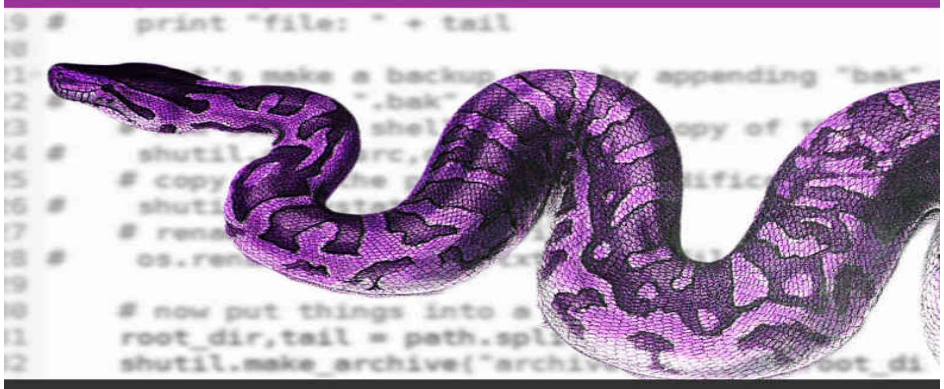
INTRODUCTION TO PYTHON PROGRAMMING

BEGINNERS' GUIDE TO COMPUTER PROGRAMMING AND MACHINE LEARNING

CRAIG KAPP

INTRODUCTION TO PYTHON PROGRAMMING

BEGINNERS' GUIDE TO COMPUTER PROGRAMMING AND MACHINE LEARNING



CRAIG KAPP

Introduction To Python Programming

Beginner's Guide To Computer Programming And Machine Learning

Craig Kapp

Table Of Contents

Module #1 (Variables, Statements, etc.)

Welcome to the first module for Introduction to Computer Programming! This module is designed to get you up and running with Python on your own computer as well as introduce you to a variety of elementary programming techniques. You should proceed through the content below in a linear fashion.

Installing Python

Directions for Microsoft Windows

1. Visit https://www.python.org/downloads/windows/
2. Download the latest Python 3 release (currently version 3.4.3 but the current version number may be higher) - do not download Python 2!
3. Once the download completes you can double-click on the file that was downloaded to begin the installation process
4. You will be asked if you want to run the file that was downloaded - click "Run"
5. Select "Install for all users" and click Next
6. You will then be asked to specify a destination directory for Python - you can just click Next here as well.
7. The following screen asks you to customize your version of Python - you can click Next here.
8. The installation wizard will begin installing Python onto your computer. You may be asked to confirm this - click "Yes" to continue
9. Finally, click the Finish button once the installation is complete.
10. You will be able to access Python by clicking on the Start button and then on All Programs->Python 3.4->IDLE (Python GUI)

Directions for Macintosh

1. Visit https://www.python.org/downloads/mac-osx/

2. Download the latest Python 3 release (currently version 3.4.3 but the current version number may be higher) - do not download Python 2!
3. Once the download completes you can double-click on the file that was downloaded to begin the installation process
4. Hold down the Control key on your keyboard and double-click on the "Python.mpkg" file. Click the "Open" button to continue.
5. An installation wizard will appear - click the Continue button three times. You will then be prompted to agree to a software license - click Agree.
6. Click the Install button.
7. You may need to provide your administrator password to complete the installation process.
8. You will be able to access Python by navigating to your Applications folder and finding the "Python 3.4 folder" - inside of this folder you will find a file labeled "IDLE.app" - double click on this file to launch the program
9. Note: if you are running Mac OS 10.9 (Mavericks) you may see the following error message when launching IDLE:

WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.

If you see this message you should quit IDLE and install ActiveTcl version # 8.5.*.* from this website http://www.activestate.com/activetcl/downloads. Note that the * characters here can be substituted with any number (i.e. currently the version available on the website is 8.5.18.0 but there may be a newer version available when you access this page). Make sure to install version # 8.5.*.* and NOT the latest version (any version that starts with 8.5 will solve the problem for you)

Using IDLE & Writing your first program

IDLE stands for "Integrated Development Environment" - it's designed to make it easy for you to both write and run Python programs.

IDLE has two modes – interactive mode and script mode. Interactive mode functions a lot like a calculator – you issue a command to the program, the command is executed and the result is immediately displayed. Script mode allows you to compose your code in a separate window and save it as a new text document on your computer. This document can then be read by IDLE at a later time and executed. You can run code written in Script mode by clicking on Run->Run Module

It's important to note that code is executed in sequential order – statements that appear at the top of your documents will run before statements that appear later.

Functions & Function Calls

A "function" is a pre-written block of computer code that will perform a specific action or set of actions. Python comes with a number of built-in functions, and you can also write your own (more on that later in the semester)

Functions always begin with a keyword followed by a series of parenthesis. For example:

```
print ()
```

You can "pass" one or more "arguments" into a function by placing data inside the parenthesis. For example:

```
print ("Hello World!")
```

Different functions expect different arguments. The `print` function, for example, expects zero or more arguments (multiple arguments are separated by commas) that will be printed to the screen.

When you ask Python to run a function we say that you have "called" the function.

Strings & String Literals

In a programming language data that is textual in nature (i.e. the phrase "Hello, World!") is called a "string". Strings can contain 0 or more printed characters. A string with zero characters is called an "empty string".

"String Literals" are strings that you define inside your program. They are "hard coded" values and must be "delimited" using a special character so that Python knows that the text you've typed in should be treated as printed text (and not a function call). For Example:

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

Python supports four different delimiters:

- The single apostrophe (')
- The quotation mark (")
- The triple quote (""")
- The triple apostrophe (''')

String literals must use the same delimiter at the beginning of the string as well as at the end of the string

Basic Formatting using the print() Function

The print() function has two default behaviors:

- It will always print a "line break" character at the end of a line
- It will always print a "space" character between multiple arguments

For example, given this program:

```
print ("Hello", "there")
print ("everybody!")
```

We would expect to see the output:

```
Hello there
everybody!
```

However, we can override this default behavior using a special set of "keyword" arguments. These arguments are called "sep" for "separator" and "end" for "line ending". Each of these arguments expects a String. For example, let's say you had a program where you wanted to print "*" characters instead of spaces between each item. You could do the following to make this happen:

```
print ("a", "b", "c", sep="*")
```

Note how the "sep" keyword is used here in conjunction with the equal sign - this is essentially telling Python to use the "*" character as the separator between each item.

Likewise, you can override the default behavior of the print() function to print a linebreak at the end of a line by using the "end" keyword argument, like this:

```
print ("a", "b", "c", end="#")
print ("d", "e", "f")
```

This will generate the following output:

```
a b c#d e f
```

... and if you don't want to print ANYTHING at the end of a line or between a character you can use an "empty string", like this:

```
print ("a", "b", "c", sep="")
```

Which will generate:

```
abc
```

You can combine these two techniques as well!

```
print ("a", "b", "c", sep="*", end="#")
```

Variables

A variable is a storage location and an associated symbolic name (an identifier) which contains some known or unknown quantity or information. Variables are one of the most fundamental aspects of computer programming. You can think of a variable as a "container" that temporarily stores information in your computer's memory.

You can create a variable by using the following syntax (similar to punctuation in a natural language):

```
variablename = somedata
```

The = symbol is called the 'assignment operator' and will cause Python to store the data on the right side of the statement into the variable name printed on the left side

When creating a variable you need to follow these naming rules:

- Variables can’t contain spaces (though you can use the underscore character ("_") in place of a space)
- The first character of a variable name must be a letter or the underscore character.
- No characters are allowed in a variable name besides alphanumeric (alphabet or numeric) characters and the underscore ("_") character (no special characters like &, !, +, $, etc.)
- Python is case sensitive, so two variables with the same name but different case (i.e. Craig vs craig) are two different variables
- You can't use any of Python's built in "reserved words" (i.e. you can't create a variable called "print" because that would conflict with the print function)

You can print the contents of a variable by simply printing the variable name, like this:

```
name = "John Smith"
print (name)
```

Output:

```
John Smith
```

Variables can "vary" over time, meaning that you can re-assign the value of a variable in your program and change the data that is being stored in that variable

You also need to make sure that your variables have been created before you use them. The following program will not run because the variable "foo" is not available when we attempt to access it in our program (it is declared AFTER we attempt to use it)

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

Output:

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

The "input" function

Sometimes you want to ask a user a question while your program is running. One of the simplest ways to do this is to request information from the keyboard using the `input` function.

`input` is a built-in function in Python. It accepts a single string as an argument. It then prompts the user with that string and waits for them to type in a series of characters. Your program will resume when the user hits the ENTER key. Whatever the user typed in during that time is sent back to your program as a string which can be stored in a variable. For example:

```
user_age = input("How old are you?")
```

The `input` function always "returns" a string. This means that when the function finishes running it will produce a string which can be assigned to a variable (using the assignment operator `=`)and used in your program.

Numeric Literals

When you want to store a number in a variable you can do so by placing the number on the right side of the assignment operator in the same way you would if you were storing a String.

```
number_of_items = 10
```

Note that you do not delimit numeric literals with quotation marks as you would with a string - simply type the number "as-is" and Python will interpret it as a numeric value as opposed to a sequence of characters.

We call numbers that are explicitly stated in your program a "numeric literal" - you literally are storing the numeric value specified.

Python supports two numeric data types - integers and floating point numbers. To create an integer literal simply type the integer, like so:

```
speed_of_light = 300000
```

To create a floating point literal you can type the number with its decimal point, like this:

```
cost_per_carrot = 1.99
```

Note that you cannot place formatting characters into a numeric literal (no `$` or `,` characters should be used when defining a numeric literal)

Math Operators

All programming languages have tools for manipulating numeric data which we call "math operators". Here is a list of some of the basic math operators that exist in Python:

```
+  Addition
-  Subtraction
*  Multiplication
/  Division (floating point division - fractional results included)
// Division (integer division - fractional results truncated)
```

We use operators along with numeric data to create "math expressions" that Python can evaluate on our behalf. Python can output the result of a math expression using the print function, like this:

```
print (5+2)
print (100 * 5 – 12)
```

We can also store the result of a math expression in a variable, like this:

```
answer = 5 + 2
print (‘the answer to 5 + 2 is’, answer)
```

Variables can also be used in a math expression in place of a numeric literal, like this:

```
price = 100.00 sales_tax = 0.07 total = price + sales_tax*price
```

Module #2 (Types, Operations, Debugging)

Commenting Your Code

Your programs will eventually get to be very, very long. Commenting is a way for you to leave "notes" for yourself or other programmers so that you (and/or others) understand what your thinking process was when you originally wrote your program

Python supports comments via the # symbol. You can use the # symbol to tell Python to essentially ignore anything else on that line which comes after the symbol. Here are some examples:

```
Print (‘Hello, world! ‘) # this will print the string Hello World
# the following line will print my name
print (‘Craig’)
```

It's important to get into the practice of commenting early and to be consistent about leaving comments throughout your program.

Python also supports multi line comments through the use of the triple quote delimiter. For example:

```
"""
This line will be ignored
So will this one And this one
"""
# the following line will NOT be ignored
print ('hi there')
```

We sometimes use comments as a way to **pseudocode** a programming project before we launch into writing it using real code. Think of this as writing a "rough draft" or "outline" of what you want your program to do before you actually write any real Python code. Pseudocoding allows us to

outline the necessary tasks without having to be burdened with writing the actual code to accomplish the desired tasks. For example:

```
# get two test scores from the user
# add them together and divide by 200
# print out the result and report the average
# score to the student
```

Data Types

A **value** is a fundamental unit of data that a program works with. The values we have seen so far have been numeric (i.e. 7, 5.5, etc.) or strings (i.e. "Hello!")

When storing data in memory, Python needs to keep track of the value (i.e. the number 5) as well as the "**data type**" of the value (in this case, an Integer). The "data type" describes the kind of data being stored, and allows you to perform certain actions on the value (i.e. you can add or subtract two Integer values)

There are three basic types of data that we will be working with during the first half of the term:

- Strings (character-based data)
- Numbers
- Logical Values (True / False)

Python has two different data types for numeric values:

- Integers
  - Whole numbers that do not contain a decimal point
  - Abbreviated as "int" in Python
  - Examples: 5, -5, 100, 10032
- Floating Point Numbers

- Numbers that contain a decimal point
- Abbreviated as "float" in Python
- Examples: 5.0, -5.0, 100.99, 0.232132234

You can store numeric data inside variables and Python will automatically select the correct data type for you. For example:

```
num_1 = 5                    # this is an int
num_2 = 4.99                 # this is a float
```

Keep in mind that you do not use separators (e.g. commas) or symbols (e.g. $, %, etc.) when storing numeric data. Example:

```
num_3 = $5,123.99            # error!
```

You can check the data type being stored by a variable by using the "type" function. The "type" function takes one argument (the value to be analyzed) and returns the data type as a string. Here's an example:

```
var1 = "Hello"
var2 = 5
var3 = 10.532
print (type(var1))
print (type(var2))
print (type(var3))
```

The result of the program above would be to display that var1 is a string, var2 is an Integer and var3 is a floating point number.

Data Type Conversion

Sometimes we need to convert values from one data type to another. For example, say you had the following variable in your program:

```
price = "1.99"
```

The data type of the variable "price" is a string (because it is initialized using quotes). However, let's say you want to perform a math operation

using this variable. In this program you couldn't do that because strings are not numbers - they are only sequences of characters.

We can convert the value stored in the variable "price" to a floating point number using a data type conversion function:

```
price = "1.99"
# convert price to a floating point number
real_price = float(price)
```

Python has a number of data type conversion functions built in. All of these functions accept a value and attempt to convert that value into the data type that corresponds with the name of the function. The result is returned out of the function. For example:

```
# define our variables
a = "1"                    # a is a string
b = 5.75          # b is a floating point number
c = 3                    # c is an integer
# convert data types
d = int(a)                  # d is now the integer value 1
e = str(b)                  # e is now the string "5.75"
f = float(c)                  # f is now the floating point value 3.0
```

str, float and int are data type conversation functions. They each take one argument and convert that argument into the desired data type.

Note that sometimes you can't convert from one data type to another. For example, what do you think would happen if you tried to run the following code? (Hint: Python won't be very happy with this!)

```
a = "hello world!"
```

```
b = float(a)
```

Remember that we can use the input function to ask the user a question from within our programs. However, the inputfunction returns a string – this means that the data type that comes out of the input function is a series of printed characters. If we want to use the value supplied by the user in a calculation we need to convert the string into one of the two numeric data types that Python supports (float and int). Here's an example:

```
# ask the user for their monthly salary
monthly_salary = input('how much do you make in a month?')
# convert the salary into a float
monthly_salary_float = float(monthly_salary)
# calculate the yearly salary
yearly_salary = monthly_salary_float * 12
# print the result
print ('that means you make', yearly_salary, 'in a year')
```

In the previous example we performed our data type conversion in two lines. We could have done that in a single line using a technique called "nesting". Here's an example:

```
mynum = float( input("give me a number!") )
```

In this example we execute the innermost function first (the input function) - this returns a string, which is then immediately passed as an argument to the float function, which converts the string to a floating point number. The result is stored in the variable "mynum"

More Math Operations & Mixed Type Expressions

Python contains two different division operators.

- The / operator is used to calculate the floating-point result of a division operation. The result is always a floating point value.

- The // operator is used to calculate the integer result of a division operation (essentially throwing away the remainder). This operation will always round down. The result is always an integer.

Most times you will use the floating point division operator (/). Here are a few examples of these operators in action:

```
print (5/2)                    # output:  2.5
print (5//2)                   # output: 2
print (-5/2)                   # output: -2.5
print (-5//2)                  # output: -3
```

Python supports the standard order of mathematical operations (PEMDAS). You can use parenthetical notation inside your math expressions to group operations For example:

```
((5+10+20)/60) * 100
```

You can raise any number to a power by using the "**" operator. For example:

```
answer = 3 ** 2
print (answer)                 # output: 9
```

The modulo operator (%) returns the whole number remainder portion of a division operation. For example:

```
print (5/2)                    # output: 2.5
print (5%2)                    # output: 1
```

Python allows you to mix integers and floating point numbers when performing calculations. The result of a mixed-type expression will evaluate based on the operands used in the expression. Python will automatically "widen" the data type of your result based on the values in your expression (i.e. if you add together an integer and a float the result will be a float since Python will need to preserve the decimal values from the floating point operand).

Operand #1	Operand #2	Result

int	int	int
float	float	float
int	float	float
float	int	float

Basic String Operations

You can't "add" two strings together, but you can "concatenate" them into a single compound string using the same symbol that is used for numeric addition. For example:

```
a = input("first name: ")
b = input("last name: ")
fullname = b + ',' + a
print (fullname)
```

You can also "multiply" a string by an integer value to produce a larger string. The resulting string concatenates the original string 'n' times, where n is the integer you use in your expression. For example:

```
lyrics = 'Fa ' + 'La ' * 8
print (lyrics)                    # Fa La La La La La La La La
```

Formatting strings and numbers

The `format` function can be used to format a string before you decide to print it out to the user.

`format` takes two arguments – a number and a formatting pattern (expressed as a string). `format()` returns a string which can be treated like any other string (i.e. you can print it out immediately, store its value in a variable, etc.)

The first argument passed to the format function is the item that you wish to format. The second argument passed to the function is the formatting "pattern" you wish to apply to this item. This pattern varies based on what you would like to do to the item in question. Once the pattern is applied to the item the format function will return a string version of your item with the formatting pattern applied.

The format function can be used to generate a customized string version of a float or integer number. For example, here's a program that truncates a floating point number to two decimal places:

```
a = 1/6
print (a)              # 0.16666666667, float
b = format (a, '.2f')
print (b)              # 0.17, string
```

In the example above we called the format function with two arguments - a variable of type float and a formatting pattern of type string. The formatting pattern ('.2f') means "we are formatting a floating point value (as indicated by the 'f' character). We want to limit the number of decimal places to 2 (as indicated by the '.2' before the 'f' character)"

Here are some additional examples of formatting numbers using the format() function:

```
a = 10000/6                          # float
b = format (a, '.2f')                # format a as a 2 digit number - b is a string
c = format (a, '.5f')                # format a as a 5 digit number - c is a string
d = format (a, ',.5f')               # format a as a 5 digit number with the "," separator between placed into the resulting string - b is a string
```

Another common use of the format function is to generate a string that contains a known # of characters. For example, say you have the strings "Craig" and "Computer Science". You might want to generate output that looks like the following given these items:

```
Name                    Department
Craig                   Computer Science
```

In this case we need to ensure that the strings "Name" and "Craig" are the same width so that the strings that come after them ("Department" and "Computer Science") line up correctly. You can use the format function to "pad" a string with extra spaces at the beginning or the end of the string. For example:

```
x = format('Craig', '<20s')
```

This will generate a new string (x) that contains the string 'Craig' plus 15 spaces at the end of the string (because "Craig" is 5 spaces the resulting string will have 15 additional spaces added to yield a string that contains 20 total characters). The total length of the string in this case will be 20 characters. The '<' character in the formatting pattern tells Python to left justify the string and place the extra spaces at the end of the new string. You can also have Python right justify the string and place the spaces at the beginning of the new string by doing the following:

```
b = format('Craig', '>20s')
```

Note that the format function always returns a string. This means that the result of the format function is unable to be used in a math expression. However, you could use a data type conversion function (int() or float()) to turn the string generated by the format() function back into a numeric data type if you wanted to.

Special Note: The format() function DOES NOT work in the web-based code editor that is included in these online modules. To work with the format() function you should launch IDLE and run your source code from within that IDE.

Errors, Error Types and Debugging

In general, there are three different kinds of errors that can occur to you while writing a program:

- **Syntax errors**: The code that you wrote does not follow the rules of the language. For example, a single quote is used where a double quote is needed; a colon is missing; a keyword is used as a variable name. With a Syntax error your program will not run at all (IDLE will yell at you before it even attempts to run your program)
- **Runtime errors**: In this case, your code is fine but the program does not run as expected (it "crashes" at some point as it is running). For example, if your program is meant to divide two numbers, but does not test for a zero divisor, a run-time error would occur when the program attempts to divide by zero.
- **Logic errors**: These can be the hardest to find. In this case, the program is correct from a syntax perspective; and it runs; but the result is unanticipated or outright wrong. For example, if your program prints "2 + 2 = 5" the answer is clearly wrong and your algorithm for computing the sum of two numbers is incorrect.

Here are some suggestions for isolating errors and correcting them:

- Set small, incremental goals for your program. Don't try and write large programs all at once.
- Stop and test your work often as you go. Celebrate small successes!
- Use comments to have Python ignore certain lines that are giving you trouble
- Use the `print` function to print out information so that you can inspect what is happening at key points in your program

Module #3 (Boolean Logic, Using Modules)

In this module you will learn how we can design programs that let Python ask 'questions' - in essence, this lets us write programs that have the ability to make decisions based on certain conditions. To do that we'll need to understand the concepts of boolean logic and a decision structure known as an `if`statement and its related statements. In addition we'll introduce you to some basic Python Modules that add additional functionality to make our programs more interesting.

Boolean Data

So far we have discussed three different "data types" in Python - 'strings' for sequences of characters, 'integers' for whole numbers and 'floats' for numbers that contain a decimal point. The video below will introduce you to a fourth data type called the 'Boolean' data type. This data type is used to store 'logical' values - that is, values that can be one either `True` or `False`. We use logical values all the time in our program to determine if a particular condition exists (i.e. did the user type their username into a form correctly? The answer to that question is either `True` or `False`).

In addition to the 'string', 'int', and 'float' data types you've already learned about, there is another data type called the Booleans data type (bool for short). **A Boolean is a value of either `True` or `False`.**

Note: capitalization is important - `True` is not the same as `true`.

Recall that all expressions in Python evaluate to a value that has a data type. For example, the addition problem in the following expression evaluates to an 'integer' because both of the operands are 'integers':

```
print (5 + 7)
```

A **Boolean expression** is an expression that evaluates to a value that has a data type of Boolean (i.e. `True` or `False`). We can write Boolean expressions using **comparision operators** which allow us to compare two values.

There are six comparision operators that we use in Python - the table below gives examples of how each of the comparision operators can be used:

```
x == y     # EQUALITY:                is x equal to y?
x != y     # INEQUALITY:              is x not equal to y?
x > y      # GREATER THAN:            is x greater than y?
x >= y     # GREATER THAN OR EQUAL TO: is x greater than or equal to y?
x < y      # LESS THAN:               is x less than y?
x <= y     # LESS THAN OR EQUAL TO:   is x less than or equal to y?
```

Any expression that uses a comparision operator evaluates to a Boolean value. For example, `5 > 3` asks a question - "is 5 greater than 3?" - the answer here is "Yes", so the expression evaluates to the Boolean value of `True`

Note: the = symbol is the assignment operator used to assign values to variables, while == is a comparison operator used to check for equality. The two are not interchangeable.. Also note that there is no =<, =>, or =! operators, the non-equals symbols (!, <, and >) always come first when writing a comparision operator.

Logical Operators

A logical operator is an operator that lets you combine multiple boolean expressions. There are three logical operators: and, or, and not.

For example, assume you have a variable called 'x' that is holding the value 5:

```
x = 5
```

If we wanted to test to see if the value being held in the variable 'x' is between 3 and 7 we would need to test two different conditions:

```
is x >= 3? is x <= 7?
```

However, each one of these expressions only tests one condition - the first one determines if x is greater than 3, and the second one determines if x is less than 7. We actually need the answer to both of these questions to be True to determine if a number is between 3 and 7. We can do this using the and logical operator, like this:

```
x > 3 and x < 7
```

Python first evaluates each of the Boolean expressions independently. Next, it sees the "and" logical operator between them - this "connects" the two expressions and says "if the first expression evaluates to True AND the second expression evaluates to True then the whole expression evaluates to True"

The semantics (the meaning of) the logical operators in Python are the same as they are in a natural language (a spoken language). Here are a few tables that show how these operators work:

"and" operator

```
# the "and" operator is the most restrictive logical operator
# it will evaluate to True only if all of its operands also evaluate to True
Operand 1          Operand 2          Result
True               and                True               True
True               and                False              False
False              and                True               False
False              and                False              False
```

"or" operator

```
# the "or" operator is less restrictive than the "and" operator
# it will evaluate to True if any one of its operands evaluete to True
Operand 1          Operand 2          Result
True               or                 True               True
True               or                 False              True
```

False	or	True	True
False	or	False	False

"not" operator

```
# the "not" operator negates a Boolean expression
# it "flips" True's to False's and False's to True's
```

Operand 1	Result
not True	False
not False	True

Here are some examples:

```
x < 0 or x > 10     # True if x is less than 0 *or* x is greater than 10;
                    # otherwise False
x > 0 and x < 100   # True if x is greater than 0 *and* x is less than 10;  # otherwise False
not x < 0           # True if x is *not* less than 0; otherwise False
```

Order of operations

When we see an expression that has multiple operators (any combination of mathematical, comparison, or logical operators) we need to know what order to perform those operations. The order is determined by predefined precedence given by the following table which is ordered from the highest precedence (done first) to the lowest precedence (done last):

Order of Operations	**Operator(s)**	**Operator Type**
Done 1st	( )	specify order of operations
2nd	**	exponentiation
3rd	*, /, //, %	multiplicative
4th	+, -	addition
5th	==, !=, <=, >=, <, >	comparison
6th	not	logical
7th	and	logical
Done last	or	logical

Given this order of operations notice the equivalence relationship between the following statements:

```
# Following two lines are equivalent:
```

```
x-1 >= 2 and y*3 <= 4
((x-1) >= 2) and ((y*3) <= 4)
# Following three lines are equivalent:
x == 1 and not y == 2
(x == 1) and not (y == 2)
(x == 1) and (not (y == 2))
```

Conditional Statements

Conditional statements (also known as "selection statements" or "if statements") are statements that execute different lines of code based on a Boolean condition. These kinds of statements allow us to have our programs "ask questions" and make their own decisions while the program is running.

One-way "if" Statements

Conditional statements are written in Python using the if keyword followed by a Boolean expression and a colon. An ifstatement will execute the statements inside of its body given a certain condition is True. Let's begin with a simple "one-way 'if' statement" example:

```
x = 5
# evaluate whether x is holding a value > 0
if x > 0:
  print (“x is positive! ”)

# after the ‘if ‘ statement (note the indentation)
print (“This line will always print”)
```

Output appears below:

```
x is positive!
This line will always print
```

In this example we are evaluating a condition using an if statement. This tells Python to compute the result of a Boolean expression - if the result is True then the code that is indented under the if statment will be executed. Note the 'colon' character at the end of the line that contains the if statement - this is required by Python and will generate a syntax error if it is omitted. Note that the print statement that is not indented under the if statement will always execute since it is not part of the "block" of code indented under the if statement.

There can be multiple statements inside a code block and these statements are executed in sequential order (one after another).

Note: Indention is critical! Python needs you to consistently indent your code so that it can determine which statements are associated with which if statements.

Two-way "if" statements

If the condition attached to an if statements evalutes to True then the code block indented under that if statement is executed. However, there are times when we want to run a different block of code if the result of the Boolean expression isFalse We can do this by using the optional else statement.

The else statement is positioned after the if statement at the same level of indentation. If the Boolean condition associated with the if statement evaluates to True then Python will execute the code indented under the if statement and skip the else statement. However, if the condition evalutes to False then Python will skip the if statement and execute the else statement instead. Here's an example:

```
x = -5

# evaluate whether x is holding a value > 0
if x > 0:
        print ("x is positive!")

# if the condition above evalutes to False then we should
# execute the block associated with the "else" statement
else:
        print ("x is less than 0!")

# after the 'if' statement (note the indentation)
print ("This line will always print")
```

Output appears below:

```
x is less than 0!
This line will always print
```

Multi-Way "if" Statements

There are times when a two-way if statement won't work because the quesiton you are asking has more than two possible answers. We can do this in Python by using a "multi-way 'if' statement" which is implemented using the optional elifkeyword. In essence, this keyword lets us ask "follow-up" questions after we ask the intial question using an if statement. Here's an example:

```
x = 0

# evaluate whether x is holding a value > 0
if x > 0:
        print ("x is positive!")
```

```
# if the above condition evalutes to False then we will
# ask a "follow-up" question using the "elif" statement
elif x == 0:
        print ("x is zero!")

# if the condition above evalutes to False then we should
# execute the block associated with the "else" statement
else:
        print ("x is less than 0!")

# after the 'if' statement (note the indentation)
print ("This line will always print")
```

Output appears below:

```
x is zero!
This line will always print
```

Nested "if" Statements

Sometimes you will want to ask additional questions inside of the body of another if statement. We can do this by "nesting" an if statement inside of another if statement. Here's are a few examples of this process:

```
x = 7

# determine if x is positive

if x > 0:

        print ("x is postiive!")

        # now determine if x is even

        if x % 2 == 0:

                print ("Evenly divisible by 2! x is even!")

        # if it's not even then it has to be odd!

        else:

                print ("Not evenly divisible by 2! x is odd!")

# note that this "else" is attached to the "if" above at

# the same level of indentation

else:

        print ("x is not positive")
```

Output appears below:

```
x is postiive!
Not evenly divisible by 2! x is odd!
```

Note: with each layer of nesting, the indention amount increases by a constant amount. You can tell which else blocks match up with which if blocks by seeing which if statements line up vertically with which else statements.

Short-Circuit Evaluation

When we have a Boolean expression that uses and or or, most programmming languages iwill use what is called **short-circuit evaluation** to evaluate the expression in the most efficient way possible.

In programming, certain Boolean expressions can be fully evaluated without having to execute every single part of the statement. For example, consider the following example:

```
A > B or C > D
```

If Python executes this statement it will begin by computing A > B, which can be either True or False. Remember that the or operator will yield a True result if **either** statement it connects evaluates to True. So if A > B evaluates to Truethen there is really no need to move on and evaluate C > D - so Python skips it and moves on!

Likewise, consider the following statement:

```
A > B and C > D
```

Python will begin to evaluate this expression just in the previous example, but in this case we are connecting the two sub-expressions with the and operator. If the first expression evaluates to False then there is really no need to continue - there's no way the expression can evaluate to True since the and operator will only yield a True result if **both** Boolean expressions evaluate to True.

Basic Python Modules

The Python programming language contains a core set of functions that you can freely use in your programs, such as printformat and input. However, there are times when you will want to access more "specialized" functions that are not part of the core Python package. For example, let's say you are writing a video game that needs to compute the angle between two characters on the screen. To do this you would probably need access to some basic trigonometry functions such as sin,cos and tan - but these functions are not built into Python. Thankfully Python supports the ability to import specialized function libraries, know as "modules", into your programs. These modules give you access to functions that are not normally available to you in Python.

Importing and Using Modules

To use a module we first have to **import** them into our own programs. Importing a module tells Python that we're going to need to access that module and all of the functions that it contains. In order to import a module into your program you first need to issue the import command - this command tells Python to load the module into memory and make available all of the functions in that module. Note that this command should happen at the very beginning of your program - the first few lines of your code should contain all of your "import" statements. Here's an example that imports the "math" module into a program:

```
# ask Python to give us access to all of the functions in the "math" module import math
```

Next, we will probably want to use the functions that exist within our newly imported module. We can do this using the period character (.) along with the name of the module to tell Python to run a function that exists within tha tmodule. We call this "dot syntax" - here's an example:

```
# ask Python to give us access to all of the functions in the "math" module import math

# now run the "sin" function inside of the "math" module

# note how we have to first tell Python to look inside of the "math" module

# then we use the "dot" character followed by the name of the function we want to access x = math.sin(3.14159)
```

How do you know what functions are available inside of a module? One great way to explore the functions inside of a module is to refer to Python's online documentation website which is abailable at http://docs.python.org/3/. This page contains everything you need to know about how Python works, but if you're interested in learning more about a specific module you can click on the Global Module Index link and then scroll down to the module you are interested in. If you look at the documentation for the Math Module, you'll notice that it provides a lot of mathematical functions and constants, including:

- math.sin: computes the sine of a floating point number (returns a float)
- math.radians: converts degrees to radians (returns a float)
- math.pi: the constant PI computed to 15 decimal places

If you're using IDLE in "Interactive" mode you can also access the documentation straight from there by running the following command:

```
help("modules")     # gets a list of available modules on your system

help("math")       # gets the documentation for the Math module
```

The Math Module

The math module contains a number of math-related functions and "constants" (read-only variables that you often need to use, such as the number Pi). Here are a few examples:

```
# make all of the functions in the "math" module available to this program
import math

# print some mathematical constants
# constants are considered "read-only" variables - you can't change them
print("PI is: ", math.pi)
print("e is: ", math.e)

# call some mathematical functions
print("The square root of 100 is", math.sqrt(100))
print("30 degrees is", math.radians(30), "radians")

# we can also nest together calls to module functions
print("The sine of 90 degrees is:", math.sin(math.radians(90)))
```

Output appears below:

```
PI is:  3.14159265359
e is:  2.71828182846
The square root of 100 is 10
30 degrees is 0.523598775598 radians
The sine of 90 degrees is: 1
```

The Random Module

Random numbers are often used in computer programs to give your programs the ability to make their own decisions. For example, let's say you were writing a program to play a game of Rock-Paper-Scissors with the user. We could easily ask the user which symbol they wanted to play using the input function, but how can the computer pick its symbol? One answer would be to have the computer pick a random number between 1 and 3. If the computer picks 1, then we can say that it picked 'Rock' - if it picked 2, 'Paper' - and if it picked 3, 'Scissors'. By ascribing meaning to these numbers we can give our programs the illusion of being able to "think" for themselves!

Here are some common examples of how we could use random numbers in a Python program:

- To play a game of chance where the computer needs to throw some dice, pick a number, or flip a coin
- To shuffle a deck of playing cards
- To randomly place a new enemy character in a game
- To simulate possible rainfall amounts in a computerized model for estimating the environmental impact of building a dam
- For encrypting your banking session on the Internet

Python provides a Random module that helps with tasks like this. You can take a look at it in the documentation.

The easiest way to generate a random number is to use the random.randint function. This function requires two argumenst - a low bound and a high bound. It then returns a random integer within this range (inclusive of the end points) - here's an example:

```
import random

# get a random number between 1 and 3
# it could be 1, 2 or 3 - the function will decide when it runs
num = random.randint(1,3)
```

If you need to generate a random floating point number you can use the random.random function. This function returns a floating point number in the range [0.0, 1.0) — the square bracket means "closed interval on the left" and the round parenthesis means "open interval on the right". In other words, 0.0 is a possibility, but all returned numbers will be strictly less than 1.0 (i.e. 1.0 will never occur). It is usual to scale the results after calling this method, to

get them into a range suitable for your application (i.e. multiplying this number by 10 will generate a random between 0.0 and 10.0).

It is important to note that the Python random number generator is based on a deterministic algorithm which means that it is both repeatable and predictable. This is why we call this method of generating random number "pseudo-random" generator — the numbers that are generated are not really random at all - they are actually based off of a "seed" value which is usually determined by your computer's built in clock. Each time you ask for another random number, you'll get one based on the current seed (the system clock), and the state of the seed (which is one of the attributes of the generator) will be updated. The good news is that each time you run your program, the seed value is likely to be different meaning that even though the random numbers are being created algorithmically, you will likely get random behavior each time you execute.

Alternate Import Option

The preferred method of importing a module into your program is to use the import command at the top of your program - you can then use "dot syntax" along with the name of your module to access functions organized within that module. However, there is another way to access modules which is sometimes used - this way is generally frowned upon as it clutters your programs namespace (what functions and data are defined in the current environment), but you should be aware of it as you may see it used elsewhere.

Instead of importing the module, you can import specific items into your programs namespace. The syntax is one of the following:

```
from MODULE_NAME import ITEM_NAME
from MODULE_NAME import ITEM_NAME,ITEM_NAME,...
from MODULE_NAME import *
```

This last option uses what we call a **wildcard**. A wildcard is a character that means it can match anything, on most systems that character is the asterisk (*). If you use the wildcard you're importing all the items from the entire module.

Here are some examples:

```
# import all of the functions from the "random" module
from random import *
# call the "randint" function - note how we don't need to call it using
# the module name (i.e. random.randint(1,10) can be written as randint(1,10)
print(randint(1,10))
```

```
# import just two functions from the "math" module
from math import pow,sqrt
# now we can call these functions without prefacing them with the "math." prefix
print(pow(2,16))
```

```
print(sqrt(16))
```

Module #4 (while Loops)

Repetition Structures and "while" Loops

When writing a program you will find that you will often need to repeat certain statements over and over again. For example, say you wanted to write a program to calculate commission on sales for a small sales team. You could easily do this for one person by using a sequence structure like the following:

```
# get sales figures
sales = float(input("how much did they make: "))
rate  = float(input("sales rate: "))
# calculate
print ("you owe them: ", sales * rate)
```

However, for 3 people this will begin to get a little more challenging, as the only way you could do this using a sequence structure is to copy and paste the code multiple times:

```
# get sales figures
sales = float(input("how much did they make: "))
rate  = float(input("sales rate: "))
# calculate
print ("you owe them: ", sales * rate)
# get sales figures
sales = float(input("how much did they make: "))
rate  = float(input("sales rate: "))
# calculate
print ("you owe them: ", sales * rate)
# get sales figures sales = float(input("how much did they make: "))
```

```
rate  = float(input("sales rate: "))
# calculate
print ("you owe them: ", sales * rate)
```

There are several disadvantages to this approach.

- Your programs will tend to get very large
- Writing this kind of program can be extremely time consuming
- If part of the duplicated code needs to be corrected then the correction must be implemented many times

One solution to this kind of problem is to use a repetition structure, which involves writing the code for a given operation one time and then placing it into a special structure that causes Python to repeat it as many times as necessary. We call this a "repetition structure" or, more commonly, a "loop."

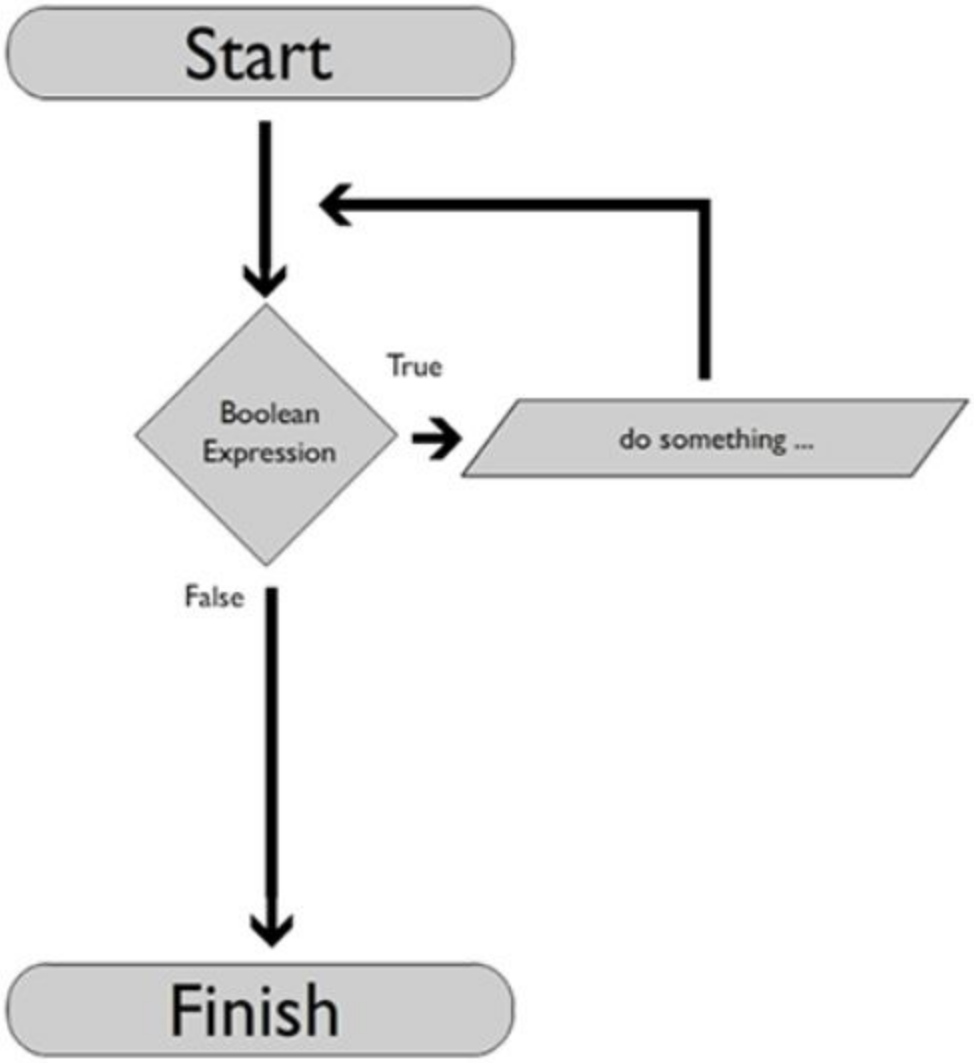
There are a few different kinds of loops in Python. This module discusses the `while` loop which is known as a “condition controlled loop” -- this means that the looping behavior of the structure is dependent on the evaluation of a condition (i.e. a Boolean expression). The syntax for a condition controlled loop is almost identical to the `if` statement that we covered in the previous module. For example:

```
while condition:
  print (“this will continue to print over and over again”)
  print(“… as long as the condition above evaluates to True”)
```

In Python, a `while` loop works as follows:

1. Evaluate a Boolean expression
2. If it is False, skip the block of statements associated with the while loop
3. If it is True
    - Execute the block of statements associated with the while loop
    - Go back to step 1

Here's what this looks like as a flowchart - note that the structure is almost identical to an if statement, but with a while loop we re-evaluate the condition being tested until it eventually evaluates to False.



The trick with while loops is to control the condition that is being evaluated. One way to do this is to create a variable which can be used in a boolean expression to determine if a loop needs to iterate again.

Here is the "commission calculator" program written using a while loop:

```
# set up our loop
```

```
# assume that the manager wants to calculate sales
# for their employees (our "control variable")
c = 'yes'
# loop until the manager no longer wants to
# calculate sales - we will re-evaluate this
# condition every time the "while" loop iterates
while c == 'yes':
    # get sales figures
    sales = float(input("how much did they make: "))
    rate  = float(input("sales rate: "))
    # calculate
    print ("you owe them: ", sales * rate)
    # ask the manager if they want to continue
    c = input('Do you want to continue? Type yes or no ')
print ("thanks for using our program")
```

We refer to the process of going through a loop as an “iteration,” so if a loop cycles through 5 times then we say we have “iterated” through it 5 times

The while loop is considered a “pre-test” loop, meaning that it only iterates upon the successful evaluation of a condition. This means that you always need to “set up” your loop prior to Python being able to work with it (i.e. setting up a control variable).

Loops are often used in conjunction with variables to control the number of times they iterate. Often we want to make changes to a variable inside of a loop (for example, to compute the running total of a series of numbers). We can do this by setting up a variable outside of our loop and then referencing that variable inside of our loop. We call these "accumulator variables." For example:

```
# set up a variable
```

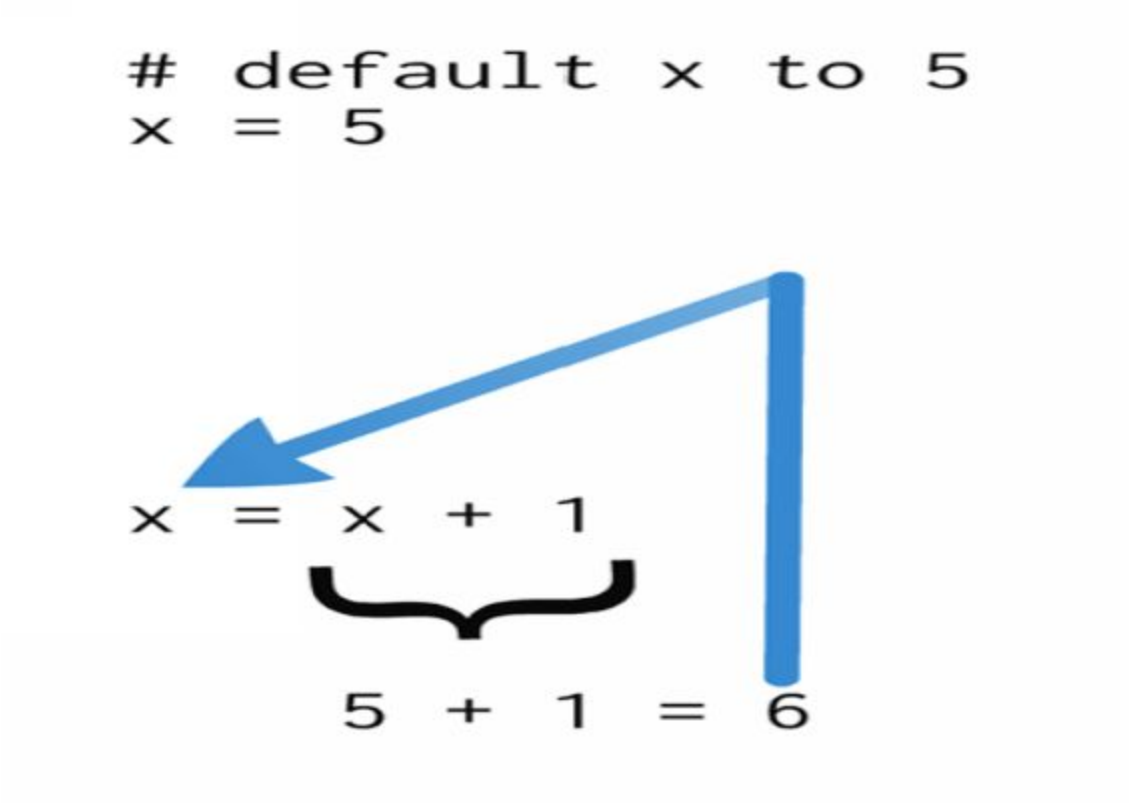
```
a = 0

# enter into a loop
while a < 5 :

        # add 1 to “a” – this is what we call a “self referential assignment”
        # we are essentially adding 1 to the current value of  “a” and then
        # storing the result back into the variable
        a = a + 1

        print (a)
```

The self-referential assignment statement works as follows:



Python also supports a series of "shortcuts" for self-referential assignments:

Operator	Usage	Equal to
+=	c += 5	c = c + 5
-=	c -= 2	c = c – 2
*=	c *= 3	c = c * 2
/=	c /= 3	c = c / 3
%=	c %= 3	c = c % 3

while loops are often used because they execute an indeterminate number of times. You can cause a while loop to execute for a set # of times (i.e. loop exactly 5 times) by using an accumulator variable. For example:

```
# set up a counter variable to keep track of
# how many times we have looped
numloops = 0

# create a loop that keeps going as long as
# our "numloops" variable is less than 5 – this will
# evaluate to True the first time we encounter the loop
while numloops < 5 :
        # update our "numloops" variable to make a note
        # that we have looped one more time
        numloops += 1

        # do something
        print ("Iteration #", numloops)
```

Another example:
This program uses a "while" loop to execute a "print" statement 10 times.

```
# define our accumulator variable
counter = 0

# enter into a while loop as long as our accumulator variable is < 10
while counter < 10 :
```

```
    print (“Hello! Counter is currently”, counter)

    # increase counter by 1
    counter += 1
```

Output appears below:

```
Hello!  Counter is currently 0
Hello!  Counter is currently 1
Hello!  Counter is currently 2
Hello!  Counter is currently 3
Hello!  Counter is currently 4
Hello!  Counter is currently 5
Hello!  Counter is currently 6
Hello!  Counter is currently 7
Hello!  Counter is currently 8
Hello!  Counter is currently 9
```

"while" loop control

There is always a danger with while loops that the condition associated with the loop will never evaluate to False. If that happens, the loop will continue running forever. We call this an "infinite loop". If you do end up with an infinite loop, you can force Python to break out of it by hitting Control->C on your keyboard. Here's an example of an infinite loop:

```
a = 5

# the value stored in the variable “a”
# never change – it’s a;ways less than 10!
while a < 10 :
        print (“ I’m looping!”)
```

And here’s another example:

```
# no way to ever get out of this loop!
# True will never be anything but True, which
# means the condition attached to this loop will
# never be False!
while True :
        print (“I’m looping!”)
```

There are two special commands called break and continue that you can use inside of a while loop to further control the flow of repetition. These commands can only be used inside of a while loop - they won't work anywhere else in your program.

The break command is used to immediately end a loop. Once a break statement is encoutered the loop immediately stops and Python resumes execute on the line directly after the end of the loop.

Sample program : This program uses the “break” statement to prematurely end a loop.

```
# create an accumulator variable
counter = 0

# enter into an into an infinite loop
while True :
        # add 1 to our counter variable
        counter += 1

        # if the counter is >= 5 , we can “break” out of the loop and immediate end it!
        if counter >= 5 :
        print (“Breaking the loop!”)
        break
```

```
        # note that this line won't print when counter == 5 because we "broke" the loop
        # in the "if" statement above
        print ("counter is:" , counter)
```

Output appears below:

```
counter is: 1
counter is: 2
counter is: 3
counter is: 4
Breaking the loop!
```

The continue command is used to cause the loop to immediately cease its current iteration and re-evaluate its condition. The continue command doesn't end the loop -- it simply causes the loop to "jump" to the next iteration. If the condition evaluates to True the loop iterates again, but if it evaluates to False it will not.

Sample program: This program uses the "continue" statement to end a loop iteration when a condition is met.

```
# create an accumulator variable
counter = 0

# enter into an infinite loop
while True:

        # add 1 to our counter variable
        counter += 1

         # if the number is even we should skip it and cause the loop to re-iterate
        if counter % 2 == 0:
                print ("Even number!", counter, " -- skipping!")
                continue

        print ("The number is:", counter)
```

```
        # if the counter is >= 5, we can "break" out of the loop and immediately end it!
        if counter >= 5:
            print ("Breaking the loop!")
            break
```

Output appears below:

```
The number is: 1
Even number! 2  -- skipping!
The number is: 3
Even number! 4  -- skipping!
The number is: 5
Breaking the loop!
```

Boolean variables are often used with `while` loops to control their execution. We sometimes refer to these as "flags" and we can easily check their value (`True` or `False`) to determine if we should allow a loop to continue iterating or not.

Sample program: This program demonstrates how a Boolean variable can be used as a “flag” to control a `while` loops.

```
# a program that keeps looping as long as the user supplies even numbers

# assume that we should continue to loop
keeplooping = True

# keep looping as long as our "flag" is set to True
while keeplooping:
        # get a number from the user
    number = int(input("Enter a number: "))

    # is this an even number?
```

```
    if number % 2 == 0:
            print ("This is an even number!")

    # is this an odd number?
    else:
            print ("This is an odd number!  No more loops for you!")

            # update our "flag" to prevent another iterate
            keeplooping = False
```

Module #5 (For Loops, Nested Loops)

In this module you will learn about another way to 'repeat' blocks of code using the `for` loop. We'll also learn how we can handle more complex situations like iterating over 2 or 3-dimensional data using nested loop structures. And finally we'll discuss how to make our programs more robust using simple data validation.

The `while` loop structure that we learned about in Module 4 is especially useful when iterating an unknown number of times. However, when you want to write code that iterates a fixed number of times it's usually more convenient to use a "count controlled loop." In Python this kind of loop is called a `for` loop.

A `for` loop can iterate over any fixed set of items. Those items can be as simple as a series of numbers, or more complex like a series of strings or other data types. A `for` loop has the following form:

```
for VARIABLE in ITEMS:
    STATEMENTS
```

In plain English this could be read as "for each VARIABLE that exists in ITEMS execute STATEMENTS." In this example the words in UPPERCASE will be replaced by the data you're working with. You can create a list of items by placing values separated by commas in between an opening and closing square brace like so: `[ VALUE_1, VALUE_2, ..., VALUE_N ]`. The code in STATEMENTS is the body of the `for` loop. The code in the body is going to be run once for each item in the list. In addition, each item in the list is temporarily stored in the specified VARIABLE for one iteration (execution) of the body.

Let's take a look at a concrete example to make it more clear:

```
# Basic for loop with a list of numbers
for i in [ 10, 20, 30, 40, 50 ]:
    print("The value of i is:", i)
```

Output appears below:

```
The value of i is: 10
The value of i is: 20
The value of i is: 30
The value of i is: 40
The value of i is: 50
```

Note: Previously we said that you should use descriptive variable names (i.e. names that actually describe the data they hold). You will notice though that we used the variable 'i' in our code above. The 'i' in this case is short for "iterator" or "integer" and is considered a standard coding convention to use, so it is acceptable in this case.

You must have at least one statement in your for loop, but you can have as many additional statements as you want. You can also use accumulators (like you did with while loops) or other variables from outside of your for loop as well.

```
# For loop with multiple statements and an accumulator
total = 0
for i in [ 10, 20, 30, 40, 50 ]:
    total += i
    print("The value of i is:", i)
print("The sum of all the values is:", total)
```

Output appears below:

```
The value of i is: 10
The value of i is: 20
The value of i is: 30
The value of i is: 40
The value of i is: 50
The sum of all the values is: 150
```

The for loops in Python can iterate over any type of data making it possible to use with strings, floats, characters, etc. In addition we can nest other programming structures inside of a for loop, so we can include statements like if, elif, else statements inside of a for loop. With each layer of nesting, we have to make sure our indention amount is equivalent to the level of nesting that we want in our program.

```
# For loop using strings and if statements
for fruit in [ "apples", "plums", "oranges", "lemons", "grapefruit", "limes" ]:
    print("Did you know: ")

    # Apples and plums get a special message,
    # all other fruits get a standard message.
    if (fruit == "apples"):
        print("  there are ~7,500 varieties of apples in the world?")
    elif (fruit == "plums"):
        print("  prunes are just dried plums?")
    else:
        print("  " + fruit + " are considered a citrus fruit?")

    # print a blank line after each fruit fact
    print()
```

Output appears below:

```
Did you know:
  there are ~7,500 varieties of apples in the world?

Did you know:
  prunes are just dried plums?

Did you know:
  oranges are considered a citrus fruit?

Did you know:
  lemons are considered a citrus fruit?

Did you know:
  grapefruit are considered a citrus fruit?

Did you know:
  limes are considered a citrus fruit?
```

The range() function

In this module, the first program we showed you looped through the set of integers from 10 to 50 skipping 10 each time. But what if we want to iterate over a much larger set of numbers (say, the numbers 1 through 100)? We could spend our time writing out every single number we want to iterate over inside of a list, but that would be a very inefficient use of our time. Instead we can use a new function called range to generate a series of numbers for us.

The simplest form of the range() function is to pass it one parameter. For example:

```
range(10)
```

will return the series of numbers:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

So we see that range(N) returns the set of numbers from 0 inclusively (including that number) to N exclusively (excluding that number).

Note: When representing a range of numbers in mathematical notation, we might use the symbols [and] to mean inclusivity, while the symbols (and) would mean exclusivity. Some examples:

- [1,5] would give the numbers: 1, 2, 3, 4, 5
- [1,5) would give the numbers: 1, 2, 3, 4
- (0,4) would give the numbers: 1, 2, 3
- (0,4] would give the numbers: 1, 2, 3, 4

We now have a quick and easy way to get a large list of numbers:

```
# Sums the values from 0 to 9,999 inclusively
total = 0
for i in range(10000):
    total += i    # "total += i" is shorthand for "total = total + i"
print("Sum of all numbers from [0,10000]:", total)
```

Output appears below:

```
Sum of all numbers from [0,10000]: 49995000
```

Using `range()` with a starting point

You may not always want to start counting at `0`. Let's take the following situation for example. Suppose you want to ask the user what number to start and end? You could write code like this:

```
start = int(input("Enter starting number"))
end = int(input("Enter ending number"))
for i in range(end + 1):      # +1 to make it inclusive
    if i < start:
      # ignore values less than the starting point
      continue
    total += i
    print("Sum from", start, "to", end, "inclusively:", total)
```

The code works, but it's not very pleasant to read and there is a better way. The `range()` function is pretty flexible and has some extra functionality built-in to it that we can take advantage of. It has what we call "optional" parameters. We can pass the `range()` function both a starting and a stopping value in the form `range(START_VALUE, END_VALUE)`.

Important: The order of these parameters is critical - the starting number must be supplied first and the ending number must be supplied second.

We can simplify our code from above to incorporate these values and get rid of the if statement:

```
start = int(input("Enter starting number"))
end = int(input("Enter ending number"))
for i in range(start, end + 1):      # +1 to make it inclusive
    total += i
```

```
    print("Sum from", start, "to", end, "inclusively:", total)
```

Using range() with a step size

If we want to find all the multiples of 5 from 1 to 50, we could have the following code:

```
for i in range(1,50+1):
if i % 5 == 0:
  print(i, "is a multiple of 5")
```

As we said before, the range() is very flexible, it actually has a way for us to use a step size in how much i will be incremented with each iteration. For that, we include a 3rd parameter so the syntax becomes:range(START_VALUE, STOP_VALUE, STEP_SIZE). In that case, our above code can be simplified to:

```
for i in range(1, 50+1, 5):
  print(i, "is a multiple of 5")
```

Using range() in reverse

So far we have show you how to count from a starting point and approaching an ending point and we have seen that by default the step value for each iteration is 1, but that value can be modified. Let's say that we want to write a countdown timer for a rocket launch system. We want that count to start at 10 and count down to 0. To do that, we can just set the step value to a negative value, so that it is subtracting instead of adding with each iteration!

Note: When using a negative step size, you still need to keep your inclusivity and exclusivity rules in mind as they don't change!

Sample Program: counts backwards using range()

```
# Countdown before a rocket launch

print("Starting countdown!")
```

```
for i in range(10,0,-1):
    print(i)
print("BLASTOFF!!!!!")
```

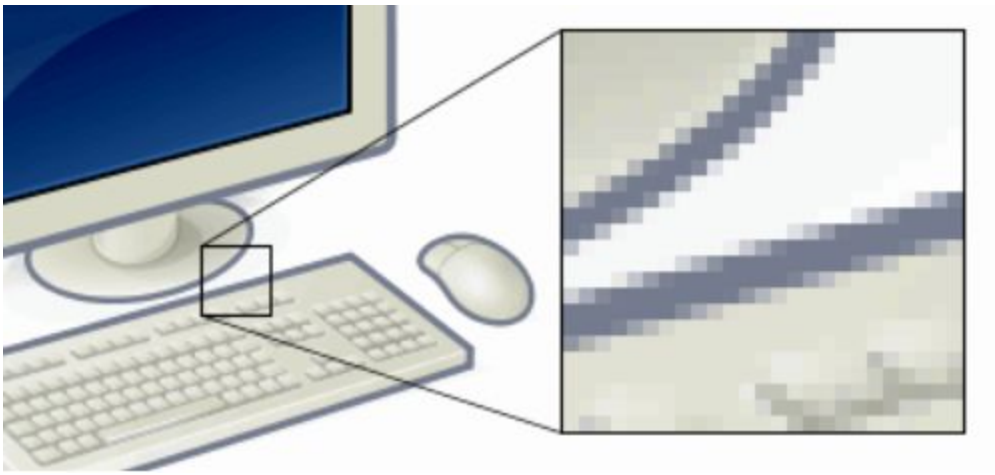
Output appears below:

```
Starting countdown!
10
9
8
7
6
5
4
3
2
1
BLASTOFF!!!!!
```

Nested Loops

A "nested" loop is a loop that has been placed inside of the body of another loop. The most common form of nested loop structures is probably a `for` loop nested inside of another `for` loop, but you can also have any set of nested combinations of structures such as: a `for` loop inside of a `while` loop, a `while` loop inside of a `for` loop, or a `while` loop inside of a`while` loop.

Nested `for` loops can be extremely useful when dealing with multi-dimensional data. A good example is any time you need to work with data that has two or more pieces of information per entry. Let's take an image for example. Every computer image we look at on a screen is made up a of a series of colored squares called "pixels" that are arranged as a grid. A pixel is the smallest representable part of an image. If we zoom in on a picture we can see the individual pixels as block like structures. Here is an example:



In this example, we will use some pseudocode to demonstrate how we could iterate through each pixel in an image.**Pseudocode** is a design tool that we use as programmers to lay out the structural design of a program, without worrying about details of a particular language. Pseudocode tends to look a lot like Python code because Python code was significantly influenced by the simplicity of it.

```
# To process the entire image one pixel at a time, we first subdivide
# the problem into processing each row of pixels, one row at a time.
# process each row individually
for every row in image_pixels:
        # Process an entire row, by iterating
        # over the row one pixel at a time
        for every pixel in row:
         # analyze and process a single pixel in the row.
         # after analyzing, the for loop will move to the next pixel
# if no more pixels are in the row, it jumps back to the outer loop
        # When a row has been completed,
        # the program will move to the next row
        # if all the rows have been completed, the outer for loop will finish
# At this point all the pixels will have been processed
```

Another example of using nested loops might be to ask the user for a number of years - we can then iterate over these years and then iteratve over the months in those years. Here's an example:

```
for year in range(2015, 2017):
    print ("Year: ", year)
    # now iterate over months in this year
    for month in range(1, 13):
        print ("Month #: ", month)
```

Simple Data Validation

All of us have had programs crash on us or do things that we may not have expected. Often these are results of bugs in the program. When we design programs we want to write programs that are **robust** so that they crash as rarely as possible. Finagle's Law states: "Anything that can go wrong, will —at the worst possible moment." Our programs should be ready to handle these situations in a graceful way. That means that we should employ some amount of defensive programming when we write code. One way of writing code defensively is to check that the data we're receiving is within the range of valid values.

Take a look at this sample code:

```
numerator = input("Enter a numerator: ")
denominator = input("Enter a denominator: ")
result = numerator/denominator
print(numerator, "divided by", denominator, "is:", result)
```

If the user enters a denominator of 0, the program will crash:

```
Enter a numerator: 10
Enter a denominator: 0
Traceback (most recent call last):
  File "code01.py", line 3, in
    result = numerator/denominator
ZeroDivisionError: division by zero
```

Not only do these kinds of errors crash the program, but the error messages for the user can be very confusing and are full of extraneous information that they don't need to know about (but are helpful in debugging).

A better way to handle the situation is to check the data before you use it using the if statement to make sure it's within the proper range of values.

Sample Program: This program shows how you can perform simple data validation to check that invalid data is not being used in a calculation.

```
# Simple data validation example, avoids dividing a number by zero.
numerator = int(input("Enter a numerator: "))
denominator = int(input("Enter a denominator: "))
if (denominator != 0):
    result = numerator/denominator
    print(numerator, "divided by", denominator, "is:", result)
else:
        print("Can't divide by 0!")
```

Or better yet, we can **wrap the code** that asks the user for a value inside of a while loop so that it keeps asking the user until we receive a valid input.

Below are a couple ways this can be done. Compare the different ways and think about how each functions differently and what advantages each may have:

```
# ask the user for a numerator
numerator = int(input("Numerator: "))

# get a denominator
denominator = int(input("Denominator: "))

# keep asking the user to enter a denominator if they enter a zero
while denominator == 0:
        print ("Denominator cannot be zero!")
        denominator = int(input("Denominator: "))

# output
print (numerator, "/", denominator, "=", numerator/denominator)
```

Module #6 (Functions)

In this module you will explore how to write your own custom functions in Python. Functions are often used as a technique to break up complicated problems into simple, reusable blocks of code.

Basic User Defined Functions

> A function is a group of statements that exist within a program for the purpose of performing a specific task. Since the beginning of the semester, we have been using a number of Python's built-in functions, including the print, len, and range functions.

As you continue to write programs, they will inevitably become more and more complex. In order to help organize and make sense of such a large amount of code, programmers often organize their programs into smaller, more manageable chunks by writing their own functions. This lets us break down a program into several small functions, allowing us to "divide and conquer" a programming problem. Some benefits to using functions include:

- **Simpler code**: programs tend to look "cleaner" when broken down into smaller, more manageable tasks
- **Reusable code**: once you write a function you can use it any number of times in your program without having to "copy and paste" the same blocks of code over and over again
- **Easier Testing**: you can easily comment out a function all to "turn off" large portions of your program, allowing you the ability to isolate areas of your code that may need additional attention
- **Collaboration**: with well-defined functions you can easily divide up the work on a large programming project among a group of programmers and then use those functions as part of a larger software system.

Defining your own functions is easy in Python. To begin, you need to make up a name for your function. Functions follow the same naming rules that exist for variables, which include:

- You cannot use any of Python's keywords
- No spaces
- The first character must be A-Z or a-z or the "_" character
- After the first character you can use A-Z, a-z, "_" or 0-9
- Uppercase and lowercase characters are distinct

Defining functions requires the def keyword followed by the name of the function you wish to define. Next, you need to add two parenthesis after the name of the function and the "colon" character. From there you can indent any number of lines of code within your newly created function. Here's an example function definition:

```
# this defines a function called "myfunction"
def myfunction():
        print ("Printed from inside a function")
```

Defining a function doesn't run the code inside of the function - it simply tells Python that you have created a function with a particular name. To actually use your function you need to call it, much like how you have to call a pre-written function like theprint function before you can use it. Here's an example:

```
def myfunction():
        print ("Hello, world!")
##### main program #####
# call the function that we just wrote
myfunction()
```

As you can see, functions must be defined before they can be used. The following code wouldn't work in Python because the function "myfunction" doesn't exist yet when it is called:

```
myfunction()
def myfunction():
    print ("Hello, world!")
```

In Python we generally place all of our functions at the beginning of our programs or in an external module.

Once a function has completed, Python will return back to the line directly after the initial function call. For example:

```
def fun1():
        print ("fun 1!")

print ("hello there!")
fun1()
print ("middle of the program")
fun1()
print ("all done!")
fun1()
```

Output appears below:

```
hello there!
fun 1!
middle of the program
fun 1!
all done!
fun 1!
```

When a function is called programmers commonly say that the “control” of the program has been transferred to the function. The function is responsible for the program's execution until it runs out of statements, at

which time control is transferred back to the point in the program that initially "called" the function.

```
# define a function to draw the "top" of a rectangle
def top():
        print ("*****")
# define a function to draw the "sides" of a rectangle
def side():
        print ("*   *")
# draw a rectangle!
top()
for x in range(5):
        side()
top()
```

Output appears below:

```
*****
*   *
*   *
*   *
*   *
*   *
*****
```

Function Arguments & Variables

You can define variables inside functions just as you would in your main program. However, variables that are defined inside of a function are

considered **local** to that function, which means that they only exist within that function and are no longer available once the function has completed executing. We refer to these variables as **local variables**.

Code that is written outside of a function will not be able to access a local variable created inside of a function. Here's an example of how this works - the code below will result in an error because the main program is trying to access a local variable that was defined inside of a function:

```
# define a function that asks the user to enter a
# number of items they wish to purchase
def purchase():
        numitems = int(input("How many items do you want to buy? "))

    print (numitems)
# call the purchase function
purchase()
print (numitems) # error!  Variable numitems doesn't exist in this scope!
```

Different functions can have their own local variables that use the same name. Even though the variables share the same name, they are considered completely different variables that have no relationship to one another.

```
def newjersey():
    num = 1000
    print ("NJ has", num, "restaurants")
def newyork():
    num = 2000
    print ("NY has", numbugs, "restaurants")
newjersey()
newyork()
```

Sometimes it is useful to not only call a function but also send it one or more pieces of data as "arguments." For example, when you call the print function you can send it an argument, like this:

```
print("hello!")
```

We can do the same thing with our own functions, like this:

```
my_awesome_function("hello!")
```

When passing arguments, you need to let your function know what kind of data it should expect to receive. You can do this by establishing a variable name in the function definition - these variable names go inside of the set of parenthesis after the function name.

```
# function that accepts one argument - a
def my_function(a):
```

Argument variables are automatically declared every time you call your function, and will assume the value of the argument passed to the function. Here's an example function that accepts a value and stores it in the local variable "num":

Sample Program: This program demonstrates how you can call a function that is designed to accept an argument.

```
def square(num):
        # num assumes the value of the
        # argument that is passed to
        # the function (5)

        print ("You sent me", num)
        print (num, "raised to the 2nd power is", num**2)

# call the function with the number 5
```

```
# which will be sent into the function and assumed
# by the local variable "num"
square(5)
```

Output appears below:

```
You sent me 5
5 raised to the 2nd power is 25
```

You can actually pass any number of arguments to a function. One way to do this is to pass in arguments "by position." Here's an example:

```
# the average function expects 3 arguments
def average(num1, num2, num3):
        sum = num1+num2+num3
        avg = sum / 3
        print (avg)
# call the function with 3 arguments
average(100,90,92)
```

In the above example, the function argument variables (num1, num2 and num3) assume the values of the arguments passed into the function (100, 90, and 92)

When we pass an argument to a function in Python we are actually passing its "value" into the function, and not an actual variable. We call this behavior "passing by value." We are essentially creating two copies of the data that is being passed – one that stays in the main program and one that is passed as an argument into our function. This behavior allows us to set up a "one way" communication mechanism – we can send data into a function as an argument, but the function cannot communicate back by

updating or changing the argument variable because it is "local" to the function.

Sample Program: This program demonstrates how arguments allow you to "pass by value" - the argument variable is merely a "copy" of the data being sent into the function.

```
# define a function that accepts a single argument and updates it to a new value

def change_me(v):

        print ("function: I got:", v) # will print 5

        # change the argument variable

        v = 10

        print ("function: argument is now:", v) # will print 10

# main program

myvar = 5

print ("starting with:", myvar)    # will print 5

change_me(myvar)

print ("ending with:", myvar)      # will print 5
```

Output appears below:

```
starting with: 5
function: I got: 5
function: argument is now: 10
ending with: 5
```

When you create a variable inside a function we say that the variable is "local" to that function. This means that it can only be accessed by statements inside the function that created it. When a variable is created outside all of your functions it is considered a "global variable" — global variables can be accessed by any statement in your program, including by statements in any function. All of the variables we have been creating so far in class up until this module have been global variables.

Sample Program: This program demonstrates how a global variable can be viewed from within a function

---

```
def showname():
        print ("Function:", name)
# the "name" variable is global to the entire program (it is defined without
# any indentation so it can be "seen" by all parts of the program, including functions)
name = 'Craig'
print ("Main program:", name)
showname()
```

Output appears below:

```
Main program: Craig
Function: Craig
```

However, if you want to be able to change a global variable inside of a function you must first tell Python that you wish to do this using the global keyword inside your function. If you don't use this keyword, Python will assume that you want to create a new local variable with the same variable name as the global variable. Technically this isn't an error, but it can get confusing!

Sample Program: This program demonstrates how a global variable can be viewed and changed from within a function

---

```
# name is a global variable
name = 'Craig'

def showname():
        # this line lets us change the value of name inside this function
        # there is only one variable called "name" in this program - it is
        # a global variable that is defined outside of this function
    global name
    print ("Function 1:", name)
    name = 'John'
    print ("Function 2:", name)
print ("Main program 1:", name)
showname()
print ("Main program 2:", name)
```

Output appears below:

```
Main program 1: Craig
Function 1: Craig
Function 2: John
Main program 2: John
```

Function Return Values

"Value returning functions" are functions that have the ability to send back, or return, information to the part of your program that called them. We've been using value returning functions since the beginning of the semester. For example, the "input" function is a value returning function - it "returns" a string when it is finished executing. We often capture that string and store it in a variable for later use in our program.

```
name = input("What is your name? ")
```

Syntactically value returning functions are almost identical to the types of functions we have been writing so far. The only difference is that you need to include a return statement in your function to tell Python that you intend to send back a value to whatever part of your program called the function. Here's an example function that returns an integer to the caller:

```
def myfunction():
    x = 5
    return x
somenumber = myfunction()
print (somenumber)
```

When a function reaches a return statement it immediately ends the function and sends back the value attached to the statement. For example, this function will end early if you send it an even number as an argument because the return statement is reached prior to the second print statement:

Sample Program: The return statement will immediately cause a function to end and send back whatever value is attached to the return statement.

```
def demo_function(a):
        print ("In the function")
        if a % 2 == 0:
```

```
        return "even"

    print ("If this prints then we haven't hit a return statement yet!")

    return "odd"

answer1 = demo_function(3)

print ("Return value #1:", answer1)

print ()

answer2 = demo_function(4)

print ("Return value #2:", answer2)
```

Output appears below:

```
In the function
If this prints then we haven't hit a return statement yet!
Return value #1: odd

In the function
Return value #2: even
```

Note that in the above example we are storing the return value in a variable. This is functionally identical to when we store the string result that gets returned from the input function or the integer result that gets returned from the len function.

Sample Program: This program contains two functions - one function is used to ask the user for a price value and another function is used to apply a discount to a price value. Both functions are then used as part of a larger program.

```
# this function gets a price value from the user

# and returns it to the portion of the program that called the function
```

```
def getprice():
        userprice = float(input('price? '))
        return userprice
# this function accepts a user price as an argument and computes a 20% discount
# the discount is then returned to the portion of the program that called the function
def apply_discount(userprice):
        discount = userprice - userprice*0.2
        return discount
#### main program ####
price = getprice()
newprice = apply_discount(price)
print ("starting price:", price)
print ("discount:", newprice)
```

In Python, functions can return more than one value. You can do this by separating the return values using a comma. Remember that if you return more than one value then your calling program must be expecting the same number of values. Here's an example:

Sample Program: This program shows how a function can return multiple values in Python. Note that if you return multiple values from a function you also need to capture those values when calling the function. You can do this by separating the variables used to capture the return values using commas.

```
def powers(a):
```

```
        pow_2 = a**2
        pow_3 = a**3
        pow_4 = a**4
        return pow_2, pow_3, pow_4
# send 1 argument to the function and receive 3 return values
x,y,z = powers(2)
print (x,y,z)
```

Output appears below:

```
4 8 16
```

Module #7 (Strings, Sequences, Slicing)

In this module, we will discuss more in depth the string object and it's capabilities, along with the properties of iterable objects. In addition we'll learn about how we can take advantage of "slicing" to simplify access to specific iterable objects.

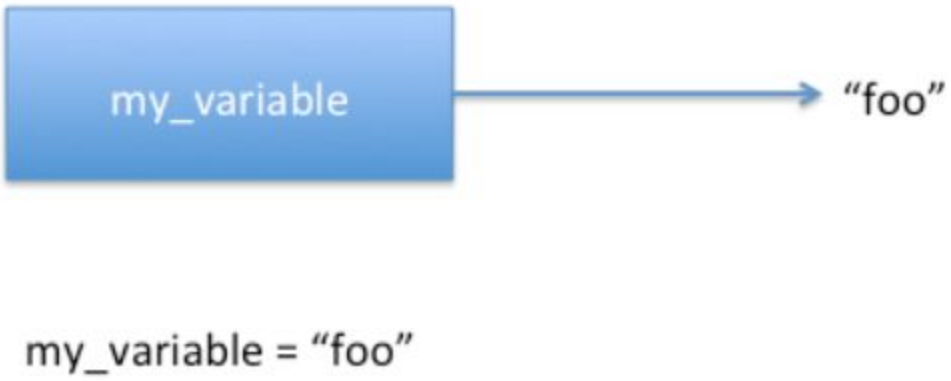
String Basics

As you know, a string is what we usually refer to as text. More specifically a string is what we call an "object" which in its simplest form is a set of data and the "methods" associated with that data. The characters are the data associated with that string. A "method" is just fancy word for a function associated with a specific type of object.

When a new string literal is created, it is "immutable" (incapable of being modified). Conversely, a piece of data that is capable of being modifiable is called "mutable". If we want to change the data a variable stores, we instead create a new string literal and update the variable so it points to the new string literal. In the following example it is worth noting that the original data has not changed, instead the variable now references a new string with a different value.

```
my_name = "Andrew Case"
my_name = "Andrew I. Case"
```

Here is a graphical example that demonstrates the updated reference:



This difference may seem irrelevant now but it becomes relevant when we talk about string methods that seem like they would change the text of a string, but instead only return a new string with different text.

The text associated with each string object is what we call a "sequence" of characters. Let's think for a moment about how a computer represents these characters.

Characters

A character (or "chr" as Python refers to them) is either a single letter (alpha), number (numeric), symbol, or a special control character (e.g. \n for new lines). Many control characters are not able to be printed to the screen, but are usually inputable via the keyboard in some way (e.g. "ctrl", "backspace", or "new line" for example). Although we claim that we store characters, computers really only store and work with numbers. To get around this limitation, we have an encoding or one-to-one mapping that matches each character up to a specific numerical value. A "plain-text" file (source code for example) is a file that is comprised of text that is encoded using the ASCII (short for "American Standard Code for Information Interchange") encoding scheme (or a slight derivative of it). Below is a table that shows each printable character along with its encoded numerical equivalent:

- special control characters

- alphanumeric printable characters
- non-alphanumeric printable characters

ASCII value	Character Value
0	NULL (Null character)
1	SOH (Start of Header)
2	STX (Start of Text)
3	ETX (End of Text)
4	EOT (End of Transmission)
5	ENQ (Enquiry)
6	ACK (Acknowledgement)
7	BEL (Bell)
8	BS (Backspace)
9	HT (Horizontal Tab)
10	LF (Line feed)
11	VT (Vertical Tab)
12	FF (Form feed)
13	CR (Carriage return)
14	SO (Shift Out)
15	SI (Shift In)

16	DLE (Data link escape)
17	DC1 (Device control 1)
18	DC2 (Device control 2)
19	DC3 (Device control 3)
20	DC4 (Device control 4)
21	NAK (Negative acknowledgement)
22	SYN (Synchronous idle)
23	ETB (End of transmission block)
24	CAN (Cancel)
25	EM (End of medium)
26	SUB (Substitute)
27	ESC (Escape)
28	FS (File separator)
29	GS (Group separator)
30	RS (Record separator)
31	US (Unit separator)
32	(space)
33	! (exclamation mark)

34	" (Quotation mark)
35	# (Number sign)
36	$ (Dollar sign)
37	% (Percent sign)
38	& (Ampersand)
39	' (Apostrophe)
40	((round brackets or parentheses)
41	) (round brackets or parentheses)
42	* (Asterisk)
43	+ (Plus sign)
44	, (Comma)
45	- (Hyphen)
46	. (Full stop , dot)
47	/ (Slash)
48	0 (number zero)
49	1 (number one)
50	2 (number two)
51	3 (number three)

52	4 (number four)
53	5 (number five)
54	6 (number six)
55	7 (number seven)
56	8 (number eight)
57	9 (number nine)
58	: (Colon)
59	; (Semicolon)
60	< (Less-than sign)
61	= (Equals sign)
62	> (Greater-than sign)
63	? (Question mark)
64	@ (At sign)
65	A (Capital A)
66	B (Capital B)
67	C (Capital C)
68	D (Capital D)
69	E (Capital E)

70	F (Capital F)
71	G (Capital G)
72	H (Capital H)
73	I (Capital I)
74	J (Capital J)
75	K (Capital K)
76	L (Capital L)
77	M (Capital M)
78	N (Capital N)
79	O (Capital O)
80	P (Capital P)
81	Q (Capital Q)
82	R (Capital R)
83	S (Capital S)
84	T (Capital T)
85	U (Capital U)
86	V (Capital V)
87	W (Capital W)

88	X (Capital X)
89	Y (Capital Y)
90	Z (Capital Z)
91	[(square brackets)
92	\ (Backslash)
93	] (square brackets)
94	^ (Caret or circumflex accent)
95	_ (underscore or understrike)
96	` (Grave accent)
97	a (Lowercase a)
98	b (Lowercase b)
99	c (Lowercase c)
100	d (Lowercase d)
101	e (Lowercase e)
102	f (Lowercase f)
103	g (Lowercase g)
104	h (Lowercase h)
105	i (Lowercase i)

106	j (Lowercase j)
107	k (Lowercase k)
108	l (Lowercase l)
109	m (Lowercase m)
110	n (Lowercase n)
111	o (Lowercase o)
112	p (Lowercase p)
113	q (Lowercase q)
114	r (Lowercase r)
115	s (Lowercase s)
116	t (Lowercase t)
117	u (Lowercase u)
118	v (Lowercase v)
119	w (Lowercase w)
120	x (Lowercase x)
121	y (Lowercase y)
122	z (Lowercase z)
123	{ (opening curly brackets or braces)

124	| (vertical-bar, vbar, vertical line
125	} (closing curly brackets or braces)
126	~ (Tilde)
127	DEL (Delete)

We can take any character and convert it to its numerical equivalent using the built-in `ord()` (short for "ordinal value of") function. The `ord()` function takes one parameter which is the character we want to get the ordinal value of and it returns the character for that numerical value. Here's an example from the Python Shell:

```
>>> ord('A')
65
>>> ord('a')
97
```

If you compare these numbers to the ASCII chart above, you'll see that the value `"65"` is an encoded value for the upper case letter `"A"`. And `"97"` is the encoded value for the lower case letter `"a"`.

We can also do the reverse and take any ASCII numerical value and convert it into its character equivalent using the built-in `chr()` function.
The `chr()` function takes the numerical value you want to convert as its only argument and it returns the character for that numerical value. Here's an example from the Python Shell:

```
>>> chr(90)
'Z'
>>> chr(122)
'z'
```

String indexing

Because strings are just a series of characters, we can find each character individually using what's called an "index". Given a string called

"my_string" which stores "Paddington" the corresponding index for each character is as follows:

index:	0	1	2	3	4	5	6	7	8	9
my_string =	P	a	d	d	i	n	g	t	o	n

So the first character is the '0' index and then each character after it increments that index by one. We can gain access to characters individually using square brackets to specify a specific character index using the form str[index]. Here's an example where we're accessing the first three characters (indices 0, 1, and 2) and building a new string by concatenating those three characters:

```
>>> my_string = "Paddington"
>>> new_string = my_string[0] + my_string[1] + my_string[2]
>>> print(new_string) Pad
```

Python also allows you to use negative indices which allow you to index characters starting from the end of a string. So each character in a string actually has two indices associated with them (a positive and a negative index):

index:	0	1	2	3	4	5	6	7	8	9
my_string =	P	a	d	d	i	n	g	t	o	n
negative index:	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

If you wanted to access characters starting from the end, you can use negative indices the same way as positive indices. Here is an example of creating a new string comprised of the last three characters concatenated in reverse order:

```
>>> my_string = "Paddington"
>>> new_string = my_string[-1] + my_string[-2] + my_string[-3]
>>> print(new_string)
not
```

Getting the length of a string

Python has a built-in function called len() that can be used with various data types to get the length of that data. This length function can be used to

return the number of characters in a string. Its syntax is len(str) and it returns the number of characters in the "str" string past to it. Here is an example:

```
>>> str = "Paddington is my cat"
>>> string_length = len(str)
>>> print('The string "' + str + '" is', string_length, 'characters long.')
The string "Paddington is my cat" is 20 characters long.
```

The len(str) function can be used with any type of sequence object (you'll learn about other sequence types like "lists" and "dictionaries" in another learning module).

Sample program: the following example shows a simple way of indexing through a string and then printing the ASCII value for each character. It then does it by iterating through the string in reverse. Once you understand the code, try updating the code that that it only prints the values for every other character.

```
# Get string from othe user
user_string = input("Enter a string to get the ASCII values: ")
str_length = len(user_string)

# Print string's ASCII values
for i in range(str_length):  # i by convention is short for "index"
    c = user_string[i]       # c by convention is short for "character"
    ascii_value = ord(c)     # get the ordinal value
    print("character '" + c + "' has the ASCII value", ascii_value)
# Print string's ASCII values in reverse
print("Starting from the end...")
# starting index -1, going to the last index, incrementing by -1 each time.
for i in range(-1, -(str_length+1), -1):
    c = user_string[i]
    ascii_value = ord(c)
```

```
    print("character '" + c + "' has the ASCII value", ascii_value)
```

Output appears below:

```
character 'N' has the ASCII value 78
character 'o' has the ASCII value 111
character 'n' has the ASCII value 110
character 'e' has the ASCII value 101
Starting from the end...
character 'e' has the ASCII value 101
character 'n' has the ASCII value 110
character 'o' has the ASCII value 111
character 'N' has the ASCII value 78
```

String Methods

When using built-in functions, the data you are working with is passed as a parameter to the function such as `len(str)`. When working with objects (strings are objects in Python) we mentioned that they store data along with extra functions that work with that data. We call functions that are part of an object type a "method." To call a method, we access it using the object itself. For example, if working on a string, we call a method using the form `str.method(...)` where `str` is the string variable or literal and `method` is the name of the method you're calling. In this module, we will explain many of the methods related to strings and how to use them. There are many more methods that are beyond the scope of this document. For a full hist of Python string methods, you can visit the Python Documentation or run `help(str)` on the Python Shell.

Critical: The examples below assume that `str` and `substr` are the string variables or literals that you are working with.

Boolean Checks

There are several methods that can tell us information about a string using boolean responses. Here a few important methods:

`str.isalpha()` returns `True` if `str` is all alphabet characters; otherwise `False`.

Examples of usage from a Python Shell:

```
>>> dog_name = "Anna"
>>> dog_name.isalpha()
True
>>> "Anna is my dog".isalpha()  # spaces are not in the alphabet
False
```

Note: You can access these method using either variables (as in the first example above) or literals (as in the second example above).

`str.isnumeric()` returns `True` if `str` is all numeric characters; otherwise `False`.

Example:

```
>>> dog_age = "14"
>>> dog_age.isnumeric()
True
>>> dog_age = "fourteen"
>>> dog_age.isnumeric()
False
```

`str.isalnum()` returns `True` if `str` is all alphabet or numeric characters; otherwise `False`.

Example:

```
>>> cat = "Paddington"
>>> cat.isalnum()
True
>>> cat= "P@ddington"
>>> cat.isalnum()
```

```
False
```

`str.isspace()` returns `True` if `str` is all space characters; otherwise `False`.

Example:

```
>>> text = "    "
>>> text.isspace()
True
>>> text = " text with spaces  "
>>> text.isspace()
False
```

Working with substrings

A substring is a string that is a subset (contained inside) of another string. For example, given the string "my cat's name is paddington", both of the strings "cat" and "paddington" are substrings of the original string. A The following methods deal with with finding and counting substrings.

`str.find(substr)` returns the index of the first occurring `substr` in `str`.

`str.count(substr)` returns the number of occurrences of `substr` in `str`.

Sample Program: example usage of the `count()` and `find()` methods. Once you understand the code, update it so that the search string is inputted from the user instead and then test it using different strings.

```
# initialize strings
my_string = "I have a dog named Anna.  I also have a cat named Paddington."
search_string = "named"

# check for strings
first_occurrence = my_string.find(search_string)
```

```
num_occurences = my_string.count(search_string)

# print results
print('Found', num_occurences, 'instance(s) of "' + search_string + '".')
print('The first occurrence was at index:', first_occurrence)
print('The character at this index is:', my_string[first_occurrence])
```

Output appears below:

```
Found 2 instance(s) of "named".
The first occurrence was at index: 13
The character at this index is: n
```

Note: If you are only trying to determine if a substring exists inside of a string, you can use the `in` keyword which will return a boolean value of `True` or `False`. Here's an example:

```
if (substr in str):
    print(substr + " is in " + str)
else:
  print(substr + " is NOT in " + str)
```

Modifying Strings

There are many methods that allow us to create new string objects that are based on the string being accessed.

Critical: Keep in mind that strings are immutable and therefore the literal storage for each string can not be modified. None of these methods modify the string being accessed, but instead return a new string. If you want to change the text of a variable, you can set that variable to the new string returned by that method.

`str.find(substr)` returns the index of the first occurring `substr` in `str`.

`str.lower()` returns a copy of `str` with all the characters converted to lowercase.

`str.upper()` returns a copy of `str` with all the characters converted to uppercase.

`str.capitalize()` returns a copy of `str` with the first character in uppercase.

`str.replace(oldsubstr, newsubstr[,count])` returns a new copy of `str` with all occurrences of`oldsubstr` replaced by `newsubstr`. If the optional argument `count` is given, only the first `count`occurrences are replaced.

Sequences and Iterables

In Python, a "sequence" is an object that has a series of ordered items. An "iterable" is an object that is capable of returning its members one at a time. All sequences are iterables and therefore because a string is a sequence of characters, a string is also iterable. Being iterable means that we can use it whenever we want to iterate over that object.

Sample Program: Looping through each character of a string:

```
user_string = input("Enter a string to iterate over: ")
for character in user_string:
    print(character)
```

More interestingly, lets say we want to take a message from the user and encode that message using the ASCII table:
Sample Program: Converting a string to ASCII values:

```
user_string = input("Enter a string to iterate over: ")
for character in user_string:
    print(ord(character)) # print out each character's ordinal value
```

Slicing

Slicing is a technique used to select a series of items from a sequence. Slicing works in a similar fashion to indexing. Using indexing, you can select one item from a sequence of items:

```
>>> my_string = "supercalifragilisticexpialidocious"
>>> my_string[0]
's'
```

Often though we will want to access a subset of a sequence. Slicing allows us to select a series of items from a sequence in one easy statement. The basic form for slicing is `seq[start:end]` where `seq` is a sequence data type such as a string, `start` is the starting index of items to obtain, and `end` is the ending index (exclusively) of the items to obtain. Here's an example:

```
>>> my_string = "supercalifragilisticexpialidocious"
>>> my_string[0:5]
'super'
```

You also have the option of specifying a step size for the iteration of characters (just like with the `range()` function) in the form `seq[start:end:step_size]`. So you can use code like the following: >>> my_string = "supercalifragilisticexpialidocious" >>> my_string[0:5:2] 'spr' Any or all of the parameters (start, end, step_size) can be omitted and a default value will be used in its place. The default starting value is index 0. The default end index is the length of the sequence. The default step size is 1.

Sample Program: Try to predict the results of the following examples before you run it.

```
digits = "0123456789"
```

```
print(digits[:])
print(digits[::])
print(digits[::2])
print(digits[0:])
print(digits[0::])
print(digits[:9:])
print(digits[:10:])
print(digits[:11:])
print(digits[0:10:2])
print(digits[1:10:2])
```

Output appears below:

```
0123456789
0123456789
02468
0123456789
0123456789
012345678
0123456789
0123456789
02468
13579
```

Module #8 (Lists)

Lists are the primary 'sequence object' in Python which allows you to store an almost infinite number of values within a single identifier. This module introduces you to the basics of using Lists in Python as well as a number of different kinds of problems that can be easily solved using a variety of list-specific functions and methods.

Sequence Objects

The programs that we have written so far have been designed to operate using textual data (strings), logical data (booleans) and numeric data (integers and floating Point numbers). These data types have the ability to represent and store one piece of data at a time - for example:

```
x = 5                    # integer
y = 5.0                  # floating point number
z = 'hello'              # string
q = True                 # boolean
```

However, there are times when we need to write a program that keeps track of many values at the same time. For example, if we wanted to write a program to keep track of final exam scores for a group of 50 students in a class we would need to create 50 different variables, like this:

```
test_01 = 95.45
test_02 = 89.35
test_03 = 76.43
...
...
test_50 = 97.11
```

One way to solve this problem is to use a "sequence" data type, which has the ability to hold multiple values within the same identifier. In many programming languages we call these "arrays," but in Python we refer to this data type as a `list`.

List Basics

An easy way to think about a list in Python is to think about how a book operates in the real world. A book is a single object (i.e. "Harry Potter and the Chamber of Secrets") that can contain any number of sub-items (i.e. pages).

You can create a list in Python by using square brackets, which almost look like the covers of a book in the real world. For example:

```
my_book = ["Page 1", "Page 2", "Page 3"]
```

The above code will create a new list in Python that holds three strings – "Page 1", "Page 2" and "Page 3" – in that order.
Lists can contain any data type that we have covered so far - for example:

```
my_list = [100, 200, 300]
```

Lists can also mix data types.

```
my_list = ['Craig', 5.0, True, 67]
```

You can print the value of a list using the print function. This will print all of the values stored in the list in the order in which they are represented:

```
my_list = ["a", "b", "c"]
```

```
print (my_list)
```

Just like with a string, you can use the repetition operation (*) to ask Python to repeat a list. For example:

```
my_list = [1, 2] * 3
print (my_list)
```

```
#>> [1, 2, 1, 2, 1, 2]
```

You can use the concatenation operation (+) to ask Python to combine lists, much like how you would combine two strings. For example:

```
my_list = [1, 2] + [99, 100]
print (my_list)
```

```
# >> [1, 2, 99, 100]
```

Indexing List Elements

In a book, you can reference a page by its page number and in a list you can reference an element stored in a list using its index. Indexes are integer values that represent the position of an item within a list. Indexes always start at zero (the same way a string index begins at zero). For example:

```
my_list = ["Apple", "Pear", "Peach"]
print (my_list[0])

# >> Apple
```

You will raise an exception if you attempt to access an element outside the range of a list. For example:

```
my_list = ["Apple", "Pear", "Peach"]
print (my_list[4]) # Index doesn't exist!
```

Lists are "mutable" data types, which means that they can be changed once they have been created (unlike strings). If you know the index of an item you wish to change, you can simply use the assignment operator to update the value of the item at that position in the list. For example:

```
my_list = [1, 2, 3]
print (my_list)
# >> [1,2,3]
my_list[0] = 99
print (my_list)
# >> [99,2,3]
```

Sample program: This program demonstrates list creation, repetition, concatenation and accessing individual elements within a list.

```
# create a new list with 5 elements
my_list = [95, 3, 12, 17, -4]
print ("List contents:", my_list)

# print just the first element of the list
print ("Element #0:", my_list[0])
```

```
# create a second list
second_list = [100, 101]
print ("Second list:", second_list)

# repeat this list 2x
second_list = second_list * 2
print ("Second list repeated 2x:", second_list)

# add the two lists together (concatenation)
big_list = my_list + second_list
print ("Big list:", big_list)
```

Output appears below:

```
List contents: [95, 3, 12, 17, -4]
Element #0: 95
Second list: [100, 101]
Second list repeated 2x: [100, 101, 100, 101]
Big list: [95, 3, 12, 17, -4, 100, 101, 100, 101]
```

Iterating over a List

When working with lists you will often need to access many or all of the elements within a listto solve a certain problem. For example, imagine that you had the following list of price values:

```
prices = [1.10, 0.99, 5.75]
```

If you wanted to compute 7% sales tax on each price you would need to access each item in the list and multiply it by 0.07. For a list with three elements this is pretty easy:

```
tax_0 = prices[0] * 0.07
tax_1 = prices[1] * 0.07
tax_2 = prices[2] * 0.07
```

However, as your lists become larger and larger this technique will become unmanagable (imagine you had 1,000 prices in the list!) -- the solution to this problem is to use a repetition structure to iterate over the contents of the list.

The simplest way to iterate over a list is to use a for loop to iterate over all items in thelist. When you do this, the target variable of your loop assumes each value of each element of the list in order.

Sample Program: The program below demonstrates how to quickly iterate over alist using a for loop.

```
prices = [1.10, 0.99, 5.75]
for item_price in prices:
        print ("Original Price:", item_price, "; tax due:", item_price * 0.07)
```

Output appears below:

```
Original Price: 1.1 ; tax due: 0.077
Original Price: 0.99 ; tax due: 0.0693
Original Price: 5.75 ; tax due: 0.4025
```

As you can see, a for loop is a convenient way to sequentially iterate through a list. The target variable in a for loop assumes the value of the current item in the list as you iterate. However, the target variable isn't very helpful if you want to change the value of an item in alist since it just represents a copy of the data. For example:

```
prices = [1.10, 0.99, 5.75]
for item_price in prices:
        item_price *= 1.07
print (prices) # prices remains unchanged!
```

Output appears below:

```
[1.1, 0.99, 5.75]
```

In order to change a list item you need to know the index of the item you wish to change. You can then use that index value to change an item at a given position in the list. For example:

```
prices = [1.10, 0.99, 5.75]
prices[0] *= 1.07
prices[1] *= 1.07
```

```
prices[2] *= 1.07

print (prices)
```

Output appears below:

```
[1.177, 1.0593, 6.1525]
```

There are two main techniques for iterating over the index values of a `list`:

- Setting up a counter variable outside the `list` and continually updating the variable as you move to the next position in the `list`
- Using the `range` function to create a custom range that represents the size of your `list`

If you set up an accumulator variable outside of your loop, you can use it to keep track of where you are in a `list`. For example:

```
prices = [1.10, 0.99, 5.75]

# set up counter variable
position = 0

# enter into a loop
while position < 3:
        prices[position] *= 1.07
        position += 1

# output
print (prices)
```

Output appears below:

```
[1.177, 1.0593, 6.1525]
```

To improve upon this example, we can use the len function to determine the size of our list(rather than just hard coding our loop to end after 3 iterations). The len function can take a list as an argument and will return the integer value of the size of the list. Example:

```
prices = [1.10, 0.99, 5.75]

# set up counter variable
position = 0

# enter into a loop
while position < len(prices):
        prices[position] *= 1.07
        position += 1

# output
print (prices)
```

Output appears below:

```
[1.177, 1.0593, 6.1525]
```

You can also use the range function to construct a custom range that represents all of the indexes in a list. This technique can be a bit cleaner to implement since you don't need to worry about setting up and maintaining a counter variable:

```
prices = [1.10, 0.99, 5.75]

# use the range() function to generate the range
# of valid index positions in this list
for i in range(0, len(prices)):
        prices[i] *= 1.07

```

```
# output
print (prices)
```

Output appears below:

```
[1.177, 1.0593, 6.1525]
```

Creating Lists

You can create an empty list with no elements using the following syntax:

```
mylist = []
```

With an empty list, you cannot place content into the list using index notation since there are no "slots" available to be used in the list. You can, however, append values to the list using the concatenation operator, like this:

```
mylist = []
mylist += ["hello"]
mylist += ["world"]
print (mylist)

# >> ["hello","world"]
```

Since you cannot access an element outside of the range of a list it is sometimes necessary to set up a correctly sized list before you begin working with it. For example:

```
# create a list of 7 zeros
daily_sales = [0] * 7
```

Slicing Lists

Sometimes you need to extract multiple items from a list. Python contains some built in functions that make it easy for you to “slice” out a portion of a

list. The syntax for list slicing is identical to the syntax for string slicing. To slice a list you use a series of "slice indexes" to tell Python which elements you want to extract. Example:

```
new_list = old_list[start:end]
```

Python will copy out the elements from the list on the right side of the assignment operator based on the start and end indexes provided. It will then return the result set as a new list. Note that slice indexes work just like the range function – you will grab items up until the end index, but you will not grab the end index itself. Here's an example:

```
list_1 = ['zero', 'one', 'two', 'three', 'four', 'five']
list_2 = list_1[1:3]
print (list_1)
print (list_2)

# >> ['zero', 'one', 'two', 'three', 'four', 'five']
# >> ['one', 'two']
```

If you omit the start_index in a slice operation, Python will start at the first element of the list. If you omit the end_index in a slice operation, Python will go until the last element of the list. If you supply a third index, Python will assume you want to use a step value. This works the same as the step value you would pass to the range function

Finding items in a list

You can easily test to see if a particular item is in a list by using the in operator. Here's an example:

```
my_list = ['pie', 'cake', 'pizza']

if 'cake' in my_list:
        print ("I found cake!")
```

```
else:
        print ("No cake found.")
```

The in operator lets you search for any item in a list. It will return a Boolean value that indicates whether the item exists somewhere in the list.

Appending Items to a List

You have already seen a few ways in which you can add items to lists:

- Repeat the list using the * operator
- Concatenate the list using the + operator

Another way to add items to a list is to use the append method.
The append method is a function that is built into the list datatype. It allows you to add items to the end of a list. Example:

```
mylist = ['Christine', 'Jasmine', 'Renee']
mylist.append('Kate')
print (mylist)
```

Removing Items from a List

You can remove an item from a list by using the remove method. Here's an example:

```
prices = [3.99, 2.99, 1.99]
prices.remove(2.99)
print (prices)
```

Note that you will raise an exception if you try and remove an item that is not in the list. In order to avoid this, you should make sure to test to see if it is in the list first using the inoperator, or use a try / except block to catch any errors you might raise.

Sometimes you want to delete an item from a particular position in a list. You can do this using the del keyword. For example, say you wanted to delete the item in position #0 in the following list:

```
my_list = ['pie', 'cake', 'pizza']
print (my_list)

# remove the item at position 0
del my_list[0]

print (my_list)
```

Output appears below:

```
['pie', 'cake', 'pizza']
['cake', 'pizza']
```

Re-ordering List Items

You can have Python sort items in a list using the sort method. Here's an example:

```
my_list = ['pie', 'cake', 'pizza']
my_list.append('apple')
print (my_list)

# sort the list
my_list.sort()

# the list is now in ascending order
print (my_list)
```

Output appears below:

```
['pie', 'cake', 'pizza', 'apple']
['apple', 'cake', 'pie', 'pizza']
```

Python can also reverse the items in a list using the reverse method. This method will swap the elements in the list from beginning to end (i.e. [1,2,3] becomes [3,2,1]) - note that this method does not sort the list at all. It simply reverses the values in the list. Here's an example:

```
my_list = [9,3,11]
my_list.reverse()

print (my_list)
```

Output appears below:

```
[11, 3, 9]
```

Getting the Location of an Item in a List

You can use the index method to ask Python to tell you the index of where an item can be found in a list. The index method takes one argument – an element to search for – and returns an integer value of where that element can be found. Caution: The index method will throw an exception if it cannot find the item in the list. Here's an example:

```
my_list = ['pizza', 'pie', 'cake']
if 'pie' in my_list:
        location = my_list.index('pie')
        print ('pie is at position #', location)
else:
        print ('not found!')
```

Output appears below:

```
pie is at position # 1
```

Getting the Largest and Smallest Items in a List

Python has two built-in functions that let you get the highest and lowest values in a list. They are called min and max – here's an example:

```
prices = [3.99, 2.99, 1.99]
biggest = max(prices)
smallest = min(prices)

print (smallest, 'up to', biggest)
```

Output appears below:

```
1.99 up to 3.99
```

Module #09 (Exceptions, Input/Output)

In this module we'll talk about how to prevent our programs from crashing when exceptional situations occur. We'll also learn about getting input from a device other than the terminal such as a local file stored on the computer or a file stored on a website.

Basic Exception Handling

An "exception" can be described as an error condition that causes a program to halt while it is running. In IDLE it is the angry-looking red message that appears when you encounter a runtime error. Here's an example of some code that would cause an exception to occur in your program (you can't convert the string "Craig" to an integer):

```
x = "Craig"
y = int(x)
```

When an exception occurs, Python will generate a "traceback." Tracebacks give information about the exception that occurred, including the line number that caused the issue and the name of the exception. When this happens, programmers generally say that an exception has been "raised" by the program. Many times you can avoid exceptions all together by simply adjusting how we code our algorithms. For example, the following code will raise an exception if x = 0:

```
x = int(input("give me a number"))
print (100/x)
```

Yet we can avoid the issue by wrapping the potentially dangerous code inside a selection statement. For example:

```
x = int(input('give me a number'))
if x != 0:
        print (100/x)
```

Python has an "exception handling" statement that can be used to "catch" exceptions and prevent them from crashing your program. Here's what it looks like:

```
try:
```

```
        # questionable code goes here
except:
        # this block will run if the code in the
        # 'try' block above raised an exception
else:
        # this block will run if the code in the
        # 'try' block above was successful
```

This should look a bit like an `if / elif / else` statement. When Python encounters the `try` statement it attempts to run the commands inside of the `try` block. If any of those statements cause an exception to be raised it will immediately stop executing the commands inside of the `try` block and jump down to the `except` block. Any commands listed in this block will be executed at this point. If Python does not encounter any exceptions in the `try` block then it will jump down and execute the statements in the `else` block. Note that the `else` block is optional - you can leave it off if you find that you don't need it in your program.

We often use `try` blocks to attempt "risky" tasks, such as converting strings into numeric data types or opening up files from the Internet. For example, here's a program that uses the `try / except / else` suite of statements to validate the data type of inputted data. Notice how you can enter ANYTHING as input and the program won't crash!

Sample program: This program demonstrates how the `try / except / else` blocks work in Python

---

```
# attempt to get a number from the user - we know that we can't count on
the user
# to do anything right, so we have to be prepared!
try:
        num = int(input("Give me a number: "))

# if there was an error in the try block above then we can respond to it using
the except block below
except:
        print ("That didn't work. What did you do?")

# if the try block succeeded, we can do this
# note that the else block is optional - you can omit it if necessary
else:
```

```
            print ("Thanks!  That worked.")
```

Splitting Strings into Lists

Sometimes you are given access to a single string that contains multiple smaller values. For example, consider the following string:

```
student = "Tracy,NYU,Sophomore"
```

This string contains three pieces of data – a name, a school and a class year. The data has been "packaged" into a single string in a meaningful way (i.e. we know that the comma character separates the different data points)

We can use a technique called "splitting" to"unpack" a string and extract its individual values. The trick is to isolate a "separator" character that can be used to delimit the data you want to extract. Once you have identified the separator you can use the split method to cut apart the string into a list of values. Here's an example:

```
student = "Tracy,NYU,Sophomore"

# cut apart the string based on the position of the commas
splitdata = student.split(",")

# we now have a list with 3 elements (the commas have been removed)
print (splitdata)

# we can use index notation to access the individual elements of our data
print ("Student name:", splitdata[0])
print ("School:", splitdata[1])
print ("Class year:", splitdata[2])
```

Output:

```
Student name: Tracy
School: NYU
```

Reading Data from the Web

You generally won't be "hard coding" data into your programs as in the examples above. Instead, you will usually be processing information that comes to you in the form of user input or via an external data source.

One way to easily get information into Python from the "outside world" is to initiate a connection to the Web and obtain data via a URL (Universal Resource Locator, such as "http://www.google.com")

The general algorithm for reading data from a web-based source is as follows:

1. Identify a source of data on the web that you wish to work with. Copy the URL that can be used to access that data source via a web browser.
2. Create a new Python program that imports the urllib.request module.
3. Initiate a web connection to an external data source using the urllib.request module.
4. If the request is successful, obtain the data that exists at the URL. The data will be sent to your program in the form of single large string.
5. "Clean up" the string into a meaningful form that your program can understand. This usually involves making sure that the character set used in the external file is one that your computer can handle (i.e. make sure that the data is being supplied as a set of ASCII characters).
6. Parse your string into a list using the split method discussed above.
7. Process this list and generate output.

Here is a sample bit of code that reads data from the Google website and displays it using Python.

```
import urllib.request

# where should we obtain data from?
url = "http://www.google.com"

# initiate request to URL
response = urllib.request.urlopen(url)

# read data from URL as a string, making sure
# that the string is formatted as a series of ASCII
charactersdata = response.read().decode('utf-8')

# output!
print (data)
```

Keep in mind that the web is a volatile place – servers can go up and down at a moment's notice, and your connection to the network may not always be available. You need to be prepared to deal with connection errors in your programs. You can use the try / except / else suite to help “catch” errors before they crash your program. For example:

```
import urllib.request

# where should we obtain data from?
url = "http://www.google.com"

# attempt to access www.nyu.edu
try:
        # initiate request to URL
        response = urllib.request.urlopen(url)
```

```
        # read data from URL as a string, making sure
        # that the string is formatted as a series of ASCII characters

        data = response.read().decode('utf-8')
        # output!
        print (data)
# an error occurred!  handle it here
except:
        print ("Something went wrong!")
```

Programs and Memory

All of the programs that you have been writing so far have reset themselves each time they run. This is because the data associated with the program (i.e. your variables) are stored in your computer's memory (RAM). RAM is a volatile place – it serves as a computer's short-term memory, and any RAM that is being used by a program is cleared when the program stops running. If you want your programs to retain data between executions then you need to find a "permanent" way to store and save information for future use.

A computer almost always has at least one type of long-term storage device at its disposal (usually some kind of hard drive). We can use the long-term storage capabilities of a computer to store data in the form of a file. Once we save a file it will remain on the long-term storage device after the program is finished running, and can be accessed and retrieved later on by the same program or by a different program. This is a very common technique that is used by almost all programs that need to keep track of some kind of information between executions. For example, when you click the "save" button in a word processing program you are storing the data you typed onto a long-term storage device.

There are two main families of files that we work with when writing programs - text files and binary files. Text files are files that contain data that is encoded as text (i.e. ASCII or Unicode characters) - data stored in a

text file is visible and can be read by a program that is designed to view / edit textual data (i.e. a word processing program). We will be working with text files exclusively in this class.

Binary files contain data that is not encoded as a text format and thus are not "human readable". Binary files are intended to be read by other programs and not by humans directly and appear "garbled" when viewed via a word processing program.

Filenames and Working with File Objects

Every file on your hard drive must have a filename. On most operating systems a filename comes with a “file extension” which allows you to quickly tell what kind of data is stored inside that file. Extensions come after the last period in a file (i.e. “mydocument.doc” – “doc” is the file extension). File extensions tell the operating system and the user what kind of data is stored in the file and allows the OS to select an appropriate program to open a particular file. We will be working with text documents exclusively in this class, so all of our files will end with the "txt" file extension.

Now that we know how files are stored we can begin to work with them inside of Python. In order to access a file via your programs you must first created a special variable called a "file object" - this variable serves as a connection between your program and the operating system of your computer. File objects also store the name of a file that you wish to work with along with what you want to do to that file (write, read, append, etc).

Filenames are expressed as strings in Python. When you ask Python to open a file (i.e. “myfile.txt”) you are telling Python to open up a file with that name in the current directory. You can ask Python to open up a file that exists outside of the current directory by typing in an absolute path (see your book for more information on how to open files outside of your current working directory)

You can create a file Object by using the `open` function with following syntax:

```
# create a file object that opens the file "myfile.txt"
# for write access ("w")
file_object = open("myfile.txt", "w")
```

Writing Data to a File

You can write data into a file once you have created a file object and have opened a file for writing using the write function. Here's an example:

```
# open a file for writing
file_object = open('myfile.txt', 'w')

# write two strings to the file
file_object.write("hello")
file_object.write("world")

# close the file when you're done
file_object.close()
```

Note that the program above will create a text file named "myfile.txt" in your current directory which will contain the following data:

```
helloworld
```

You will probably want to try and avoid writing files that concatenate all of your data into one long line of unintelligible text. This is bad practice since it can be very difficult (or impossible) to extract out your data later on. One way to separate data in a text file is by splitting out your data into multiple lines using the \n escape character. This allows you to store a single value on each line of your text file, which will make things a lot easier for you later on when you need to read your file. Here's an example of how the\n character can be used to separate values onto new lines in a file:

```
# open a file for writing
file_object = open('myfile.txt', 'w')

# write two strings to the file with linebreaks
# ending each string
file_object.write("hello\n")
file_object.write("world\n")
```

```
# close the file when you’re done
file_object.close()
```

Appending Data to a File

When you open a file for writing using the `w` flag you are asking Python to open the file so that you can place data into it. If the file does not exist Python will create it for you. If the file does exist Python will **overwrite** the current contents of the file. This isn't always what you want to happen! If you want to write data to a file but preserve the existing contents of a file you can open up the file in "append" mode. This mode allows you to "add on" additional data at the end of a file. Here's a sample program that shows this in action:

Critical: Make sure that if you open a file using the `w` flag, you are okay with the original data being erased! Many students have accidentally overwritten their homework assignments this way!

```
# open a file for appending
file_object = open("cartoon.txt", "a")

# ask the user for a cartoon character
character = input("Enter a cartoon character: ")

# write the character to the file
file_object.write(character)

# write a line break
file_object.write("\n")

# close the file
file_object.close()
```

Reading data from a file

You can read data contained inside a file once you have created a file object and have opened a file for reading using the `read` function. Here's an example:

```
# open a file for reading
myvar = open("test.txt", "r")

# read in all data as one long string
alldata = myvar.read()

# output
print (alldata)

# close the file
myvar.close()
```

The `read` function extracts all data from a file as a string and returns a string to your program. This one large string generally needs to be further processed before it can actually be used. Often we can use the `split` method to cut apart the string into usable chunks.

Writing Numeric Data to a File

The `write()` function only accepts strings – the following code will generate a runtime error:

```
# open a file for writing
```

```
myvar = open("test2.txt", "w")

# write an integer to the file
myvar.write(55)

# close the file
myvar.close()
```

You need to convert any data that will be stored in a file into a string by using the `str` conversion function. Example:

```
# open a file for writing
myvar = open("test2.txt", "w")

# write an integer to the file in the form of a string
myvar.write(str(55))

# close the file
myvar.close()
```

Reading Numeric Data from a File

When you are reading data from a file you will often need to convert strings to a numeric data type if you want to perform calculations on the data. This process is the same as the process we have to go through when using the `input` function to read numeric data from the user.

Note: The `int` and `float` functions automatically remove line break characters for you!

Module #10

In this module you will learn about another data structure called dictionaries which can be used to store information using 'keys' rather than integer based indexes.

Dictionary Basics

A **Dictionary** is another helpful iterable data structure that is built-in to Python. In a natural language, a dictionary has a series of terms and each term has a definition. A Python dictionary isn't much different, but instead of "terms" we call them "keys" and instead of "definitions" we call them "values" or "data." In that sense **a dictionary is a set of mappings from specific keys to specific values**. The one difference between a natural language dictionary and a Python dictionary is that **a Python dictionary is unordered**. That means that unlike a list which has a set order, the order of items in a dictionary is undetermined.

Dictionaries can be incredibly useful if we want to store data and look that data up by name instead of using an index. Unlike the List sequence structure which uses an integer index to lookup values (and maintains the order of the list), a dictionary uses immutable objects (usually strings) called a **key** to find a specific value.

In other languages, dictionaries are sometimes called "hashmaps" or "associated arrays" since you can think of them as unordered arrays whose items are indexable using some associated data (the "key") instead of a number.

Creating a dictionary

You can create either empty dictionaries or dictionaries with items already in the dictionary. To create an empty dictionary we use the syntax `{ }` or `dict()`. To create a new populated dictionary we can use the form `{ KEY_1: VALUE_1, KEY_2: VALUE_2, ..., KEY_N: VALUE_N }`. Here are a few examples:

```
# Creating an empty dictionary
students = { }      # key - a student ID, value - name of the student
```

```
# Creating a dictionary with two entries (key-value pairs).
# In this case, the dictionary is of students, where the key is the
# student ID and the value is the name of the student
students = { "N12345678": "Andrew Case",
             "N87654321": "Bruce Wayne" }
```

Looking up items in a dictionary

You can access an element's value by simply using the syntax `DICTIONARY_NAME[KEY]`. So we can `print()` each student in our student dictionary defined above using:

```
print("Student: " + students["N12345678"])
print("Student: " + students["N87654321"])
```

Which results in the output:

```
Student: Andrew Case
Student: Bruce Wayne
```

Sample Program: In this sample program, we create a dictionary of technical words and allow the user to lookup the definitions for these words from the dictionary.

```
# Note: User must enter one of 'dict', 'list', 'map', or 'set'

# Creating a dictioanry of technical terms and basic definitions
#   key - the word to define
#   value - the definition of the word
tech_terms = { "dict": "stores a key/value pair",
               "list": "stores a value at each index",
               "map": "see 'dict'",
               "set": "stores unordered unique elements" }
```

```
# Ask the user for a term
user_term = input("What term do you want defined (e.g. 'dict')? ")

# Lookup the definition and print it to the user
definition = tech_terms[user_term]
print(user_term + " - " + definition)
```

Inserting/Replacing items in a dictionary

To insert or replace an item in the dictionary the form is `DICTIONARY_NAME[KEY] = VALUE`. For example:

```
students = dict()                       # Create an empty dictionary
students["N12345678"] = "Andrew Case"   # Add students to dictionary
students["N87654321"] = "Bruce Wayne"
students["N99999999"] = "Clark Kent"
```

If the item for that key doesn't exist, a new item will be created. If an item already exists for that key, the value will be overwritten with the new value.

Checking if an element exists

You can check if an element's value exists in a list using the form `KEY in DICTIONARY_NAME` like so:

```
if "N12345678" in students:
        print(students["N12345678"])
```

Output:

```
Andrew Case
```

Delete an item from a dictionary

Use the `del` keyword followed by the item to delete. The form is `del DICTIONARY_NAME[KEY]` like so:

```
del students["N12345678"]
```

Replace an item from a dictionary

Remember that the keys are immutable, so you can't change a key for an entry, but you can replace the value that is being referenced by a specific

key. For example, let's say that "Bruce Wayne" wanted to officially change his name to "Batman." We could do so using the same method that we used to insert the original value:

```
students["N99999999"] = "Superman"
```

Iterating over Dictionaries

To iterate through each key and use that to pull each value we use the `DICTIONARY_NAME.keys()` function to get a sequence of keys for that dictionary. Here's an example of how we can iterate over each key:

```
for key in students.keys():
        print(key)
```

Output:

```
N99999999
N87654321
```

To iterate through each value we use `DICTIONARY_NAME.values()` function to get a sequence of all values in that dictionary. Here's an example:

```
for value in students.values():
        print(value)
```

Output:

```
Superman
Bruce Wayne
```

To iterate through each key-value pair we use the `DICTIONARY_NAME.items()` function to get both the key and value of each item in that dictionary. Here's an example:

```
for key, value in students.items():
        print(key, "->", value)
```

Ouptut:

```
N99999999 -> Superman
N87654321 -> Bruce Wayne
```

Differences between data structures

	Lists	Dictionaries

Ordered?	ordered by index	unordered
Lookup item using:	numeric index	unique key
Duplicate values allowed:	✓	✓

When to use which data structure

To determine which data structure you should probably use, a basic approach is to follow these rules (there are times when these rules should be broken for performance reasons, but that is beyond the scope of this section):

- If order matters: **use a List**
- If you need to store/lookup values using a unique name/key: **use a Dictionary**

Sample Program: The following program repeatedly asks the user to enter a name. Once the user enters a blank name, it then prints the total number of times each name was entered.

```
''' Counts the number of times each name was entered '''

# Initialize empty data set
names = dict()

# Get names from a user
name = input("Enter a name (or a blank line to quit): ")
while name != "":
    if name in names:
        # increment number of times entered
        names[name] += 1
    else:
        # first time this name has been entered
        names[name] = 1
    # Get another name from the user
    name = input("Enter a name (or a blank line to quit): ")


# Print the results
for name, count in names.items():
```

```
    if (count > 1):
        print("The name '" + name + "' was entered " + str(count) + " times.")
    else:
        print("The name '" + name + "' was entered " + str(count) + " time.")
```

Dictionary Methods

Like lists, dictionaries are objects which have methods that can run on any dictionary that has been created. Below are some of the most commonly used methods. You can get a list of all the dictionary methods by running `help(dict)` in a Python shell, or you can visit the Python API for a complete list.

Getting the number of items

You can use the same built-in `len()` function that works with lists to also get the length of a dictionary. In this case the syntax is `len(DICTIONARY_NAME)` which returns the number of items in the dictionary provided. Here is an example:

```
students = { "N12345678": "Andrew Case",
                          "N87654321": "Bruce Wayne" }
print("Number of items:", len(students))
```

Output:

```
Number of items: 2
```

Getting a value

In addition to the syntax `DICTIONARY_NAME[KEY]` to get a value (which could error if the key doesn't exist), there is a method you can use to avoid getting a possible `KeyError`. `DICTIONARY_NAME.get(key[, default])` returns the value for `key` if `key` is in the dictionary, else `default`. If `default` is not given, it defaults to `None`. Here is an example:

```
students = { "N12345678": "Andrew Case",
                          "N87654321": "Bruce Wayne" }
print(students.get("N12345678"))
```

```
print(students.get("N11111111"))
```

Output:

```
Andrew Case None
```

Getting all keys

Sometimes you may want to lookup what keys are stored in the dictionary. To do that you'll use the `DICTIONARY_NAME.keys()` method. This method will return a list data structure where each item in the list is one of the keys from the dictionary. Here's an example:

```
students = { "N12345678": "Andrew Case",
             "N87654321": "Bruce Wayne" }

# Get all the studentIDs (the keys)
student_IDs = students.keys()

# Print all the studentIDs
for student_ID in student_IDs:
    print("ID:", student_ID)
```

Output:

```
ID: N87654321
ID: N12345678
```

> **Note**: Since a dictionary is unordred, the order of the returned items is arbitrary.

Getting all values

Other times, you may just want to get a list of all the values stored in a dictionary. To do that you'll use the `DICTIONARY_NAME.values()` method:

```
students = { "N12345678": "Andrew Case",
             "N87654321": "Bruce Wayne" }

# Get all student names (values in the dictionary)
```

```
student_names = students.values()

# Print all the students one at a time
for student in student_names:
        print("Student:", student)
```

Output:

```
Student: Bruce Wayne Student: Andrew Case
```

Removing a specific item

Often we may want to remove an item from the dictionary to do some sort of processing on it. Instead of getting the item and then deleting it, we can do so with the pop() method.`DICTIONARY_NAME.pop(key[, default])` returns the value for `key` and removes the item if it exists, otherwise `default` is returned. If `default` is not specified and `key` is not in the dictionary, a`KeyError` is raised. Here is an example:

```
students = { "N12345678": "Andrew Case",
             "N87654321": "Bruce Wayne" }

# Remove myself and print the results
student = students.pop("N12345678")
print("Student =", student)

# Print remaining students in the dictionary
print("Students =", students)
```

Output:

```
Student = Andrew Case
```

```
Students = {'N87654321': 'Bruce Wayne'}
```

Removing an unspecified item

Often we may not care which item to pull out of a dictionary. We can do this using the DICTIONARY_NAME.popitem() method which removes and returns an arbitrary (key, value) pair from the dictionary. This can be useful if you want to remove and process items at the same time. If the dictionary is empty, calling popitem() raises a KeyError. Here's an example:

```
students = { "N12345678": "Andrew Case",
                    "N87654321": "Bruce Wayne" }

# Remove and print one item
student = students.popitem()
print(student)

# Remove and print another item
student_id, student_name = students.popitem()
print("Student ID:", student_id)
print("Student:", student_name)
```

Output:

```
('N87654321', 'Bruce Wayne')
Student ID: N12345678
Student: Andrew Case
```