapon == 'Paper' and python_weapon == 'Rock') or
15.          (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.
20.
21. def make_choice():
22.     choice = input("""Choose your weapon:
23. 1: Rock
24. 2: Paper
25. 3: Scissors
26. """)
27.     return choice
28.
29. def main():
30.     weapons = ['Rock', 'Paper', 'Scissors']
31.     python_weapons = roshambo(weapons)
32.     choice = make_choice()
33.     while choice in ['1', '2', '3']:
34.         play(weapons, choice, python_weapons)
35.         choice = make_choice()
36.     print('Goodbye!')
```

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.

Note that this 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]
```

Exercise 22 Rolling Dice

10 to 20 minutes

In this exercise, you will rewrite a dice simulation script to use a generator.

1. Open `fl_w-control/Exercises/rolls.py` in your editor. Notice the three-part `if` condition. Pretty cool, right?
2. Run the program and continue pressing **Enter** to roll until you roll triples. Then press **Ctrl+C** to quit.
3. Now modify the program:
 - A. Replace the `roll_dice()` function with a `dice_rolls()` generator function that infinitely yields tuples of three random integers between 1 and 6.
 - B. Change the `play()` function to use a `for` loop to iterate through the generator.
 - C. Rather than holding the count in a local variable, use the `enumerate()` function in the `for` loop.

Exercise Solution

fl w-control/Solutions/rolls.py

```
1.      from random import randint
2.
3.      def dice_rolls():
4.          while True:
5.              yield (randint(1,6), randint(1,6), randint(1,6))
6.
7.      def play():
8.          for r in enumerate(dice_rolls(),1):
9.              print('{}, {}, {}'.format(r[0], r[1][0], r[1][1], r[1][2]))
10.             if r[1][0] == r[1][1] == r[1][2]:
11.                 print('Wow! Triples!')
12.             input()
13.
14.     def main():
15.         input('Press ENTER to roll the dice. ')
16.         play()
17.
18.     main()
```

Code Explanation

Note that the version with the generator isn't necessarily better than the original version. It's just a different way of accomplishing the task.

6.6 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. For example, in the following demo, we use a list comprehension to create a list of three-letter words from an existing list of words:

Code Sample

fl w-control/Demos/list_comp1.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()

```

And in the following demo, we use a list comprehension to create a list of people's initials from a list of their names:

Code Sample

fl w-control/Demos/list_comp2.py

```

1.  def initials(name):
2.      fullname = name.split(' ')
3.      initials = (fullname[0][0], fullname[1][0])
4.      return initials
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(i[0] + '.' + i[1] + '.')
12.
13. main()

```

6.7 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.

7. Virtual Environments

In this lesson, you will learn...

1. What a virtual environment is.
2. How to create a virtual environment.
3. How to activate and deactivate a virtual environment.
4. How to delete a virtual environment.

A virtual environment is a container for a Python development platform on your computer. Packages can be installed in the virtual environment that are separate from your standard, or non-virtual environment.

7.1 Virtual Environment

A virtual environment gives a development team its own Python platform. Scripts can be run in a virtual environment that potentially have dependencies that are different from other development projects that may be running in separate virtual environments. For example, your team might be working on a Human Resources project that has dependencies that are distinct from an Order Entry application.

A Python script can determine if it is operating within a particular virtual environment. The script can then determine the type of processing to perform. You will see an example in the exercise at the end of the lesson.

If you are using Anaconda, these instructions for using Virtual Environments very likely will not work for you. Anaconda has its own way of managing virtual environments, which is very well documented at <https://conda.io/docs/user-guide/tasks/manage-environments.html>.

Creating a Virtual Environment

To create a virtual environment you will use the `venv` module that is already installed with Python 3. You should also create a new directory to serve as the file system for your new virtual environment.

1. Create a directory named /PythonVirtualEnv.

Virtual Environments

2. Navigate to the directory you just created in a Terminal window or Command Prompt. Then type the following command and press **Enter**:

```
python -m venv myVirtualEnv
```

where `myVirtualEnv` is the name we are assigning to the virtual environment.

3. View the directory contents of `/PythonVirtualEnv`. You will observe a folder named `myVirtualEnv` that has been created by the `venv` module. The folder contains several subfolders including `Scripts` (Windows) or `bin` (Mac) that contains the `python` executable file and `Lib` (Windows) or `lib` (Mac) that contains the packages available to the virtual environment.

Activating and Deactivating a Virtual Environment

To work within your virtual environment, you must first activate the virtual environment.

Windows

In a Command Prompt, navigate to `/PythonVirtualEnv`. Type in the following command and press **Enter**:

```
./myVirtualEnv/Scripts/activate
```

Mac

In a Terminal window, navigate to `/PythonVirtualEnv`. Type in the following command and press **Enter**:

```
source myVirtualEnv/bin/activate
```

Notice the name of your virtual environment, enclosed in parentheses, appears next to the prompt:

```
PS C:\PythonVirtualEnv> ./myVirtualEnv/Scripts/activate  
(myVirtualEnv) PS C:\PythonVirtualEnv>
```

You can now invoke the Python interpreter or install packages using `pip` within the virtual environment. The executable files are resolved within the virtual environment because the `$PATH` environment variable has been modified. Let's take a look at the value of this environment variable.

Windows (using PowerShell)

\$Env:PATH:

The path of the `Scripts` directory has been prepended to the list of paths in the `$PATH` environment variable:

```
(m_VirtualEnv) PS C:\PythonVirtualEnv> $Env:PATH  
C:\PythonVirtualEnv\myVirtualEnv\Scripts;C:\app\admin\product\12.1.0\dbhome_1;C:\app\Stephen\product\11.2.0\dbhome_1\bin;C:\Java8\bin;C:\ProgramData\Orac
```

At the standard Windows Command Prompt, you would enter:

```
echo %PATH%
```

Mac Terminal

```
echo $PATH
```

The path of the `bin` directory has been prepended to the list of paths in the `$PATH` environment variable:

```
[myVirtualEnv] Nat's MacBook Pro:learning natdunn$ echo $PATH  
/Users/natdunn/Documents/personal/learning/myVirtualEnv/bin:/Library/Frameworks/  
Python.framework/Versions/3.7/bin:/Users/natdunn/.local/bin:/usr/local/bin:/usr/  
local/mysql/bin:/Users/natdunn/anaconda3/bin:/usr/local/bin:/usr/bin:/bin:/usr/  
bin:/sbin
```

Therefore the Scripts or bin directory will be scanned first in order to resolve references to executable files. Once you deactivate the virtual environment, the Scripts and bin directory will be removed from \$PATH.

To deactivate (exit) the virtual environment type the following and press **Enter**:

deactivate

Now you are back in the original prompt.

Deleting a Virtual Environment

You delete a virtual environment by using operating system commands to delete the folder (e.g., `myVirtualEnv`) that was automatically built when you created the virtual environment. Before deleting the folder, copy any scripts and/or files you wish to save.

Exercise 23 Working with a Virtual Environment

15 to 25 minutes

In this exercise you will create, activate, deactivate and delete a virtual environment.

1. In a Terminal window or the Command Prompt, navigate to [./virtual-environments/Exercises/check_virtual_env.py](#).
2. Create a virtual environment named `exerciseVirtualEnv`:

```
python -m venv exerciseVirtualEnv
```

3. Take a few moments to examine the folder structure created as a result of executing `venv`.
4. Activate the virtual environment:

Windows

```
./exerciseVirtualEnv/Scripts/activate
```

or

Mac

```
source exerciseVirtualEnv/bin/activate
```

5. You can inspect an environment variable named `$VIRTUAL_ENV` to verify you are within the virtual environment:

Windows PowerShell

```
$Env:VIRTUAL_ENV
```

or

Windows Command Prompt

```
echo %VIRTUAL_ENV%
```

or

Mac

```
echo $VIRTUAL_ENV
```

The value of the environment variable will be displayed and will contain [/exerciseVirtualEnv](#).

6. We can also check to see if `$VIRTUAL_ENV` is set in a Python script. The script [./virtual-environments/Exercises/check_virtual_env.py](#) performs a simple check to determine if the script is operating in a virtual environment. Run the script now from the Terminal window or Command Prompt:

```
python check_virtual_env.py
```

The output will echo the value of the `$VIRTUAL_ENV`.

7. Deactivate the virtual environment.
8. Run `./virtual-environments/Exercises/check_virtual_env.py` again from the Terminal window or Command Prompt. The output will be:

```
We are NOT in a virtual environment!
```

9. Inspect `$VIRTUAL_ENV` as you did earlier from the command line. The value of the environment variable will be empty.
10. Using operating system commands, delete the virtual environment by removing `exerciseVirtualEnv`.

Exercise Code

virtual-environments/Exercises/check_virtual_env.py

```
1. import os
2. try:
3.     virtual_env = os.environ["VIRTUAL_ENV"]
4.     print (f"Virtual environment variable: {virtual_env}")
5. except KeyError:
6.     print ("We are NOT in the virtual environment!")
```

Code Explanation

The script attempts to access the key `$VIRTUAL_ENV` in the `os.environ` dictionary. If the key does not exist (i.e., the environment variable is not set), then a `KeyError` results.

7.2 Conclusion

In this lesson, you have learned the purpose of a virtual environment and how to create a virtual environment. You have also learned how to activate, deactivate, and delete a virtual environment.

8. Regular Expressions

In this lesson, you will learn...

1. How to create regular expressions for pattern matching.
2. How to use regular expressions within Python.

Regular expressions are used to do pattern matching in many programming languages, including, Java, PHP, JavaScript, C, C++, and Perl. We will provide a brief introduction to regular expressions and then we'll show you how to work with them in Python.

8.1 Regular Expression Syntax

Regular Expression Tester

We will use the online RE testing tool at <https://pythex.org/> to demonstrate and test our regular expressions.

Start and End (^ \$)

A caret (^) at the beginning of a regular expression indicates that the string being searched must start with this pattern.

- The pattern `^foo` can be found in "food", but not in "barfood".

A dollar sign (\$) at the end of a regular expression indicates that the string being searched must end with this pattern.

- The pattern `foo$` can be found in "curfoo", but not in "food".

Word Boundaries (\b \B)

Backslash-b (\b) denotes a word boundary. It matches a location at the beginning or end of a word (a sequence of alphanumeric characters).

- The pattern `foo\b` can be found in "curfoo", but not in "food" or "foobar".

Backslash-B (\B) is the opposite of backslash-b (\b). It matches a location that is **not** a word boundary.

- The pattern `foo\B` can be found in "food" and "foobar", but not in "curfoo".

Try it

1. Open the online testing tool available at <https://pythex.org/>.
2. Under **Your regular expression**, type:

```
^foo
```

3. Be sure to select **MULTILINE** so that each line is evaluated individually.
4. Under **Your test string**, enter:

```
food
barfood
```

5. The page should look something like this:

The screenshot shows the pythex website. At the top, the word "pythex" is displayed in a large, bold, dark font. To its right is a link labeled "Link to this regex". Below the title, there are two input fields: "Your regular expression:" containing the pattern `^foo` and "Your test string:" containing the text `food`, `barfood`, and `foobar`. Below these fields are four buttons: IGNORECASE, MULTILINE (which is highlighted in yellow), DOTALL, and VERBOSE. Further down, there are two more sections: "Match result:" showing the original text with "foo" highlighted in green, and "Match captures:" which displays the message "No groups." twice.

6. Notice that "foo" in "food" and in "foobar" are highlighted, meaning that those are matches. But "foo" in "barfood" is not highlighted. It's not a match because "barfood" doesn't *start with* "foo".
7. Now change the regular expression to `foo$` and the string to search to

```
curfoo
food
```

8. Notice that "foo" in "curfoo" is highlighted, meaning that it is a match. But "foo" in "food" is not highlighted. It's not a match because "food" doesn't *end with* "foo".

Try the Regular Expressions

As we go through the syntax of regular expressions, try out some patterns in pythex.

Number of Occurrences (? + * { })

The following symbols affect the number of occurrences of the preceding character: ?, +, *, and { }.

A question mark (?) indicates that the preceding character should appear zero or one times in the pattern.

- The pattern `foo?` can be found in "food" and "fod", but not "faod".

A plus sign (+) indicates that the preceding character should appear one or more times in the pattern.

- The pattern `fo+d` can be found in "fod", "food" and "foood", but not "fd".

An asterisk (*) indicates that the preceding character should appear zero or more times in the pattern.

- The pattern `fo*d` can be found in "fd", "fod" and "food".

Curly brackets with one parameter ({ n }) indicate that the preceding character should appear exactly n times in the pattern.

- The pattern `fo{3}d` can be found in "foood", but not "food" or "fooood".

Curly brackets with two parameters ({ n1 , n2 }) indicate that the preceding character should appear between n1 and n2 times in the pattern.

- The pattern `fo{2,4}d` can be found in "food", "foood" and "fooood", but not "fod" or "fooood".

Curly brackets with one parameter and an empty second parameter ({ n , }) indicate that the preceding character should appear at least n times in the pattern.

- The pattern `fo{2,}d` can be found in "food" and "foooood", but not "fod".

Common Characters (. \d \D \w \W \s \S)

A period (.) represents any character except a newline.

- The pattern fo. d can be found in "food", "foad", "fo9d", and "fo*d".

Backslash-d (\d) represents any digit. It is the equivalent of [0-9] (to be discussed soon).

- The pattern fo\d d can be found in "fold", "fo4d" and "fo0d", but not in "food" or "fodd".

Backslash-D (\D) represents any character except a digit. It is the equivalent of [^0-9] (to be discussed soon).

- The pattern fo\D d can be found in "food" and "foad", but not in "fo4d".

Backslash-w (\w) represents any word character (letters, digits, and the underscore (_)).

- The pattern fo\wd can be found in "food", "fo_d" and "fo4d", but not in "fo*d".

Backslash-W (\W) represents any character except a word character.

- The pattern fo\W d can be found in "fo*d", "fo@d" and "fo.d", but not in "food".

Backslash-s (\s) represents any whitespace character (e.g., space, tab, newline, etc.).

- The pattern fo\s d can be found in "fo d", but not in "food".

Backslash-S (\S) represents any character except a whitespace character.

- The pattern fo\S d can be found in "fo*d", "food" and "fo4d", but not in "fo d".

Character Classes ([])

Square brackets ([]) are used to create a character class (or character set), which specifies a set of characters to match

- The pattern f [aeiou] d can be found in "fad" and "fed", but not in "food", "faed" or "fd".
 - [aeiou] matches an "a", an "e", an "i", an "o", or a "u"
- The pattern f [aeiou] {2} d can be found in "faed" and "feod", but not in "fod", "fed" or "fd".
- The pattern [A-Za-z] + can be found in "Webucator, Inc.", but not in "13066".
 - [A-Za-z] matches any letter, regardless if it is lower case or upper case.

- `[A-Z]` matches any upper case letter.
- `[a-z]` matches any lower case letter.
- The pattern `[1-9] +` can be found twice in "13066", but not in "Webucator, Inc."

Negation (`^`)

When used as the first character within a character class, the caret (`^`) is used for negation.

- The pattern `f[^aeiou]d` can be found in "fqd" and "f4d", but not in "fad" or "fed".

Groups (`()`)

Parentheses (`()`) are used to capture subpatterns and store them as groups, which can be retrieved later.

- The pattern `f(oo)?d` can be found in "food" and "fd", but not in "fod".

Alternatives (`|`)

The pipe (`|`) is used to create optional patterns.

- The pattern `foo$|^bar` can be found in "foo" and "bar" and "barfoo", but not "foobar".

Escape Character (`\`)

The backslash (`\`) is used to escape special characters.

- The pattern `fo\..d` can be found in "fo.d", but not in "food" or "fo4d".

Backreferences

Backreferences are special wildcards that refer back to a group within a pattern. They can be used to make sure that two subpatterns match. The first group in pattern is referenced as `\1`, the second group is referenced as `\2`, and so on.

For example, the pattern `([bmpw])\1` matches "bob", "mom", "pop", and "wow", but not "bop" or "pow".

A more practical example has to do with matching the delimiter in social security numbers. Examine the following regular expression:

Regular Expressions

```
^\d{3}([-\ ]?)\d{2}([-\ ]?)\d{4}$
```

Within the caret (^) and dollar sign (\$), which are used to specify the beginning and end of the pattern, there are three sequences of digits, optionally separated by a hyphen or a space. This pattern will be matched in all of the following strings (and more):

- 123-45-6789
- 123 45 6789
- 123456789
- 123-45 6789
- 123 45-6789
- 123-456789

The last three strings are not ideal, but they do match the pattern. Backreferences can be used to make sure that the second delimiter matches the first delimiter. The regular expression would look like this.

```
^\d{3}([-\ ]?)\d{2}\1\d{4}$
```

The \1 refers back to the first subpattern. Only the first three strings listed above match this regular expression.

8.2 Python's Handling of Regular Expressions

In Python, you use the `re` module to access the regular expression engine. Here is a very simple illustration. Imagine you're looking for the pattern "r[aeiou]se" in the string "A rose is a rose is a rose."

1. Import the `re` module:

```
import re
```

2. Compile the pattern:

```
p = re.compile('r[aeiou]se')
```

3. Search the string for a match:

```
result = p.search('A rose is a rose is a rose.')
```

4. Print the result:

```
print(result)
```

This will print the following, showing that the result is a `match` object and that it found the match "rose" starting at index 2 and ending at index 6:

```
<_sre.SRE_Match object; span=(2, 6), match='rose'>
```

Compiling a regular expression pattern into an object is a good idea if you're going to reuse the expression throughout the program, but if you're just using it once or twice, you can use the module-level `search()` method, like this:

```
result = re.search('r[aeiou]se', 'A rose is a rose is a rose.')
```

Raw String Notation

Python uses the backslash character (\) to escape special characters. For example \n is a newline character. A call to `print('a\nb\nc')` will print the letters a, b, and c each on its own line. If you actually want to print a backslash followed by an "n", you need to escape the backslash with another backslash, like this: `print('a\\nb\\nc')`. That will print the literal string "a\nb\nc".

Python provides another way of doing this. Instead of escaping all the backslashes, you can use rawstring notation by placing the letter "r" before the beginning of the string, like this: `print(r'a\nb\nc')`.

While this may not come in very handy in most areas of programming, it is very helpful when writing regular expression patterns. That is because the regular expression syntax also uses the backslash for special characters. If you don't use raw string notation, you may find your patterns filled with backslashe

The takeaway: Always use raw string notation for your patterns.

Regular Expression Object Methods

Method	Description	Returns
p. <code>search(string)</code> *	Finds first substring that matches pattern	Match object or None
p. <code>match(string)</code> *	Like <code>search()</code> , but the match must be found at the beginning of the string.	Match object or None
p. <code>fullmatch(string)</code> *	Like <code>search()</code> , but the whole string must match. New in Python 3.4.	Match object or None
p. <code>.findall(string)</code> *	Finds all non-overlapping matches.	List of strings
p. <code>finditer(string)</code> *	Finds all non-overlapping matches.	Iterator of match objects
p. <code>split(string, maxsplit=0)</code>	Splits the string on pattern matches. If <code>maxsplit</code> is nonzero, limits splits to <code>maxsplit</code> .	List of strings
p. <code>sub(repl, string, count=0)</code>	Replaces all non-overlapping matches in <code>string</code> with <code>repl</code> . If <code>count</code> is nonzero, limits replacements to <code>count</code> . More details on <code>sub()</code> under Using <code>sub()</code> with a Function (see page 210).	String

*Method includes `start` and `end` arguments that indicate what positions in the string to start and end the search.

All of these methods have equivalent or near-equivalent module-level methods.

Flags

The `compile()` method takes an optional second argument: `flags`. The flag are constants that can be used individually or combined with a pipe (|).

Syntax

```
re.compile(pattern, re.FLAG1|re.FLAG2)
```

The table below describes the flags

Regular Expression Flags

Long Flag	Short Flag	Description
<code>re.IGNORECASE</code>	<code>re.I</code>	Makes the pattern case insensitive.
<code>re.MULTILINE</code>	<code>re.M</code>	Makes <code>^</code> and <code>\$</code> consider each line as an independent string.
<code>re.DOTALL</code>	<code>re.S</code>	Makes <code>.</code> match any character including the newline character .
<code>re.ASCII</code>	<code>re.A</code>	Makes the following special characters match only ASCII characters: <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\d</code> , <code>\D</code> , <code>\s</code> , <code>\S</code>
<code>re.VERBOSE</code>	<code>re.X</code>	Causes whitespace in the pattern to be ignored and allows for single-line comments using <code>#</code> .
<code>re.DEBUG</code>	N/A	Displays debug information about a compiled pattern.
<code>re.LOCALE</code>	<code>re.L</code>	Avoid this one. Makes the following special characters locale dependent: <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\s</code> , <code>\S</code>

Groups

As discussed earlier, parentheses in regular expression patterns are used to capture groups. You can access these groups individually using a match object's `group()` method or all at once using its `groups()` method.

The `group()` method takes an integer²⁶ as an argument:

- `match.group(0)` returns the whole match.
- `match.group(1)` returns the first group found
- `match.group(2)` returns the second group found.
- And so on...

You can also get multiple groups at the same time returned as a tuple of strings by passing in more than one argument (e.g., `match.groups(1, 2)`).

When nested parentheses are used in the pattern, the outer group is returned before the inner group. Here is an example that illustrates that:

26. Groups can also be named through a Python extension to regular expressions. For more information, see <https://docs.python.org/3/howto/regex.html#non-capturing-and-named-groups>.

Regular Expressions

```
p = re.compile(r'(\w+)@(\w+\.( \w+))')
match = p.match('andre@example.com')
email = match.group(0)
handle = match.group(1)
domain = match.group(2)
domain_type = match.group(3)
print(email, handle, domain, domain_type, sep='\n')
```

The output:

```
andre@example.com
andre
example.com
com
```

Notice that "example.com" is group 2 and "com", which is nested within "example.com" is group 3.

And you can use the groups method to get them all at once:

```
print(match.groups())
```

The output:

```
('andre', 'example.com', 'com')
```

Match Object Methods

Method	Description	Returns
match.group([group1, ...])	Explained above.	String or a tuple of strings
match.groups()	Explained above.	A tuple of strings
match.start(group=0)	Gets the index of the start of the substring matched.	Integer
match.end(group=0)	Gets the index of the end of the substring matched.	Integer
match.span(group=0)	Returns a tuple containing match.start(group) and match.end(group) .	Tuple

Using sub() with a Function

The `sub()` method can either replace each match with a string or with the return value of a specified function. The function receives the match as an argument and must return a string that will replace the matched pattern. Here is an example:

```
import random
def clean_cuss(match):
    cuss = match.group(0)
    l = len(cuss)
    s = '!@#$%^&*'
    while l > len(s):
        s += s
    return ''.join(random.sample(s,l))

p = re.compile(r'\b[a-z]*(stupid|stinky|darn|shucks|crud|slob) [a-
z]*\b', re.IGNORECASE|re.MULTILINE)
s = '''Shucks! What a cruddy day I've had. I spent the whole darn
day \
with my sloppiest friend darning his stinky socks.'''
result = p.sub(clean_cuss,s)
```

And here is a possible result:

```
@%#!*! What a @^#!*& day I've had. I spent the whole ^!&% day with
my *^&!%$*# friend @!#$&*% his ^#&*!$ socks.
```

Want to learn more about regular expressions?

To learn more about regular expressions, see [Python's Regular Expression HOWTO²⁷](https://docs.python.org/3/howto/regex.html).

²⁷. See <https://docs.python.org/3/howto/regex.html>.

9. Unicode and Encoding

In this lesson, you will learn...

1. About binary and hexadecimal numbers.
2. How to convert numbers between number systems.

9.1 Bits and Bytes

What is a bit? A bit is **binary digit**. It has only two possible values: 0 and 1. Compare that to a decimal digit, which has ten possible values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Computers store data as binary numbers. All the meaningful stuff you see (e.g., letters and integers) is ultimately stored as bits. For example, the capital letter "A" is stored as 1000001. It takes 7 bits²⁸ to store the letter "A".

How would you write the binary number 1000001 as a decimal? First, it helps to understand how decimal numbers work. Decimal numbers have places. For example, take the decimal number 127:

1. 127. The 7 is in the ones place. There are 7 ones. $7 \times 1 = 7$.
2. 127. The 2 is in the tens place. There are 2 tens. $2 \times 10 = 20$.
3. 127. The 1 is in the hundreds place. There is 1 hundred. $1 \times 100 = 100$.

Add the three results up and we get $100 + 20 + 7 = 127$.

Notice as we move from right to left through the number the places go up by a factor of 10, which is the base: the ones place, the tens place, the hundreds place, the thousands place, etc.

Binary numbers work the same way. The places go up by a factor of the base, 2: the ones place, the twos place, the fours place, the eights place, the sixteens place, etc.

Looking again at the binary number 1000001:

1. 1000001. The right-most 1 is in the one's place. There is 1 one. $1 \times 1 = 1$.
2. 1000001. The 0 is in the twos place. There are 0 twos. $0 \times 2 = 0$.
3. 1000001. The 0 is in the fours place. There are 0 fours. $0 \times 4 = 0$.
4. 1000001. The 0 is in the eights place. There are 0 eights. $0 \times 8 = 0$.
5. 1000001. The 0 is in the sixteens place. There are 0 sixteens. $0 \times 16 = 0$.

28. The most basic common encoding system is ASCII and it uses 8 bits (1 byte) to store characters, but it could have gotten away with just 7 bits.

6. 1000001. The 0 is in the thirty-twos place. There are 0 thirty-twos. $0 \times 32 = 0$.
7. 1000001. The left-most 1 is in the sixty-fours place. There is 1 sixty-four. $1 \times 64 = 64$.

Add the seven results up and we get $64 + 0 + 0 + 0 + 0 + 0 + 1 = 65$. So, the decimal number 65 is equal to the binary number 1000001.

You can see why humans prefer decimal numbers.

9.2 Hexadecimal Numbers

What is a hexadecimal number? A binary number is **base 2**. A decimal number is **base 10**. And a hexadecimal number is **base 16**. In the numbering system we use, there are only characters representing digits (0 - 9). Hexadecimal numbers require characters for 16 digits. The additional ones are represented with a - f. So, the hexadecimal digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. The letters a through f represent 10 through 15.

So, how do we represent the decimal number 65 as a hexadecimal number? We have to consider the number places again. Just as with binary and decimal numbers, we start with the ones place and then multiply by the base, 16, with each move to the left: the ones place, the sixteens place, the two-hundred-fifty-sixes place, the four-thousand-ninety-sixes place, etc.

So, to find out the hexadecimal number for 65, we can look at the places and ask how many of each number at that place do we need. To do this, it is easier to go from left to right, starting with the first place that is smaller than our number. For the decimal number 65, that's the sixteens place. It turns out we need 4 sixteens ($4 \times 16 = 64$) and 1 one ($1 \times 1 = 1$). Add those two values together ($64 + 1$) and we get 65. So, the decimal number 65 is equal to the hexadecimal number 41.

Exercise 24 Converting Numbers between Number Systems

10 to 15 minutes

In this exercise, you will use pencil and paper to try to convert numbers between different number systems.

1. Convert the decimal number 100 to a hexadecimal number.
2. Convert the decimal number 10 to a binary number.
3. Convert the binary number 101 to a decimal number.

***Challenge**

1. Convert the decimal number 1000 to a hexadecimal number.
2. Convert the binary number 10010 to a hexadecimal number.

Exercise Solution

1. **Convert the decimal number 100 to a hexadecimal number.** To make 100, we need
 - A. 6 sixteens. $6 \times 16 = 96$.
 - B. 4 ones. $4 \times 1 = 4$.So, the hexadecimal number is **64**.
2. **Convert the decimal number 10 to a binary number.** To make 10, we need:
 - A. 1 eight. $1 \times 8 = 8$.
 - B. 0 fours. $0 \times 4 = 0$.
 - C. 1 two. $1 \times 2 = 2$.
 - D. 0 ones. $0 \times 1 = 0$.So, the binary number is **1010**.
3. **Convert the binary number 101 to a decimal number.**
 - A. **101**. There is 1 one. $1 \times 1 = 1$.
 - B. **101**. There are 0 twos. $0 \times 2 = 0$.
 - C. **101**. There is 1 four. $1 \times 4 = 4$.So, the decimal number is $1 + 0 + 4 = 5$.

Challenge Solution

1. **Convert the decimal number 1000 to a hexadecimal number.** To make 1000, we need
 - A. 3 two-hundred-fifty-si es. $3 \times 256 = 768$.
 - B. 14 sixteens. $14 \times 16 = 224$. Remember that 14 is represented by the character `e`.
 - C. 8 ones. $8 \times 1 = 8$.So, the hexadecimal number is **3e8**.
2. **Convert the binary number 10010 to a hexadecimal number.** The easiest way to do this is to first co vert it to a decimal and then convert that decimal to a hexadecimal:
 - A. Convert the binary number 10010 to a decimal number.
 - i. **10010**. There are 0 ones. $0 \times 1 = 0$.
 - ii. **10010**. There is 1 two. $1 \times 2 = 2$.
 - iii. **10010**. There are 0 fours. $0 \times 4 = 0$.
 - iv. **10010**. There are 0 eights. $0 \times 8 = 0$.
 - v. **10010**. There is 1 sixteen. $1 \times 16 = 16$.So, the decimal number is $2 + 16 = 18$.
 - B. Convert the decimal number 18 to a hexadecimal number. To make 18, we need:
 - i. 1 sixteen. $1 \times 16 = 16$.
 - ii. 2 ones. $2 \times 1 = 2$.So, the hexadecimal number is **12**.

Exercise Solution

unicode-encoding/Solutions/answers.py

```
1. # Convert decimal 100 to hex:  
2. print(hex(100))  
3. # Convert the decimal number 10 to a binary number:  
4. print(bin(10))  
5. #Convert the binary number 101 to a decimal number:  
6. print(int('101', 2))  
7. # Convert the decimal number 1000 to a hexadecimal number:  
8. print(hex(1000))  
9. # Convert the binary number 10010 to a hexadecimal number:  
10. print(hex( int('10010', 2) ))
```

hex(), bin(), ord(), chr(), and int()

Python provides functions for converting between characters and numbers of different bases:

- `hex(int)` - Returns the hexadecimal string representation of `int`.
- `bin(int)` - Returns the binary string representation of `int`.
- `ord(char)` - Returns the Unicode code point (a decimal number) mapped to `char`.
- `chr(int)` - Returns the character mapped to the Unicode code point at `int`.
- `int(strnum, base)` - Returns `strnum` of base `base` as a decimal number.

Let's see these functions in action:

1. Run [unicode-encoding/Demos/conversions.py](#) to see how `hex()`, `ord()`, `chr()`, and `int()` work.
2. Note that string representations of hexadecimal numbers are prefixed with "0x" and string representations of binary numbers are prefixed with "0b".

9.3 Encoding

Encoding issues can be incredibly difficult to solve, especially if you're not sure how encoding actually works. Here's what you need to know to help you figure out how to prevent and resolve encoding problems.

1. Unicode is the prevailing standard for mapping code points (integers) to characters.
2. Characters are representations of Unicode code points.
3. Unicode code points are integers ranging from 0 to 1,114,111 (but only about 10% of those are used).
 - A. If you know the code point, you can find the character. For example, code point 65 maps to the Latin capital letter A. You can count on this no matter what modern operating system or software you are using.
4. Unicode code points are not stored in memory. Instead, encoded versions of those code points are stored as bytes or sequences of bytes.
 - A. Here lies the difficulty. Knowing the bytes is not enough to determine the characters. You must also know how to decode them. In other words, you must know how the text you are working with was encoded.

Encoding Text

1. First came ASCII with 95 mappings to characters. Each mapping used a single byte (8 bits).

Unicode and Encoding

2. Then came ISO-8859-1 with 191 mappings to characters, consisting of the 95 one-byte ASCII mappings and 96 new one-byte mappings.
 - A. Windows-1252 (or cp1252) is a superset of ISO-8859-1.
3. Later came UTF-8, consisting of the 95 one-byte ASCII mappings and mappings to all Unicode code points.
4. UTF-8 and ISO-8859-1 do NOT use the same mappings. UTF-8 uses two-byte mappings for all the ISO-8859-1 mapped characters. For example, for the letter é UTF-8 uses c3 a9 and ISO-8859-1 uses e9.
5. UTF-8 uses between one and four bytes to encode each Unicode code point. See the table below.

UTF-8 Bytes per Mapping

Bits of code point	First code point		Last code point		Bytes in sequence
	Hex	Dec	Hex	Dec	
7	\u0000	0	\u007F	127	1
11	\u0080	128	\u07FF	2047	2
16	\u0800	2048	\uFFFF	65535	3
21	\U00010000	65536	\U0010FFFF	1114111	4

Encoding and Decoding Files in Python

Unfortunately, there is no sure way of determining how a document is encoded just by looking at the bytes. You can write code to look for clues, but doing so is quite complex and beyond the scope of this course.²⁹

However, Python does make it easy for you to see the preferred encoding of your operating system by running this code:

```
Get the Preferred Encoding on Your OS  
import locale  
locale.getpreferredencoding(False)
```

On Windows machines, that's likely to output cp1252. On other machines, it will most likely output utf-8.

Python's built-in `open()` function includes two parameters related to encoding:

29. To learn more, see [How to guess the encoding of a document? \(http://unicodebook.readthedocs.org/en/latest/guess_encoding.html\)](http://unicodebook.readthedocs.org/en/latest/guess_encoding.html).

1. `encoding` - the name of the encoding used by the file. This defaults to the encoding returned by `locale.getpreferredencoding(False)`.
2. `errors` - string specifying how to handle encoding and decoding errors. Options are:
 - A. 'strict' - Raise a `ValueError`. This is the default.
 - B. 'ignore' - Ignore errors. Bytes that cannot be encoded or decoded will be omitted.
 - C. 'replace' - Replace bytes that cannot be decoded with a replacement marker (such as '?').
 - D. 'surrogateescape' - Replace undecodable bytes with code points in the Unicode Private Use Area so that the same bytes can be restored when writing the data after processing. Useful when you're not sure how file are encoded.
 - E. 'xmlcharrefreplace' - Replace unencodable characters with an XML character reference. Only used when writing to a file
 - F. 'backslashreplace' - Replace unencodable characters with Python's backslashed escape sequences. Only used when writing to a file

The script below shows you how different encodings encode the pseudo-word È Ñ Ç ð Ð ¡ Ñ G:

Code Sample

unicode-encoding/Demos/encoding.py

```

1. word = 'Ã Ä Å Æ Ð Æ ¡ Ä G'
2. encodings = ['utf-8', 'cp1252', 'iso-8859-1', 'ascii']
3. for encoding in encodings:
4.     print(word.encode(encoding, errors='replace'))
5. print(word.encode('ascii', errors='xmlcharrefreplace'))

```

Code Explanation

The script will output the following:

```

b'\xc3\x8a \xc3\x91 \xc3\x87 \xc3\xb0 \xc3\x90 \xc2\xa1 \xc3\x91
G'
b'\xca \xd1 \xc7 \xf0 \xd0 \xa1 \xd1 G'
b'\xca \xd1 \xc7 \xf0 \xd0 \xa1 \xd1 G'
b'? ? ? ? ? ? G'
b'         G'

```

And here are the same results shown in a table. Note that Python outputs the actual characters for characters encoded with one byte. In the table below, we have replaced the G with the actual encoding, 47, and we have removed the \x to just show the hexadecimal byte codes.

Different Encodings

Encoding	Ê	Ñ	Ç	ð	Đ	í	ñ	G
UTF-8	c3 8a	c3 91	c3 87	c3 b0	c3 90	c2 a1	c3 91	47
cp1252	ca	d1	c7	f0	d0	a1	d1	47
ISO-8859-1	ca	d1	c7	f0	d0	a1	d1	47
ASCII	?	?	?	?	?	?	?	47
ASCII (XML)	Ê	Ñ	Ç	ð	Ð	¡	Ñ	47

Things to notice:

1. UTF-8 uses two bytes for all but the Latin G.
2. cp1252 and ISO-8859-1 use the same encoding for these characters.
3. ASCII can only encode the Latin G.
4. The last row shows how `xmlcharrefreplace` works when a character cannot be encoded using ASCII.

Converting a File from cp1252 to UTF-8

1. Open the `characters_dos.txt` in `unicode-encoding/Demos` in your editor. The file just contains the pseudo-ord `Ê Ñ Ç ð Đ ï ñ G`. The file as encoded as cp1252.
2. The following scripts can be found in `unicode-encoding/Demos`.
3. Run `read_cp1252_byte_mode`. This opens the `characters_dos.txt` in **byte mode**, prints the bytes and then prints the decoded bytes using the `decode()` method of a bytes object.
 - A. Notice that all the characters were encoded as single bytes. Given that these characters are not part of the the ASCII character set, this demonstrates that the file is not encoded as UTF-8
4. Run `read_cp1252_text_mode`. This opens the same file in **text mode**, prints the encoded characters using the `encode()` method of a string, and then prints the plain characters.
5. Run `convert_to_utf8_fil`. This opens the same file in text mode using cp1252 encoding and then writes it to a new file using UTF-8 encoding
6. Run `read_utf8_fil`. This attempts to open the UTF-8-encoded files using cp1252 encoding. The result is a `UnicodeDecodeError` error.
7. Run `read_utf8_file_with_errors_paramete`. This again attempts to open the UTF-8-encoded files using cp1252 encoding, but it uses `errors='replace'` to replace characters it cannot decode. It doesn't error, but it returns garbage because it's decoding a UTF-8 file using cp1252

8. Run `read_utf8_file_encodin` to correctly open the file using UTF-8 encoding.
9. Run `read_utf8_file_byte_mod` to open the file in byte mode and then decode it with UTF-8.

Exercise 25 Finding Confusables

15 to 25 minutes

In this exercise, you will search through a list of URLs for fake URLs masked with confusables (characters that look like Latin characters, but are actually characters from a different language). The URLs are shown below:

```
1 https://www.google.com
2 https://www.webucator.com
3 https://twitter.com
4 https://www.facebook.com
5 https://www.microsoft.com
6 https://www.youtube.com
7 https://www.python.org
8 http://www.amazon.com
9 https://www.usa.gov
10 https://www.wikipedia.org
```

1. Open `unicode-encoding/Exercises/finding_confusables.py` in your editor.
2. The script begins by opening `URLs.txt` and creating a list of URLs from it.
3. Implement the `is_ascii()` and `contains_confusables()` functions as described in the functions' docstrings.
4. In the `main` function, write code to loop through the list of URLs. For each URL:
 - A. If the URL does not contain any confusables, print it as is.
 - B. If the URL does contain one or more confusables, print the original URL followed by the corrected URL on a new line (and tabbed in).

Your output should look like this:

```
https://www.google.com
*https://www.googl[b'\xd0\xb5'].com

https://www.webucator.com

https://twitter.com

https://www.facebook.com

https://www.microsoft.com
*https://www.micr[b'\xd0\xbe']s[b'\xd0\xbe']ft.com

https://www.youtube.com

https://www.python.org
*https://www.[b'\xd1\x80']ython.org

http://www.amazon.com

https://www.usa.gov
*https://www.[b'\xcfc\x85']sa.gov

https://www.wikipedia.org
```

Unicode and Encoding

Exercise Solution

unicode-encoding/Solutions/finding_confusables.p

```
1.     with open('URLs.txt', encoding='utf-8') as f:
2.         urls = f.read().splitlines()
3.     def is_ascii(c):
4.         """Returns boolean indicating if c is ascii
5.
6.         Keyword arguments:
7.             c (str) - one-character string to check
8.         """
9.         return ord(c) < 256
10.
11.    def contains_confusables(s):
12.        """Searches string for confusables.
13.        Returns 2-element tuple:
14.            t[0]: Boolean - indicates if s contains one or more confusables.
15.            t[1]: String - copy of original string, but with confusables utf-
>>> 8 encoded.
16.
17.        Keyword arguments:
18.            s (str) - string to check
19.        """
20.        u = []
21.        confusable = False
22.        for c in s:
23.            if is_ascii(c):
24.                u.append(c)
25.            else:
26.                u.append('[' + str(c.encode('utf-8')) + ']')
27.                confusable = True
28.        s2 = ''.join(u)
29.
30.        return (confusable, s2)
31.    def main():
32.        for url in urls:
33.            t = contains_confusables(url)
34.            if t[0]:
35.                print(url, '*' + t[1], sep='\n\t', end='\n\n')
36.            else:
37.                print(url, end='\n\n')
38.    main()
```

9.4 Conclusion

It's possible that you will never need the `hex()`, `bin()`, `ord()`, `chr()` functions or need to convert binary or hexadecimal numbers to decimal numbers. But at some point or other, you will run into encoding errors and when you do, it is very helpful to understand what's going on behind the scenes. As the joke goes, there are 10 types of developers, those who understand binary and those who don't.

10. File Processing

In this lesson, you will learn...

1. To read files on the operating system
2. To create and write to files on the operating system
3. To access and work with directories on the operating system.
4. To work with the `os` and `os.path` modules.

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

10.1 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. The most basic syntax for opening a file is shown below:

Syntax

```
open(path_to_file, file_mode)
```

`path_to_file` can either be a relative or an absolute path. File modes are described below.

File Modes

File Mode	Description
r	Open for reading (default). Returns <code>FileNotFoundException</code> if the file doesn't exist.
w	Open for writing. If the file exists, existing content is erased. If the file doesn't exist, a new file is created
x	Create and open for writing. Returns <code>FileExistsError</code> if the file already exists.
a	Open for appending to end of content. If the file doesn't exist, a new file is created
r+	Open for reading and writing.
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
b	Opens in binary mode.
t	Opens in text mode. This is the default.

Opening Files Examples

```
open('poem.txt') #Open for reading
open('poem.txt', 'w') #Open for writing
open('poem.txt', 'r+') #Open for reading and writing
open('logo.jpg', '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.
- `list(f)` - does the same thing as `f.readlines()`.

You can also loop through a file line by line

```
for line in f:  
    print(line)
```

Because each line ends with a newline character and the `print()` function appends a newline character by default, the code above will print two newline characters at the end of each line. The following example illustrates this:

Code Sample

file-processing/Demos/looplines1.py

```
1. f = open('my_files/zen_of_python.txt')
2. for line in f:
3.     print(line)
4. f.close()
```

Code Explanation

Output:

The Zen of Python
Tim Peters
<https://www.python.org/dev/peps/pep-0020/>

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Use the end parameter of print() to remove one of the newline characters:

File Processing

Code Sample

file-processing/Demos/looplines2.py

```
1. f = open('my_files/zen_of_python.txt')
2. for line in f:
3.     print(line, end='')
4. f.close()
```

Code Explanation

Output:

```
The Zen of Python
Tim Peters
https://www.python.org/dev/peps/pep-0020/

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
```

Another option, which would have the same result, is to use `rstrip()` to strip the whitespace at the end of each line:

Code Sample

file-processing/Demos/looplines.rstrip.py

```
1. with open('my_files/zen_of_python.txt') as f:
2.     for line in f:
3.         print( line.rstrip() )
```

Closing Files

You should always be sure to close files when you are done with them. The two examples above use the `close()` method to do this. 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.

A better practice is to use the `with` keyword when working with files, like this:

Syntax

with Keyword

```
with open(file) as f:  
    do_stuff()
```

Using this structure, we don't have to explicitly close the file. 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. Here is our previous example rewritten to use `with`:

Code Sample

file-processing/Demos/looplines3.py

```
1.     with open('my_files/zen_of_python.txt') as f:  
2.         for line in f:  
3.             print(line,end='')
```

Exercise 26 Finding Text in a File

15 to 25 minutes

In this exercise, you will create a tool for searching through a file

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 183).

*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.
>>>
```

File Processing

Exercise 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.     for result in results:
16.         print(word, ' appears on line ', result[0],
17.               ': ', result[1], sep='', end='')
18.
19. main()
```

Writing to Files

Files opened with a mode that permits writing can be written to using the `write()` method as shown below:

Code Sample

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 file-processing/Demos/simple-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 27 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/Ercises/simple_write.py` in your editor.
2. Before running this file, look in the `file-processing/Ercises/mv_file` folder. It should **not** contain a `yoda.txt` file. If it does, delete it.
3. Now run `file-processing/Ercises/simple_write.py`.
4. Now look again in the `file-processing/Ercises/mv_file` folder. It should now contain a `yoda.txt` file. Open it in a text editor to see its contents.
5. In `simple_write.py`, change the string passed to the `write()` method and run the file again. It should completely overwrite the file.
6. Now change the mode from '`w`' to '`a`' and run it again. This time, it should append to the file.
7. 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 230) to see if you can figure out why and 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 print the results of your `f.read()` call, you will likely get just a newline character. This is because 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.

File Processing

Exercise 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())
```

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())
```

Exercise 28 List Creator

20 to 30 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/Ercises/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\(\) \(iterables#splitlines\)](#) method of a string object useful.
 - C. `delete_list()`
 - D. `print_list()`

File Processing

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""")
```

```
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.         elif choice.lower() == 'p':
50.             print_list()
51.         elif choice.lower() == '--all':
52.             delete_list()
53.         elif len(choice) and choice[0] == '+':
54.             add_item(choice[1:])
55.         elif len(choice) and choice[0] == '-':
56.             remove_item(choice[1:])
57.         else:
58.             print("I didn't understand.")
59.             show_instructions()
60.
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.

File Processing

Exercise 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'):
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())
-----Lines 39 through 77 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-----
```

10.2 The os and os.path Modules

os

The `os` module is a built-in Python module for interacting with the operating system. The table below shows some of its most useful methods:

os Methods

Method	Description
<code>os.chdir(path)</code>	Changes the current directory to <code>path</code> .
<code>os.getcwd()</code>	Returns the current working directory as a string.
<code>os.listdir(path)</code>	Returns a list of files and subdirectories in <code>path</code> , which defaults to the current directory.
<code>os.mkdir(dirname)</code>	Creates a directory named <code>dirname</code> . Errors if directory already exists.
<code>os.makedirs(path)</code>	Like <code>mkdir()</code> , but creates all the necessary directories along the path.
<code>os.remove(path)</code>	Deletes file at <code>path</code> . Errors if <code>path</code> points to a directory or doesn't exist.
<code>os.unlink(path)</code>	Same as <code>os.remove()</code> .
<code>os.rmdir(path)</code>	Deletes directory at <code>path</code> . Errors if directory is not empty or doesn't exist.
<code>os.removedirs(path)</code>	Like <code>rmdir()</code> , but removes all empty directories in <code>path</code> , starting with the leaf.
<code>os.rename(src, dst)</code>	Renames file or director , changing name from <code>src</code> to <code>dst</code> .
<code>os.renames(oldpath, newpath)</code>	Like <code>rename()</code> , but starts with creating all necessary directories in <code>newpath</code> and ends with removing all empty directories in <code>oldpath</code> .
<code>os.walk(top)</code>	Walks the directory tree, yielding for each directory a three-element tuple containing <code>dirpath</code> , <code>dirnames</code> , <code>filenames</code> . More details below.

Here are some examples illustrating the methods above:

```
os.listdir()
```

Code Sample

file-processing/Demos/directory_list.py

```
1. import os
2. dir_contents = os.listdir()
3. for item in dir_contents:
4.     print(item)
```

Code Explanation

This just prints out a list of files and directories in the current directory.

```
os.renames()
```

Code Sample

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.renames(foo,bar)
8.     print('Renamed', foo, 'to', bar)
9.
10. def bar2foo():
11.     os.renames(bar,foo)
12.     print('Renamed', bar, 'to', foo)
13.
14. foo2bar()
```

Code Explanation

1. Before running this file, look in the file-processing/Demos/my_files folder. It should contain a file named foo.txt.
2. Now look in the file-processing/Demo folder. It should **not** contain a my_new_file folder. If it does, delete it.
3. Now run file-processing/Demos/renames.py.

4. Now look again in the `file-processing/Demo` folder. A `mv_new_file` folder should have appeared. Open it to find a `bar.txt` file, which is the `foo.txt` renamed and moved.
5. Now look again in the `file-processing/Demos/my_file` folder. The `foo.txt` file should be gone. The `mv_file` folder remains, because it has other things in it.
6. Now run `bar2foo()` at the Python prompt:
 - A. `file-processing/Demos/my_files/foo.t` returns.
 - B. `file-processing/Demos/mv_n_w_files/ba.txt` gets deleted.
 - C. `file-processing/Demos/mv_n_w_file` also gets deleted as the only file it contained was `bar.txt`.

`os.walk()`

Full `os.walk()` Signature

```
walk(top, topdown=True, onerror=None, followlinks=False)
```

The `os.walk()` method returns a generator by walking the directory tree and yielding for each directory a three-element tuple 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 though; you could end up with an infinite loop

Code Sample**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, run `dashes2spaces()` at the Python prompt to replace the dashes with spaces again and then `spaces2dashes()` to re-replace the spaces with dashes.

os.path

The `os.path` module is used for performing functions on path names. The table below shows some of its most useful methods:

File Processing

os Methods

Method	Description
<code>os.path.abspath(path)</code>	Returns the absolute path of path.
<code>os.path.basename(path)</code>	Returns the basename of path.
<code>os.path.dirname(path)</code>	Returns the path to the directory containing the leaf element of path.
<code>os.path.exists(path)</code>	Returns True if path exists.
<code>os.path.getatime(path)</code>	Returns the time path was last accessed in number of seconds from the epoch ³⁰ .
<code>os.path.getmtime(path)</code>	Returns the time path was last modified in number of seconds from the epoch.
<code>os.path.getctime(path)</code>	Returns the time path was created in number of seconds from the epoch.
<code>os.path.getsize(path)</code>	Returns the size of path.
<code>os.path.isfile(path)</code>	Returns True if path is a file
<code>os.path.isdir(path)</code>	Returns True if path is a directory.
<code>os.path.relpath(path, start)</code>	Returns a relative path to path from start, which defaults to the current directory.
<code>os.path.join(path, *paths)</code>	Returns a path by joining the passed-in paths intelligently.
<code>os.path.split(path)</code>	Returns a 2-element tuple containing <code>os.path.dirname(path)</code> and <code>os.path.basename(path)</code> .
<code>os.path.splitext(path)</code>	Returns a 2-element tuple containing <i>the path minus the extension</i> and <i>the extension</i> . For example, ('my_files/logo', '.jpg').

30. Midnight, January 1, 1970.

Code Sample**file-processing/Demos/os_path.py**

```
1. import os.path
2. print('1.', os.path.abspath('.'))
3. print('2.', os.path.basename('my_files/logo.jpg'))
4. print('3.', os.path.dirname('my_files/logo.jpg'))
5. print('4.', os.path.exists('my_files/logo.jpg'))
6. print('5.', os.path.getatime('my_files/logo.jpg'))
7. print('6.', os.path.getmtime('my_files/logo.jpg'))
8. print('7.', os.path.getctime('my_files/logo.jpg'))
9. print('8.', os.path.getsize('my_files/logo.jpg'))
10. print('9.', os.path.isabs('my_files/logo.jpg'))
11. print('10.', os.path.isfile('my_files/logo.jpg'))
12. print('11.', os.path.isdir('my_files/logo.jpg'))
13. print('12.', os.path.join('my_files', 'logo.jpg'))
14. print('13.', os.path.relpath('my_files/logo.jpg'))
15. print('14.', os.path.split(os.path.abspath('my_files/logo.jpg')))
16. print('15.', os.path.splitext('my_files/logo.jpg'))
```

File Processing

Code Explanation

Run this file and compare the code with the results (sh wn below).

```
>>>
1. D:\_Webucator\courseware\Python\ClassFiles\file-processing\Demos
2. logo.jpg
3. my_files
4. True
5. 1435002060.0521188
6. 1434391261.618303
7. 1435002060.0521188
8. 20360
9. False
10. True
11. False
12. my_files\logo.jpg
13. my_files\logo.jpg
14. ('D:\\_Webucator\\courseware\\Python\\ClassFiles\\file-processing\\
\\Demos\\my_files', 'logo.jpg')
15. ('my_files/logo', '.jpg')
>>>
```

10.3 Conclusion

In this lesson, you have learned to work with files and directories on the operating system.

11. Exception Handling

In this lesson, you will learn...

1. To handle exceptions in Python.

By this time, you've probably run into some exceptions (errors) in your Python scripts. For example, if you try to divide by zero, you will get a `ZeroDivisionError` exception:

```
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    1/0
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:

Code Sample

exception-handling/Demos/simple.py

```
1.  try:
2.      1/0
3.  except:
4.      print('You cannot divide by zero!')
```

You can have multiple `except` clauses with each clause specifying the type of exception to catch:

Code Sample

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

- 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.
- If the user enters 0 for the denominator, a `ZeroDivisionError` exception will be thrown.
- If the user quits the program by pressing **CTRL+C**, a `KeyboardInterrupt` exception will be thrown.
- The last `except` clause, which is optional, serves as a wildcard to catch any unspecified exception types.

11.1 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:

Code Sample**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.

11.2 Getting Information on Exceptions

Exceptions conveniently have a `__str__()` method. Python's built-in `str()` function can be used to call that method. To access the exception object, assign it to a variable using the `as` keyword, like this:

Code Sample**exception-handling/Demos/details.py**

```
1.     try:
2.         1/0
3.     except Exception as e:
4.         print(type(e)) #prints object type
5.         print(e) #prints value of str(e)
```

Code Explanation

Note that `e` is an arbitrary variable name. You can name the variable whatever you like.

Exercise 29 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.

Exception Handling

Exercise Solution

exception-handling/Solutions/exception_details.py

```
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.     #FileNotFoundException
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')
```

Exception Handling

Code Explanation

The above 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 'FileNotFoundException'>
[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'

>>> |
```

11.3 The else Clause

We want to limit our `try` clause to the code that might cause an exception. After the final `except` clause, we 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:

Code Sample

`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 `print()` function will not raise a `ValueError` or a `ZeroDivisionError`, it makes sense to include it within the `else` clause.

11.4 The finally Clause

The `finally` clause is for doing any necessary cleanup (e.g., closing connections, releasing resources, etc.). Here is a pseudo-example:

```
try:  
    resource = Resource() #some generic resource  
    resource.open()  
    resource.do_something()  
except Exception as e:  
    handle_exception(e) #runs only if exception is raised  
else:  
    do_something_more() #runs after try clause only if no exception  
    is raised  
finally:  
    resource.close() #always runs
```

11.5 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 `square_num()` function, which uses exception handling, with the `cube_num()` function, which uses an if condition to accomplish the same thing:

Code Sample

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.')
```

Exercise 30 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 `running_sum()` function prompts the user for a number, adds it to a total, and then recursively calls itself. If the user enters anything but an integer, the function prompts the user for integers only. Run the script to see how it works.
2. Now replace the if-else block in the `running_sum()` function with a try-except block. The script should continue to work in exactly the same way.

Exception Handling

Exercise Solution

exception-handling/Solutions/running_sum.py

```
1.      def running_sum(total=0):
2.          try:
3.              num = int(input('Enter a number: '))
4.          except ValueError:
5.              print('Integers only please. Try again.')
6.          else:
7.              total += num
8.              print('The current total is:', total)
9.
10.         running_sum(total)
11.
12.     def main():
13.         running_sum()
14.
15.     main()
```

11.6 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', 'chao': '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', 'chao': 'bye', 'hi': 'hola', 'bye': 'adios'}
```

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.

Open [exception-handling/Demos/bidict_with_exception.py](#) to try it out.

11.7 Exception Hierarchy

Below is a list of all the built-in exception types in Python 3.³¹

- `BaseException`
 - `SystemExit`
 - `KeyboardInterrupt`
 - `GeneratorExit`
 - `Exception`
 - `StopIteration`
 - `StopAsyncIteration` (New in 3.5)
 - `ArithmError`
 - `FloatingPointError`
 - `OverflowError`
 - `ZeroDivisionError`
 - `AssertionError`
 - `AttributeError`
 - `BufferError`
 - `EOFError`
 - `ImportError`
 - `ModuleNotFoundError`
 - `LookupError`
 - `IndexError`
 - `KeyError`
 - `MemoryError`
 - `NameError`
 - `UnboundLocalError`
 - `OSError`
 - `BlockingIOError`
 - `ChildProcessError`
 - `ConnectionError`
 - `BrokenPipeError`
 - `ConnectionAbortedError`
 - `ConnectionRefusedError`
 - `ConnectionResetError`
 - `FileExistsError`
 - `FileNotFoundException`
 - `InterruptedError`
 - `IsADirectoryError`
 - `NotADirectoryError`

31. The exception hierarchy for Python 2.7 is available at <https://docs.python.org/2.7/library/exceptions.html#exception-hierarchy>.

- `PermissionError`
- `ProcessLookupError`
- `TimeoutError`
- `ReferenceError`
- `RuntimeError`
 - `NotImplementedError`
 - `RecursionError`
- `SyntaxError`
 - `IndentationError`
 - `TabError`
- `SystemError`
- `TypeError`
- `ValueError`
 - `UnicodeError`
 - `UnicodeDecodeError`
 - `UnicodeEncodeError`
 - `UnicodeTranslateError`
- `Warning`
 - `DeprecationWarning`
 - `PendingDeprecationWarning`
 - `RuntimeWarning`
 - `SyntaxWarning`
 - `UserWarning`
 - `FutureWarning`
 - `ImportWarning`
 - `UnicodeWarning`
 - `BytesWarning`
 - `ResourceWarning`

11.8 Conclusion

In this lesson, you have learned to handle Python exceptions.

Exception Handling

12. Python Dates and Times

In this lesson, you will learn...

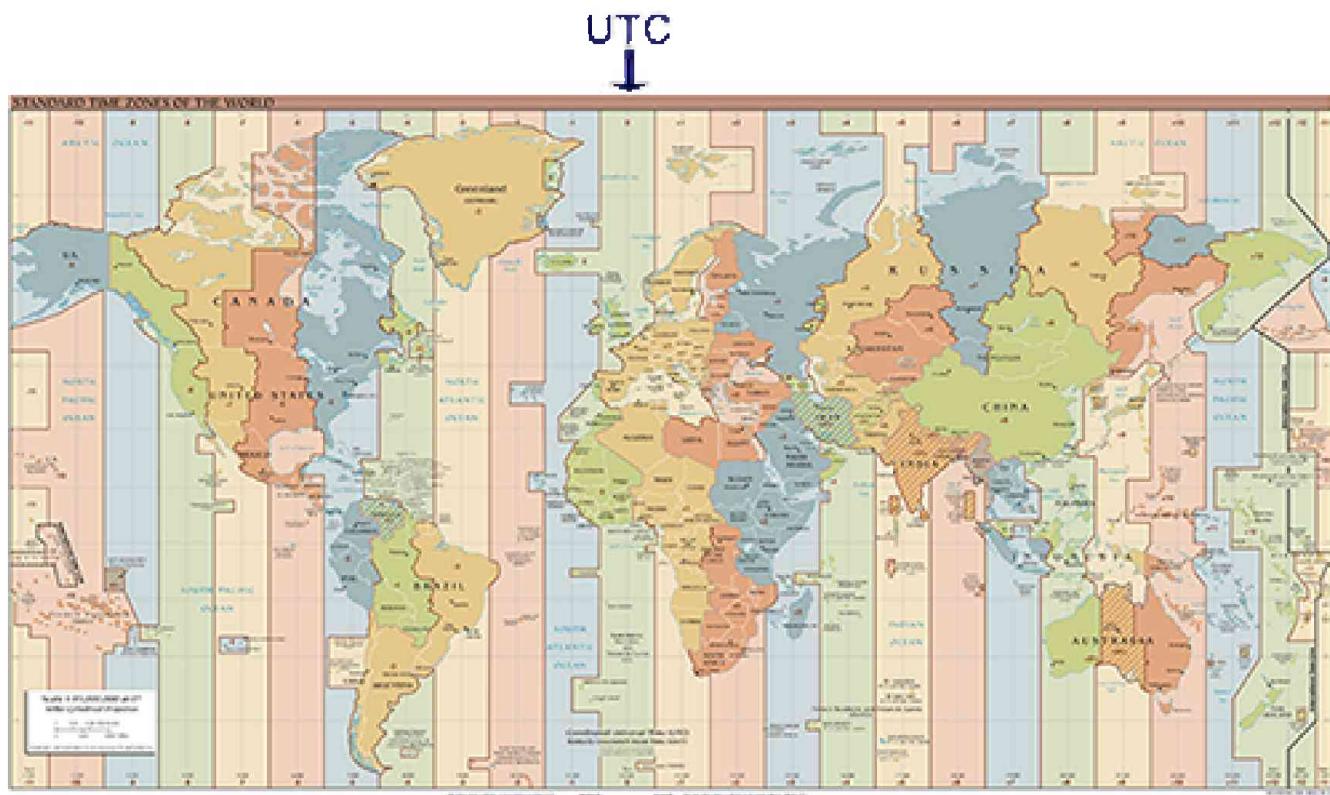
1. To work with the `time` module.
2. To work with the `datetime` module.

12.1 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. For example, in 2008 the United States [extended DST for a month as a means of saving energy](#).³² And Russia [changed its DST practices in different regions in 2011 and again in 2014](#).³³ 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

32. See https://en.wikipedia.org/wiki/Energy_Policy_Act_of_2005#Change_to_daylight_saving_time
33. See <http://www.timeanddate.com/news/time/russia-abandons-permanent-summer-time.html>.

time. The image below shows world time zones as offsets from UTC in June of 2015.



World Time Zones at 07:20, 7 June 2015.
(Source: https://commons.wikimedia.org/wiki/File:World_Time_Zones_Map.png)

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).³⁴ Times before the epoch are expressed internally as negative numbers.

34. 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 <http://grokbase.com/t/python/python-dev/086gxidb5a/epoch-and-platform>.

Python and Time

The most useful built-in modules for working with dates and times in Python are `time` and `datetime`.

12.1 The time Module³⁵

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.

Clocks³⁶

Python's `time` module includes five different types of clocks:

1. `time.clock()` - deprecated in 3.3
2. `time.monotonic()` - new in 3.3
3. `time.perf_counter()` - new in 3.3
4. `time.process_time()` - new in 3.3
5. `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.

35. Documentation on the `time` module: <https://docs.python.org/3/library/time.html>

36. For a deep, largely academic discussion on Python clocks, see: <https://www.webucator.com/blog/2015/08/python-clocks-explained/>

Code Sample

`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'))
```

Code Explanation

The above code will render the following:

```
>>>
s: 1,439,815,037.840448
m: 23,996,917.2973408
h: 399,948.62162234663
d: 16,664.52590093111
y: 45.62498535504753
>>> |
```

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 list comprehension. Here's the code:

Code Sample

date-time/Demos/compare_times.py

```
1. import time
2. import random
3.
4. num_nums = 100
5.
6. start_time = time.perf_counter()
7. numbers = str(random.randint(1,100))
8. for i in range(num_nums):
9.     num = random.randint(1,100)
10.    numbers += ' ' + str(num)
11. end_time = time.perf_counter()
12. td1 = end_time - start_time
13.
14. start_time = time.perf_counter()
15. numbers=[]
16. for i in range(num_nums):
17.     num = random.randint(1,100)
18.     numbers.append(str(num))
19. numbers = ', '.join(numbers)
20. end_time = time.perf_counter()
21. td2 = end_time - start_time
22.
23. start_time = time.perf_counter()
24. numbers = [str(random.randint(1,100)) for i in range(1,num_nums)]
25. numbers = ', '.join(numbers)
26. end_time = time.perf_counter()
27. td3 = end_time - start_time
28.
29. print('''Number of numbers: {},
30. Time Delta 1: {}
31. Time Delta 2: {}
32. Time Delta 3: {}'''.format(num_nums, td1, td2, td3))
```

Code Explanation

And here are the results using different values for num_nums:

```
>>>
Number of numbers: 100
Time Delta 1: 0.0009353939058037343
Time Delta 2: 0.0007763069622231492
Time Delta 3: 0.0007403841039952748
>>> =====
>>>
Number of numbers: 10,000
Time Delta 1: 0.05374899346017368
Time Delta 2: 0.03617758394985356
Time Delta 3: 0.03400308470050134
>>> =====
>>>
Number of numbers: 1,000,000
Time Delta 1: 4.6023194969680175
Time Delta 2: 3.7753650368885765
Time Delta 3: 3.5839815776386263
>>> =====
>>>
Number of numbers: 10,000,000
Time Delta 1: 95.63455209794158
Time Delta 2: 37.30114785195303
Time Delta 3: 34.297162280812245
>>>
```

Notice that the speed differential increases markedly when dealing with large amounts of data.

Python 2 Difference

Before Python 3.3, you would use `time.clock()` instead of `time.perf_counter()`.

The timeit Module

Python includes a [timeit](#)³⁷ module specifically for testing performance

Time Structures

Examples from this section are included in `times_as_structs.py` in the `date-time/Demos` folder.

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:

```
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)
```

The same time expressed as a string looks like this:

```
Thu Jan  1 00:00:00 1970
```

37. See <https://docs.python.org/3/library/timeit.html>.

Time Structure for the Epoch³⁸

Index	Attribute	Description	Epoch Values
0	tm_year	Year	1970
1	tm_mon	Month (1-12)	1 (January is 1)
2	tm_mday	Day of Month	1
3	tm_hour	Hour (0-23)	0
4	tm_min	Minute (0-59)	0
5	tm_sec	Second (0-59)	0
6	tm_wday	Day of Week	3 (Monday is 0)
7	tm_yday	Day of Year (1-366)	1
8	tm_isdst	Is Daylight Saving Time?	0 (0 is standard time, 1 is daylight time)

Methods that create `time.struct_time` objects include:

`time.gmtime([secs])`

Converts a time expressed in seconds since the epoch to a `struct_time` in 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.

```
epoch = time.gmtime(0)
now = time.gmtime()
yesterday = time.gmtime(time.time() - 60*60*24)
tomorrow = time.gmtime(time.time() + 60*60*24)
```

`time.localtime([secs])`

Converts a time expressed in seconds since the epoch to a `struct_time` in the local time. 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 DST applies.

38. As of Python 3.3, `time.struct_time` has two new attributes: `tm_gmtoff` and `tm_zone` available on some platforms.

```
epoch_offset = time.localtime(0)
now = time.localtime()
yesterday = time.localtime(time.time() - 60*60*24)
tomorrow = time.localtime(time.time() + 60*60*24)
```

The inverse of `time.localtime()` is `time.mktime()`, which takes a `struct_time` and returns the number of seconds since the epoch.

```
seconds_since_epoch = time.mktime(time.localtime())
```

Times as Strings

Examples from this section are included in `times_as_strings.py` in the `date-time/Demos` folder.

Methods that create times as strings include :

`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.

```
epoch = time.gmtime(0)
epoch_offset = time.localtime(0)

str_epoch = time.asctime(epoch)
str_epoch_offset = time.asctime(epoch_offset)
str_now = time.asctime()
```

`time.ctime([secs])`

Converts a time expressed in seconds since the epoch to a date in local time represented as a string. If the `secs` argument is omitted, it defaults to `time.time()`, which returns the current time in seconds since the epoch.

```
str_epoch_offset = time.ctime(0)
str_now = time.ctime()
```

Time and Formatted Strings

The `time.strftime(format [, t])` takes a `struct_time` object and returns a formatted string.

The `time.strptime(string [, format])` takes a string representing a time and returns a `struct_time` object.

The formatting directives are shown in the tables below:

String Time Formatting Directives³⁹

Directive	Meaning
%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale's equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	Time zone offset indicating a positive or negative time difference from UTC/GMT of the form +HHMM or -HHMM, where H represents decimal hour digits and M represents decimal minute digits [-23:59, +23:59].
%Z	Time zone name (no characters if no time zone exists).
%%	A literal % character.

The following demo provides some examples of `time.strftime()` and `time.strptime()`.

Code Sample**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')

```

time.sleep(secs)

The `time.sleep()` method suspends execution for a given number of seconds.
Run the demo below to see how it works.

Code Sample**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!')

```

12.2 The datetime Module⁴⁰

The `datetime` module has some overlap with the `time` module. It has some very useful methods for comparing dates and times. It includes the following types:⁴¹

1. `datetime.date` - a date with `year`, `month`, and `day` attributes.
2. `datetime.time` - a time with `hour`, `minute`, `second`, `microsecond`, and `tzinfo` attributes.

39. See <https://docs.python.org/3/library/time.html#time.strftime>.

40. Documentation on the `datetime` module: <https://docs.python.org/3/library/datetime.html>

41. The `datetime` module includes two additional types: `tzinfo` and `timezone`, which are not covered in this course.

Python Dates and Times

3. `datetime.datetime` - a combination of `datetime.date` and `datetime.time`.
4. `datetime.timedelta` - a duration expressing the difference between instances of two `datetime.date`, `datetime.time`, or `datetime.datetime` objects.

datetime.date objects

Examples from this section are included in `date_objects.py` in the `date-time/Demos` folder.

There are a number of `datetime` methods for creating `datetime.date` objects:

1. `datetime.date(year, month, day)`
2. `date.today()` - Returns the current local date.
3. `date.fromtimestamp(timestamp)` - Returns the local date corresponding to `timestamp`.

The tables below show `datetime.date` instance attributes and methods. The examples assume the following `datetime.date` object:

```
i_day = datetime.date(1776, 7, 4)
```

datetime.date Instance Attributes

Attribute	Description	Example
<code>date.year</code>	Year	<code>i_day.year</code> #1776
<code>date.month</code>	Month	<code>i_day.month</code> #7
<code>date.day</code>	Day	<code>i_day.day</code> #4

`datetime.date` Instance Methods

Attribute	Description	Example
<code>date.replace(year, month, day)</code>	Returns a new <code>datetime.date</code> instance based on <code>date</code> with the given replacements.	<code>i_day.replace(year=1826)</code> <code>#1826-07-04</code>
<code>date.timetuple()</code>	Returns a <code>struct_time</code> .	<code>i_day.timetuple()</code> <code>#struct_time obj</code>
<code>date.weekday()</code>	Returns an integer representing the day of the week	<code>i_day.weekday() #3</code> <code>(Thursday)</code>
<code>date.ctime()</code>	Returns a formatted date string. Similar to <code>time.ctime()</code> .	<code>i_day.ctime() #Thu Jul 4</code> <code>00:00:00 1776</code>
<code>date.strftime(format)</code>	Returns a formatted date string. Similar to <code>time.strftime(format [, t])</code> with the same formatting directives.	<code>i_day.strftime('%A, %B</code> <code>%d, %Y, %I:%M %p')</code> <code>#Thursday, July 04, 1776,</code> <code>12:00 AM</code>

`datetime.time` objects

Examples from this section are included in `time_objects.py` in the `date-time/Demos` folder.

`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`⁴² - defaults to None.

The tables below show `datetime.time` instance attributes and methods. The examples assume the following `datetime.time` object:

42. `tzinfo` is an abstract base class for time zone information objects. It is not covered in this course.

Python Dates and Times

```
t = datetime.time(hour=14,  
                  minute=13,  
                  second=12,  
                  microsecond=11)
```

`datetime.time` Instance Attributes

Attribute	Description	Example
<code>time.hour</code>	Hour	<code>t.hour</code> #14
<code>time.minute</code>	Minute	<code>t.minute</code> #13
<code>time.second</code>	Second	<code>t.second</code> #12
<code>time.microsecond</code>	Microsecond	<code>t.microsecond</code> #11

`datetime.time` Instance Methods

Attribute	Description	Example
<code>time.replace(hour, minute, second, microsecond)</code>	Returns a new <code>datetime.time</code> instance based on <code>time</code> with the given replacements.	<code>t.replace(hour=4) #04:13:12.000011</code>
<code>time.strftime(format)</code>	Returns a formatted date string. Similar to <code>time.strftime(format[, t])</code> with the same formatting directives.	<code>t.strftime('%I:%M%p') #02:13 PM</code>

`datetime.datetime` Objects

Examples from this section are included in `datetime_objects.py` in the `date-time/Demos` folder.

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)`
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)` - Like `time.strptime(string[, format])` with the same formatting directives.

The tables below show `datetime.datetime` instance attributes and methods. The examples assume the following `datetime.date` object:

```
mw = datetime.datetime(year=1969,
                      month=7,
                      day=21,
                      hour=2,
                      minute=56,
                      second=15)
```

`datetime.datetime` Instance Attributes

Attribute	Description	Example
<code>datetime.year</code>	Year	<code>mw.year</code> #1969
<code>datetime.month</code>	Month	<code>mw.month</code> #7
<code>datetime.day</code>	Day	<code>mw.day</code> #21
<code>datetime.hour</code>	Hour	<code>mw.hour</code> #2
<code>datetime.minute</code>	Minute	<code>mw.minute</code> #56
<code>datetime.second</code>	Second	<code>mw.second</code> #15
<code>datetime.microsecond</code>	Microsecond	<code>mw.microsecond</code> #0

**datetime* is replaced with *dt* (e.g., `dt.replace()` in place of `datetime.replace()`) in the table below for printing purposes.

`datetime.datetime` Instance Methods

Attribute	Description	Example
<code>dt.replace(year, month, day, hour, minute, second, microsecond)</code>	Returns a new <code>datetime.datetime</code> instance based on <code>datetime</code> with the given replacements.	<code>mw_50.replace(year=2015)</code> <code>#2019-07-21 02:56:15</code>
<code>dt.date()</code>	Returns a <code>datetime.date</code> object with same year, month, and day.	<code>mw.date()</code> #1969-07-21
<code>dt.time()</code>	Returns a <code>datetime.time</code> object with same hour, minute, second, and microsecond.	<code>mw.time()</code> #02:56:15
<code>dt.timetuple()</code>	Returns a <code>struct_time</code> representing the local time.	<code>mw.timetuple()</code> <code>#struct_time obj</code>
<code>dt.utctimetuple()</code>	Returns a <code>struct_time</code> representing the UTC time.	<code>mw.utctimetuple()</code> <code>#struct_time obj</code>
<code>dt.timestamp()</code>	Returns a timestamp corresponding to the <code>datetime</code> . New in Python 3.3.	<code>mw_50.timestamp()</code> <code>#1563692175.0</code>
<code>dt.weekday()</code>	Returns an integer representing the day of the week	<code>mw.weekday()</code> #0 (Monday)
<code>dt.ctime()</code>	Returns a formatted date string. Similar to <code>time.ctime()</code> .	<code>mw.ctime()</code> #Mon Jul 21 02:56:15 1969
<code>dt.strftime(format)</code>	Returns a formatted date string. Similar to <code>time.strftime(format [, t])</code> with the same formatting directives.	<code>mw.strftime('%A, %B %d, %Y, %I:%M %p')</code> #Monday, July 21, 1969, 02:56 AM

`datetime.timedelta` objects

Examples from this section are included in `time_deltas.py` in the `date-time/Demos` folder.

`datetime.timedelta` objects can be:

- `Added (t1 + t2)`. **Result:** a new `datetime.timedelta` object.

- Subtracted ($t_1 - t_2$). **Result:** a new `datetime.timedelta` object.
- Divided (t_1 / t_2). **Result:** a float
- Multiplied by an integer or float $t_1 * 2$). **Result:** a new `datetime.timedelta` object.
- Divided by an integer or float $t_1 / 2$). **Result:** a new `datetime.timedelta` object.

The tables below show `datetime.timedelta` instance attributes and methods. The examples assume the following `datetime.delta` object:

```
racetime = datetime.timedelta(days=8,
                             hours=23,
                             minutes=49,
                             seconds=9)
```

`datetime.timedelta` Instance Attributes

Attribute	Description	Example
<code>timedelta.days</code>	Days	<code>racetime.days</code> #8
<code>timedelta.seconds</code>	Seconds	<code>racetime.seconds</code> #85749 == $(23*60*60) + (49*60) + 9$
<code>timedelta.microseconds</code>	Microseconds	<code>racetime.microseconds</code> #0

`datetime.timedelta` Instance Methods

Attribute	Description	Example
<code>total_seconds()</code>	Returns the total number of seconds in the duration (with microsecond accuracy up to large numbers). New in Python 3.2.	<code>racetime.total_seconds()</code> #776949.0

Exercise 31 Report on Amtrak Departure Times

30 to 45 minutes

In this exercise, you will create a small report on Amtrak departure times from the month of July 2015. All the data is in a text file ([amtrak-data.txt](#)). The data is shown below:

Exercise Code

date-time/Exercises/amtrak-data.txt

```
1. *Sch DP Act DP
2. 07/01/2015 2:40 AM (We) 4:00AM
3. 07/01/2015 3:00 AM (We) 3:00AM
4. 07/01/2015 4:40 AM (We) 4:40AM
5. 07/01/2015 5:30 AM (We) 5:30AM
-----Lines 6 through 2478 Omitted-----
2479.07/31/2015 9:10 PM (Fr) 9:10PM
2480.07/31/2015 10:05 PM (Fr) 10:05PM
2481.07/31/2015 10:45 PM (Fr) 10:45PM
2482.07/31/2015 11:15 PM (Fr) 11:15PM
2483.*Data from http://juckins.net/amtrak\_status/archive/html/history.php
```

The file you will be working on has been started already:

Exercise Code**date-time/Exercises/amtrak.py**

```

1. import datetime
2.
3. def get_departures():
4.     departures = []
5.     with open('amtrak-data.txt') as f:
6.         for line in f.readlines():
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.datetime objects.'''
14.
15.     return (planned_departure, actual_departure)
16.
17. def main():
18.     departures = get_departures()
19.     ontime_departures = [d for d in departures if d[1] == d[0]]
20.     late_departures = [d for d in departures if d[1] > d[0]]
21.     next_day_departures = [d for d in departures if
22.                            d[0].day != d[1].day]
23.
24.     print('''Total Departures: {}'''.format(len(departures)),
25.           len(ontime_departures),
26.           len(late_departures),
27.           len(next_day_departures)))
28.
29.
30.
31.
32. main()

```

The `main()` and `get_departures()` functions are complete. Your task is to complete the `get_departure()` function.

1. Open `date-time/Exercises/amtrak.py` in your editor.

Python Dates and Times

2. Review the `main()` and `get_departures()` functions.
3. Now complete the `get_departure()` function:
 - A. Each valid passed-in line is formatted as follows:

```
07/01/2015 2:40 AM (We) \t4:00AM\n
```

- B. The first and last line in the file are not departure records. They begin with an asterisk (*). Return `None` for those lines.
- C. Each line ends with a newline character. Strip that and then split the line on the tab into a two-element list.
- D. There are a few lines that don't show an actual departure time. They are formatted like this:

```
07/03/2015 3:00 AM (Fr) \t\n
```

When you strip those lines, you will strip both the trailing tab and newline character. Splitting them on a non-existent tab will leave you with a one-element list. If the list has one or fewer elements, return `None`.

- E. If you haven't returned `None` yet, you should be left with a two-element list formatted like this:

```
['07/01/2015 2:40 AM (We)', '4:00AM']
```

- F. The first element is the scheduled departure time and the second element is the actual departure time. You need to convert these into `datetime.datetime` objects. This will be a bit tricky. A few hints:
 - i. Pay careful attention to the format of the strings.
 - ii. You will need to remove the last five characters of the scheduled departure time string before passing it to the proper `datetime` method.
 - iii. For now, just assume that the actual departure time is on the same date as the scheduled departure time.

*Challenge

Get the above working before you attempt the challenge.

After doing the above, your solution will show zero next day departures:

```
>>>  
Total Departures: 2467  
Ontime Departures: 1207  
Late Departures: 1258  
Next Day Departures: 0  
>>>
```

But there are a couple of records in which the actual departure took place the following day. You can identify these in the data because they have a scheduled PM departure time, but an actual AM departure time:

```
07/12/2015 11:05 PM (Su) 12:30AM
```

Even easier, you can identify these records in your code, because you used the same date for the actual and scheduled departures and just changed the times. If the actual departure time is earlier than the scheduled departure time, it means that you should have added a day.

1. Add the following code to your function:

```
if actual_departure < planned_departure:  
    actual_departure = add_day(actual_departure)
```

2. Then write an `add_day()` function that takes a `datetime.datetime` object and returns another `datetime.datetime` object that represents the following day.
3. Your new report should look like this:

```
>>>  
Total Departures: 2467  
Ontime Departures: 1207  
Late Departures: 1260  
Next Day Departures: 2  
>>>
```

Exercise Solution

date-time/Solutions/amtrak.py

```
-----Lines 1 through 11 Omitted-----
12. def get_departure(line):
13.     '''Return a tuple containing two datetime.datetime objects.'''
14.     if line[0] == '*':
15.         return None
16.
17.     departure = line.strip().split('\t')
18.
19.     if len(departure) <= 1:
20.         return None
21.
22.     planned_departure = datetime.datetime.strptime(departure[0][:-5],
23.                                                     '%m/%d/%Y %I:%M %p')
24.     actual_departure = datetime.datetime.strptime(departure[0][:10] +
25.                                                    departure[1],
26.                                                    '%m/%d/%Y%I:%M%p')
27.
28.     return (planned_departure, actual_departure)
>>>
-----Lines 29 through 45 Omitted-----
```

Challenge Solution

date-time/Solutions/amtrak_challenge.py

```
-----Lines 1 through 11 Omitted-----
12. def get_departure(line):
13.     '''Return a tuple containing two datetime.datetime objects.'''
14.     if line[0] == '*':
15.         return None
16.
17.     departure = line.strip().split('\t')
18.
19.     if len(departure) <= 1:
20.         return None
21.
22.     planned_departure = datetime.datetime.strptime(departure[0][:-5],
23.                                                     '%m/%d/%Y %I:%M %p')
24.     actual_departure = datetime.datetime.strptime(departure[0][:10] +
25.                                                    departure[1],
26.                                                    '%m/%d/%Y%I:%M%p')
27.
28.     if actual_departure < planned_departure:
29.         actual_departure = add_day(actual_departure)
30.
31.     return (planned_departure, actual_departure)
32.
33. def add_day(date):
34.     return date + datetime.timedelta(days=1)
>>>
-----Lines 35 through 51 Omitted-----
```

12.4 Conclusion

In this lesson, you have learned to work with the `time` and `datetime` modules.

13. Running Python Scripts from the Command Line

In this lesson, you will learn...

1. To create and run Python scripts meant to be called at the computer's command line.

13.1 sys.argv

You may find that you want to write a Python script that can be called at the computer's Terminal window or Command Prompt without opening the Python shell. You use the `argv` property of the `sys` module to create such a script.

`sys.argv` contains a list of the command line arguments, starting with the Python script itself. Here is a very simple example that prints out that list:

Code Sample

`sys-argv/Demos/simple_sys.py`

```
1. import sys
2.
3. def main(args):
4.     for arg in args:
5.         print(arg)
6.
7. if __name__ == '__main__':
8.     main(sys.argv)
```

To run this script:

1. Open a Terminal window or Command Prompt and navigate to the directory containing the script.
2. Run the following command: `python simple_sys.py foo bar`
3. It should output the following:

```
simple_sys.py
foo
bar
```

A More Useful Example

The following script compares two files containing lists (one word per line):

Code Sample

sys-argv/Demos/compare_lists.py

```
1. import sys
2.
3. def main(args):
4.     with open(args[1]) as f1:
5.         list_1 = f1.read().splitlines()
6.
7.     with open(args[2]) as f2:
8.         list_2 = f2.read().splitlines()
9.
10.    list_1_only = []
11.    list_2_only = []
12.
13.    for item in list_1:
14.        if item not in list_2:
15.            list_1_only.append(item)
16.
17.    for item in list_2:
18.        if item not in list_1:
19.            list_2_only.append(item)
20.
21.    print('List 1 Only: ', str(list_1_only))
22.    print('List 2 Only: ', str(list_2_only))
23.
24. if __name__ == '__main__':
25.     main(sys.argv)
```

Code Explanation

Notice that the script expects `sys.argv[1]` and `sys.argv[2]` to be file paths. It reads both of them into lists using `f.read().splitlines()` and then it loops through each list looking for items that are not in the other list. It then prints the unique items found in each list.

Run the script with:

```
python compare_lists.py list_1.txt list_2.txt
```

It should output:

```
List 1 Only:  ['plumb', 'grape']
List 2 Only:  ['orange']
```

13.2 sys.path

Another useful attribute of the `sys` module is `path`, which contains a list of strings specifying the search path for modules. The list is os-dependent. To see your list, run the following code at your Python prompt:

```
import sys
sys.path
```

13.3 Conclusion

In this lesson, you have learned to create scripts that can be called from the computer's command line.



**7400 E. Orchard Road, Suite 1450 N
Greenwood Village, Colorado 80111
Ph: 303-302-5280
www.ITCourseware.com**