A) Creating a simple Python Program in the Wing IDE (Integrated Development Environment)

B) Dissecting the Simple Program

Code
01 # Introduce students to CS202 02 print("Welcome to CS202") 03 print("In this course we will use Python to") 04 print("solve problems and perform tasks.") 05 print("Hope you enjoy the course!")
Output
Welcome to CS202 In this course we will use Python to solve problems and perform tasks. Hope you enjoy the course!

What does the # represent?

Everything from the # through the end of the line is ignored by the compiler.

The purpose is to indicate the purpose of the code to any programmer reading your code. Assume the programmer is Python knowledgeable. For example, they know what an if statement does.

There is no multiline comment in Python like in other languages. How can one achieve a similar effect?

a) Place a _____ at the beginning of each line; or

b) Use a docstring – enclose the desired comment with a pair of _____ or _____.

Examples of a) and b):

"Welcome to CS202" is an example of a string literal. In Python, a string literal is a sequence of characters surrounded by _____ or _____ quotes.

print is an example of a _____ : A collection of instructions that perform a particular task.

Analyze the following code and output

Code	Code	Code
01 print("JCCC") 02 print() 03 print("Cavaliers")	print(2) print(2 + 3)	print("2 + 3 =", 2 + 3, "!")
Output	**Output**	**Output**
JCCC Cavaliers	2 5	2 + 3 = 5 !

By default, print always inserts a trailing end of line marker into the output stream.

What was the result of the inserted commas in the right-most example?

Note: We will learn about f-strings soon. The right-most example above is better written as:

```
print(f"2 + 3 = {2 + 3}!")
```

Basic print statement syntax	Semantics
print()	Insert an end of line marker into standard output
print(arg1, arg2, ..., argn)	Send each argument to standard output with space delimiters. An end of line marker is inserted after the last argument.
print(arg1, ..., argn, end = val)	Send each argument to standard output with space delimiters. The value of **val** is inserted after the last argument.

C) Producing a Python Program

1) First, let's understand the Python process

Pressing the green arrow in Wing begins this process

Source code

```
score1 = 85
score2 = 90
totalScore = score1 + score2
...
```

↓

Compiler

↓

Byte Code*

```
0 LOAD_CONST               1 (85)
2 STORE_FAST               0 (score1)

4 LOAD_CONST               2 (90)
6 STORE_FAST               1 (score2)

 8 LOAD_FAST               0 (score1)
10 LOAD_FAST               1 (score2)
12 BINARY_ADD
14 STORE_FAST              2 (totalScore)
```

↓

Virtual Machine (Interpreter)

(The byte code is executed resulting in totalScore containing 175)

Source code: Instructions in a high-level language (e.g., Python, C++, Java) that need to be translated before being executed on a computer.

Compiler: A program that converts programming code in one language into another language.

Byte Code: Intermediate instructions executed by a virtual machine.

(*Note: The bytecode on the left was dissembled with the **dis** module to make it readable)

Virtual machine: A program that simulates a CPU to execute byte code.

2) Syntax Errors (also known as Compile-Time Errors)

a) **Syntax Error:** An instruction that does not follow programming language rules and is rejected by the compiler.
- A syntax error prevents the source code from being fully compiled (translated) into byte code.

b) Examples

Example 1	Why the syntax error?

Debug I/O | Python Shell

Commands execute without debug. Use arrow keys for history.

```
>>> Print("JCCC")
Traceback (most recent call last):
  Python Shell, prompt 12, line 1
builtins.NameError: name 'Print' is not defined
>>>
```

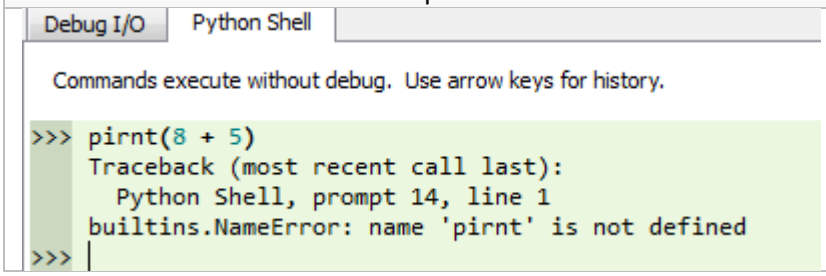
Example 2	Why the syntax error?

Debug I/O | Python Shell

Commands execute without debug. Use arrow keys for history.

```
>>> print("JCCC)
Traceback (most recent call last):
  Python Shell, prompt 13, line 1
Syntax Error: print("JCCC): <string>, line 1, pos 13
>>>
```

Example 3	Why the syntax error?

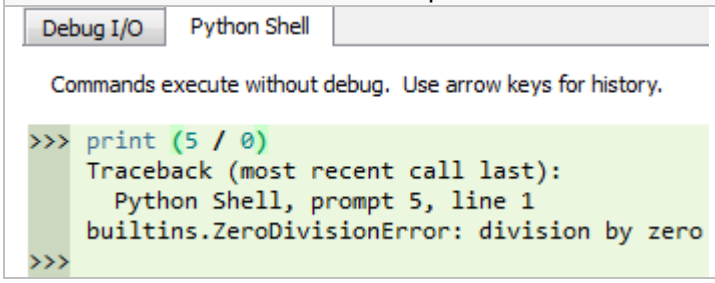


3) Logic Errors

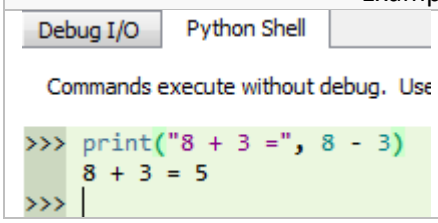
a) **Logic Error (or Run-time Error):** An error in a syntactically correct program that causes incorrect results.
- The code compiles but does not run correctly.

b) Examples

Example 1	Why the logic error?



Example 2	Why the logic error?



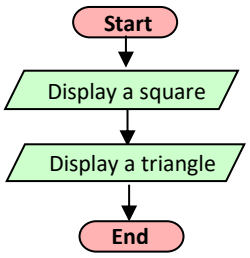
D) Additional Basic Output and Input

1) Program 1: Basic output with added documentation.

Pseudocode

Display a square
Display a triangle

Flowchart



Shapes.py

```
01  # This program displays shapes using asterisks.
02  # @author Rachel Jackson
03
04  # Display a square
05  print("*****")
06  print("*   *")
07  print("*****")
08
09  # Display a triangle
10  print()
11  print("    *  ")
12  print("  *   *")
13  print("*********")
```

Coding Style: In this course, file header documentation includes a professional description of the program and @author followed by the author's first and last name.

Coding Style: Each of these comments describes the purpose of a "paragraph" (or chunk) of related code. They often begin with an action verb. **Pseudocode can be turned into program comments**.

Output

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

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

Questions:

- What did print() on line 10 produce?
- Do blank lines in the source code (lines 03 and 04) result in blank lines being output?

Note: Comments/Documentation that span multiple lines (multi-line comments) are sometimes easier written with a docstring (documentation string) – The comment is delimited with triple quotes.

```
01  """This program displays shapes using asterisks.
02     @author Rachel Jackson"""
```

or

```
01  '''This program displays shapes using asterisks.
02     @author Rachel Jackson'''
```

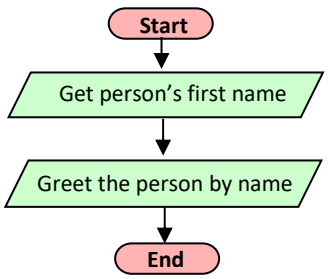
A coding style guide is used for this course that prepares students for future JCCC programming courses. If follows more of Javadoc style that differs somewhat from Python recommendations. For example, documentation and commenting do differ in format (but not necessarily in substance). You may need to adjust if strictly coding in Python for a company.

2) Program 2: Basic Input

Pseudocode

Get person's name
Greet the person

Flowchart



GreetName.py

```
01  # Get person's first name
02  name = input("First name: ")
03
04  # Greet the person
05  print("Welcome to beginning programming,", name)
```

FYI: To conserve space in the notes, not all programs will show file header documentation.

Run #1 Output (User input shown in **red**)

```
First name: Andrew
Welcome to beginning programming, Andrew
```

Run #2 Output (User input shown in **red**)

```
First name: Jessica
Welcome to beginning programming, Jessica
```

Questions:

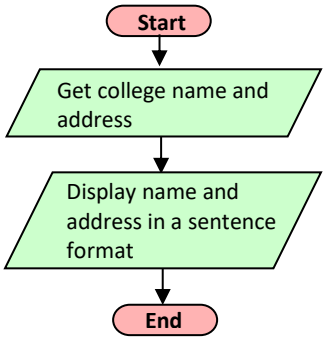
- What is the name of the instruction (function) that retrieves data from the user?
- What is the purpose of its string argument?
- What type of information does it return?
- What is the purpose of **name** followed by **=** ?

3) Program 3: Basic Input

Pseudocode

Get college name and address
Display name and adress in a sentence format

Flowchart



College.py

```
01 """ This program retrieves and displays a college name
02 and address.
03 @author Rachel Jackson """
04
05 # Get college name and address
06 collegeName = input("College's name: ")
07 collegeAddress = input("College's address: ")
08
09 # Display name and address in a sentence format
10 print()
11 print("The address of", collegeName, "is", collegeAddress)
```

Output (User input shown in red)

```
College's name: JCCC
College's address: 12345 College Blvd

The address of JCCC is 12345 College Blvd
```

Questions:

- Review: What did the print() statement on line 10 produce?
- Review: How many arguments were sent to print on line 11? What was output between each argument?
- What is output instead of collegeName and collegeAddress?
- What happens to the output for line 11 if quotes are placed around collegeName and around collegeAddress?
- What happens to the output for line 11 if multiple consecutive spaces are placed immediately after or before the comma separators (not inside quotes)?

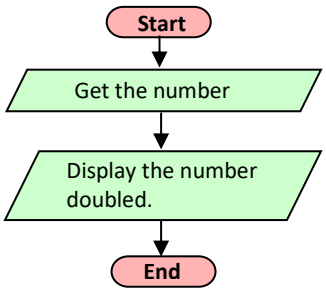
Observation: To learn a programming language, ask yourself what if questions and adjust/run the code to answer your questions. Remember, the Python interactive interpreter (Python shell window in the Wing IDE) can help with this too.

4) Program 4: Basic Input (numeric input to be used in a calculation)

Pseudocode

Get the number
Display the number doubled

Flowchart



Double.py (file header comments not shown)

```
01  # Get the number
02  print("Enter a number and I will double it!")
03  numToDouble = int(input("Your number? "))
04
05  # Display the number doubled
06  print()
07  print(numToDouble, "doubled is", numToDouble * 2)
```

Output (User input shown in red)

```
Enter a number and I will double it!
Your number? 10

10 doubled is 20
```

Questions:

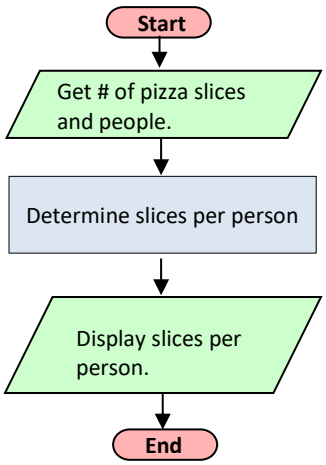
- What function was added to convert the user-entered string into an integer that could be used in a mathematical calculation?
- What symbol performed the multiplication?
- What are examples of numeric input that would **not** need conversion to an int (i.e., not used in a mathematical calculation)?

5) Program 5: Basic Input (Getting numeric input to be used in a calculation)

Pseudocode

Get # of slices and people
Determines slices per person
Display slices per person

Flowchart



PizzaSlices.py

```
01  """ This program determines how many slices of pizza each
02  person in a group may eat.
03  @author Rachel Jackson """
04
05  # Get the number of pizza slices and people
06  numSlices = int(input("How many slices of pizza? "))
07  numPeople = int(input("How many people are eating? "))
08
09  # Determine the number of slices for each person.
10  slicesPerPerson = numSlices / numPeople
11
12  # Display the number of slices per person
13  print()
14  print("Number of slices per person:", slicesPerPerson)
```

Output (User input shown in red)

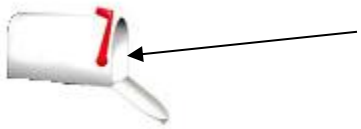
```
How many slices of pizza? 30
How many people are eating? 12

Number of slices per person: 2.5
```

A) Variable Definitions

1) Introduction

Variable: A memory (storage) location whose contents may _____ during program execution.

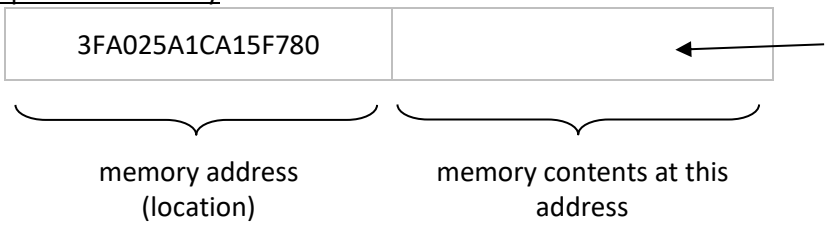
Depicted analogously



The mailbox is the location
The "letters" put into the mailbox are the contents

(image modified from: http://www.fotosearch.com/clip-art/mailbox.html)

Depicted technically



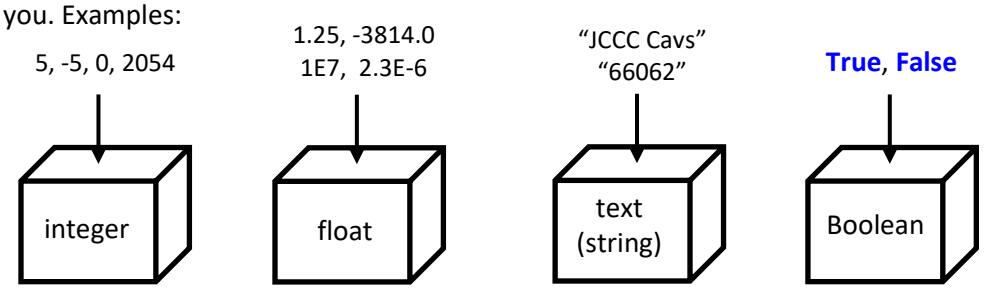
Rather than remembering a numeric address, programmers use **identifiers** to refer to this location.

Identifiers: _____ for variables and other programmer-defined items such as functions, classes, and methods.

2) Two key pieces needed to create a variable

a) Data type

Determine what type of value is to be stored into a variable (memory location). Python does this for you. Examples:



Numeric
- **Integer:** A number that does not contain a decimal point (no fractional part)
- **Floating-point number**: A number that contains a decimal point (has a fractional part)

String: A sequence of characters.
(A string literal includes beginning and ending delimiters such as double quotes.)

Boolean: A value that is either true or false.

Notes:

- Complex is also a numeric type.
- There are other data types (e.g., list) discussed later.
- There are no thousands separators in numeric literals (~~10,734~~ → 10734)
- Even if a number ends with .0 (e.g., -10.0, 23.0), the number is still considered a floating-point number. The fractional part is simply 0.
- Boolean values are not surrounded with quotes. (~~"True"~~ → True)

b) **Identifiers**

Naming rules for a Python identifier:

- May be a combination of letters, digits, or underscores.
- The first character may not be a digit.

Are the following valid (V) or invalid (I) identifiers?

7students	numStudents	num_students
_students	num-students	school.students

Note 1: Identifiers are _____

The following are **different** identifiers: **students, Students, STUdenTS**

Note 2: An identifier may not be a reserved word

Reserved word (or **keyword**): A word that has special meaning to the programming language and may not be used as a name by the programmer.

Examples:

However, a reserved word could be validly used as part of an identifier

Examples:

c) Proper Coding Style

Coding Style:

- Names for variables are **lowerCamelCase**.
- Identifiers must be descriptive of the data stored or action performed.

Examples of lowerCamelCase (The beginning of each word other than the first is a capital letter. No underscores are used between words.)

Descriptive Identifier	Not Descriptive
radius	r
numAnimals, numberOfAnimals	na, numA, num

d) Putting this together – Defining (creating) a proper variable

Define a variable holding the number of students in a course by initializing to 0

Define a variable holding the total monthly rainfall by initializing to 10.6

Define a variable holding someone's first name by initializing to your first name

Define a variable holding someone's middle initial by initializing to the empty string

e) Likely a good idea to avoid the following:

```
limit = 100
.
.
.
limit = "one hundred"
```

Language Observation: Languages such as C++, C#, and Java that implement "static typing" do not allow the same variable to first hold a number and later a different type such as a text string.

Language Observation: Variable declarations/definitions in languages such as C++, C#, Java require the programmer to specify the data type and do not necessarily need to be initialized.

```
int numAnimals;
int numAnimals = 0;
```

f) **Named Constant**: A name given for a stored value that is not changed by a program. (The stored value is known before the program runs and does not change while the program runs)

Note: Python does not implement named constants, but we can signify that variables are to be treated as constants by using proper coding style.

Coding Style:
- Named constants are UPPER_CASE with underscore separators. (e.g., HOURS_IN_WEEK = 24)
- Named constants are used in place of numeric literals. **Exceptions**: {0,1,2}; output formatting; numeric literals in formulas.

Literal: A fixed value entered into source code.
Examples of numeric literals: 10, 100.5, -100
Examples of string literals: "JCCC", "Kansas City"

Magic Number: A numeric literal whose meaning is not apparent.
Named constants are to replace magic numbers to make the code more readable.

Named constant example

Code

```
01 SALES_TAX = .081
02
03 # Get the football cost
04 footballCost = float(input("Football cost? $"))
05
06 # Calculate and display total cost
07 totalCost = footballCost + SALES_TAX * footballCost
08 print("Total cost: $", totalCost)
```

Output (user input in **red**)

```
Football cost? $19.99
Total cost: $ 21.609189999999998
```

Note: Named constants may also simplify code maintenance. If the sales tax changed at some point in the future, and if SALES_TAX were sprinkled in various places throughout a longer piece of code, code modification would only be required where SALES_TAX is first defined.

Coding Style: Insert one space on each side of **relational** (<, >, <=, >=, !=, ==), **logical** (and, or), and **assignment** (=) operators.
Proper: `totalCost = footballCost + SALES_TAX * footballCost`
Improper: `totalCost=footballCost + SALES_TAX * footballCost`

Note: A variable such as totalCost above aids in program readability. However, if a variable is only used once, it could be that the variable is not needed -- if readability is still maintained:

Code

```
06 # Calculate and display total cost
07 print("Total cost: $", footballCost + SALES_TAX * footballCost)
```

g) FYI: Visualizing what happens when Python creates an "immutable" type like an integer or string:

```
01 x = 500
02 y = x
03 y = x + 200
```

A new object is created when a modification occurs (line 03).

Python immutable types: Numbers, Booleans, Strings, Bytes, Tuples

However, at this point, it is easier to just think of the variable directly containing the value when manually tracing ("desk-checking)" code, and this is what we'll will do until we create our own classes and objects later

x	500
y	~~500~~ 700

B) Basic Arithmetic Operations

1) The operators

Binary Operation	Operator	Example	Note
Exponents	**	5**2 → 25	
Addition	+	5 + 2 → 7	
Subtraction	-	5 – 2 → 3	
Multiplication	*	5 * 2 → 10	
Division	/	5 / 2 → 2.5	In Python 3.x, **all results from / produce a float**. Example: 6 / 2 → 3.0
Floor Division (Integer Division)	//	5 // 2 → 2	The nearest **integer** <= the number. Be careful when one operand is negative: -5 // 2 → -3
Remainder (modulus)	%	5 % 2 → 1	Think of 5 / 2 (or 5 // 2), but the result is the integer remainder.

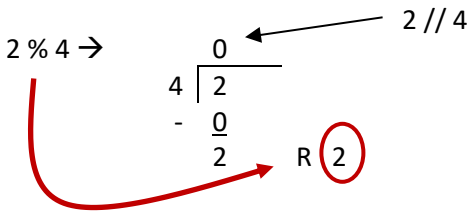
Language Observation: ** and // do not exist in languages such as C++, C#, and Java.

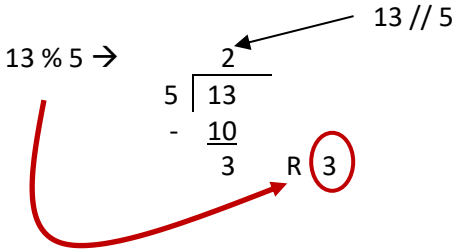
Language Observation: / results in floor division in languages such as C++, C#, and Java.

Binary operator: An operator with two operands (e.g., 5 + 2, 6 – 3)
Unary operator: An operator with one operand (e.g., -5)

Example with both a binary and unary operator:

Additional remainder (modulus) examples.





2) Operator precedence (order of operations)

Level	Operator	Evalauation order
1	()	
2	**	**right to left**
3	* / // %	**left to right**
4	+ -	**left to right**

3) Practice Examples (Some are pre-worked)

Note: Once there is a decimal point, keep it in the answer.

1 + 2.0 * 3
1 + 6.0
7.0

5 % 3 + 10 // 3

Reminder: / produces a number with a decimal point.

4 – 2 * 5 / 2

4 % 2 * 3 ** 2
4 % 2 * 9
0 * 9
0

2 + (3 * 4 / 8) – -2

4 – (3 + (1 + 5 // 2)) + 4
4 – (3 + (1 + 2)) + 4
4 – (3 + 3) + 4
4 – 6 + 4
-2 + 4
2

4) More practice evaluating expressions
Given the following declarations:
floatVar = 2.0
intVar = -4

Strategy: When evaluating by hand, place parentheses around the substituted number – especially if it is negative.

floatVar + intVar ** 3 intVar – floatVar / 2 intVar * int(floatVar)

5) Converting mathematical expressions into Python expressions:

Mathematical Expression	Converted into Python
$b^{10} - 3xy$	
$\frac{a + b}{c - d}$	

Coding Style (Optional): Except for the power operator (**), insert one space on each side of all binary, arithmetic operators.

6) What is the final value of x in the following examples?

Code		Code	Code
01	`x = 5`	`y = 2`	`y = 2`
02	`x = x - 8`	`z = 4`	`z = 4`
03		`x = y / z + y * z`	`x = y // z + y % z`
	x =	x =	x =

C) Augmented Assignment Operators (may also be called Compound Assignment Operators)

1) The operators

General Statement	Condensed Form
`variable = variable op expression`	`variable op= expression`

Assignment statement	Equivalent condensed assignment statement
`i = i + 1`	`i += 1`
`price = price - discount`	`price -= discount`
`result = result * scale`	`result *= scale`
`x = x % 2`	
`result = result / numScores`	

2) Code example

Code

```
01 # Get the integer
02 number = int(input("Enter an integer: "))
03
04 # Modify the integer
05 increase = int(input("Increase the integer by how much? "))
06 number += increase
07
08 multiplier = int(input("Now, multiply by how much? "))
09 number *= multiplier
10
11 # Display the final value
12 print("Final value:", number)
```

Output (user input in **red**)

```
Enter an integer: 8
Increase the integer by how much? 2
Now, multiply by how much? 3
Final value: 30
```

Question: Suppose x is variable that contains 2. What does **x** contain after the statement below executes? FYI: like the assignment operator, compound assignment operators are evaluated last (low operator precedence).

```
x *= 2 + 3 * 2
```

D) Math functions and the math module

1) Subset of built-in mathematical functions (math module not needed)

Function	Returns	Example
abs(x)	Absolute value of x	if x = -3, abs(x) → 3; if x = 3, abs(x) → 3
max(x_1, x_2, …,x_n)	The maximum of x_1, x_2, …,x_n	if a = 2, b = -4, c = 0, max(a, b, c) → 2
min(x_1, x_2, …,x_n)	The minimum of x_1, x_2, …,x_n	if a = 2, b = -4, c = 0, min(a, b, c) → -4
pow(base, exp)	base raised to the exp power	if base = -2, exp = 3, pow(base, exp) → -8
round(x) round(x, nDigits)	The integer nearest x The number rounded to nDigits precision	if x = 2.73, round(x) → 3 if x = 4.3143, round(x, 3) → 4.314

Warning: The round function rounds a floating-point number ending in .5 to the nearest even.

More Examples	Result
abs(-2)	
abs(16.5)	
Suppose x = -5.123, what is **round(x)** ?	
Suppose cost = 12.99, what is **round(cost)** ?	
round(11.5)	
round(12.5)	
round(-10.25619, 4)	
Suppose x = -1, y = 2, what is **max(x, y)**?	
What is min**(100, -100, 20, 0)**?	
What is **pow(3, 4)**?	

2) Other mathematical functions: The **math** module

a) Different ways to use information from the math module

from math import sqrt	The sqrt function from the math module is now available to be used in the program
from math import sqrt, pi	The sqrt function and the pi constant from the math module are now available to be used in the program
from math import *	Everything is available from the math module
import math	Everything is available from the math module but must be preceded by **math.** (e.g., **math.**sqrt(number))

b) Subset of math module functions

Function	Returns	Example
ceil(x)	x rounded up	if x = 2.2, ceil(x) → 3
sqrt(x)	The square root of x (x >= 0)	if x = 4, sqrt(x) → 2
log(x, base)	Logarithm of x to the given base Think of base$^{?}$ = x. The value of ? is returned	if x = 1000, base = 10 log(x, base) → 3

Note: A constant named **pi** also exists in the math module

c) Examples

Code

```
01 import math
02
03 print(math.sqrt(16))
04 print(math.pi)
```

Output

```
4.0
3.141592653589793
```

Code

```
01 from math import *
02
03 print(sqrt(16))
04 print(pi)
```

Output

```
4.0
3.141592653589793
```

Code

```
01 from math import ceil
02 testNum = 23.37
03 print(ceil(testNum))
04 # print(sqrt(testNum))
```

What if line 04 is uncommented?

Output

```
24
```

E) How to Avoid Code Line Wrap and/or Surpassing Maximum Line Length for Coding Style

1) Coding Style

Coding Style: Follow the 80-25 rule: long statements (> 80 characters) are clearly broken into separate lines. Note that a space is a character. Strive to limit functions and methods to 25 statements or less.

2) Strings may be split into smaller strings by using the concatenation (+) and line continuation (\) operators. The enter (or return) key must be pressed immediately after the \

Code (Line wrap and/or surpassing coding style line length)

```
01 gettysburgAddress = "Four score and seven years ago our fathers brought forth on
02 this continent, a new nation, conceived in Liberty, and dedicated to the
03 proposition that all men are created equal..."
04 print(gettysburgAddress)
```

Output

```
Four score and seven years ago our fathers brought forth on this continent, a new
nation, conceived in Liberty, and dedicated to the proposition that all men are
created equal...
```

Code (fixed)

```
01 gettysburgAddress = "Four score and seven years ago our fathers brought " + \
02     "forth on this continent, a new nation, conceived in Liberty, " + \
03     "and dedicated to the proposition that all men are created equal..."
04 print(gettysburgAddress)
```

FYI: Python permits concatenation of string literals without the +

```
01 gettysburgAddress = "Four score and seven years ago our fathers brought " \
02     "forth on this continent, a new nation, conceived in Liberty, " \
03     "and dedicated to the proposition that all men are created equal..."
04 print(gettysburgAddress)
```

Output

```
Four score and seven years ago our fathers brought forth on this continent, a new
nation, conceived in Liberty, and dedicated to the proposition that all men are
created equal...
```

3) Mathematical Expressions

a) Using the the line continuation character

Code (Line wrap and/or surpassing coding style line length)

```
01 temperature = 20
02 windSpeed = 15
03 windchill = 35.74 + 0.6215 * temperature - 35.75 * pow(windSpeed, 0.16)
04 + 0.4275 * temperature * pow(windSpeed, 0.16)
```

Code (fixed)

```
01 temperature = 20
02 windSpeed = 15
03 windchill = 35.74 + 0.6215 * temperature - 35.75 * \
04     pow(windSpeed, 0.16) + 0.4275 * temperature * \
05     pow(windSpeed, 0.16)
```

b) Using enclosing parentheses instead of the backslash – This also works for string literals.

Code (fixed)

```
01 temperature = 20
02 windSpeed = 15
03 windchill = (35.74 + 0.6215 * temperature - 35.75 *
04     pow(windSpeed, 0.16) + 0.4275 * temperature *
05     pow(windSpeed, 0.16))
```

4) FYI: Containers of delimited values

When Lists and Dictionaries are discussed later, the delimited values enclosed within brackets [ ] and braces { } may also be placed onto separate lines without the need for the line continuation character. This will also apply to function parameters, but the enclosing symbols are again parentheses.

F) Strings

1) Recall

- String: A sequence of characters
- String literal: A sequence of characters with surrounding quotes that is entered into source code
  `"I am a string literal"  '12345 College Blvd'  "~a#$%"  '29'`

2) Individual string characters may be accessed by using an index and square brackets.

1	2	3	4	5		C	o	l	l	e	g	e		B	l	v	d
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

`"This is a string"[0]` → `"This is a string"[8]` →

3) **Empty string**: A string of length 0.
4) Sample string operations and functions

Code

```
01 husband = "Jim"
02 wife = "Laurie"
03 addressNumber = 12345
04
05 print(husband + " & " + wife)
06 print(husband[0] + " & " + wife[0])
07 print(len(husband))
08 print(len("husband"))
09 print("-" + "Echo-" * 5)
10 print(str(addressNumber) + " College Blvd")
11 print("She said, 'Touchdown Chiefs!'")
12 print('She said, "Touchdown Chiefs!"')
```

Output

```
Jim & Laurie
J & L
3
7
-Echo-Echo-Echo-Echo-Echo-
12345 College Blvd
She said, 'Touchdown Chiefs!'
She said, "Touchdown Chiefs!"
```

Reminder: What did the **+** operator do in the context of strings?

What did **[0]** do when placed just after a string variable?

Given the preceding program, what would be **wife[3]**?

What does `len`**(string)** do?

Given the preceding program, what character is accessed with **wife[len(wife) - 1]**? (When we get to lists, we'll see that this is the same result as writing **wife[-1]**)

What did the * do for **"Echo-" * 5** ?

What if line 09 were instead changed to **`print(addressNumber + " College Blvd")`** ?

Notice in lines 11 and 12 how differing ending string delimiters allows single-quotes or double-quotes to be easily embedded into strings.

The following wouldn't work: **`print("She said, "Touchdown Chiefs!"")`**

5) Formatted Strings using f-Strings

a) Improved output

Code (with no formatting)

```
01 SALES_TAX_RATE = .081
02
03 # Get the item and the price
04 itemName = input("Enter the item's name: ")
05 itemCost = float( input("Enter the " + itemName + "'s price $") )
06
07 # Calculate and display the total cost
08 totalCost = itemCost + itemCost * SALES_TAX_RATE
09 print("Total Cost: $" + str(totalCost))
```

Output (user input shown in red)

```
Enter the item's name: baseball
Enter the baseball's price $4.99
Total Cost: $5.39419
```

Line 09 modified (improved output with formatting)

```
09 print(f"Total Cost: ${totalCost:.2f}")
```

Output

```
Enter the item's name: baseball
Enter the baseball's price $4.99
Total Cost: $5.39
```

b) Beginning f-string formatting

`f"{expression[:formatSpecifier]}"`

- [ ] implies optional
- String delimiters can be double-quotes, single-quotes, triple single-quotes, or triple double-quotes
- Subset of format specifiers:

Format Specifier	Output type
s	string (default if : formatSpecifier is omitted)
d	integer
[.n]f	fixed-point (default is 6 digits of precision if only f specified) Optionally, If f preceded with .n, then n is to be replaced by a reasonable integer >= 0 (precision after decimal)

- FYI: Although not shown, a positive integer may precede the format specifier to indicate field width. This integer could also be preceded by <, >, or ^ to produce left, right, or center alignment in the field. There may also be a character specified to replace the default of padded spaces in a field
  Example → f"{totalCost:*<9.2f}" (produce a string with totalCost left aligned in a field width of 9 with a precision of 2 and a padding character of * instead of a space)

5	.	3	9	*	*	*	*	*

Code

```
01 petName = "Shadow"
02 weight = 22.5
03 height = 20
04 print(petName + ": " + str(weight) + " lbs, " + str(height) + '" tall.')
05 print(f'{petName}: {weight} lbs, {height}" tall.')
06 print(f'{petName:s}: {weight:f} lbs, {height:d}" tall.')
07 output = f'{weight:.2f} lbs, {height + 2:d}" tall.'
08 print(output)
```

Output

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
S	h	a	d	o	w	:		2	2	.	5		l	b	s	,		2	0	"		t	a	l	l	.									
S	h	a	d	o	w	:		2	2	.	5		l	b	s	,		2	0	"		t	a	l	l	.									
S	h	a	d	o	w	:		2	2	.	5	0	0	0	0	0		l	b	s	,		2	0	"		t	a	l	l	.				
2	2	.	5	0		l	b	s	,		2	2	"		t	a	l	l																	

c) Practice

Code

```
01 mealPrice = 10
02 housePrice = 1555300
03 discountPrice = 8
04
05 print(mealPrice)
06 print("$" + str(mealPrice))
07 print(f"${mealPrice}")
08 print(f"Price: ${mealPrice:.2f}")
09 print(f"{mealPrice + 1.5:.3f}, {mealPrice:d}")
10
11 print(f"${housePrice:,.2f}")
12 print(f"${housePrice:,d}")
```

What happens when a comma is properly inserted with d or f?

Output

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

d) Simple comparison to other formatting possibilities

Code

```
01 name = input("Name: ")
02 print(f"{name} is the person's name.")
03 print("%s is the person's name." % name)
04 print("{} is the person's name.".format(name))
```

Output (user input shown in red)

```
Name: Taylor
Taylor is the person's name.
Taylor is the person's name.
Taylor is the person's name.
```

We will not use .format() or %. Use f-strings instead. **Also**, use f-strings in print rather than comma operators. (The textbook used the older style of %)

6) Subset of String Methods

	methods	Semantics (meaning)
String	upper()	Returns a new uppercase version of the string
	lower()	Returns a lowercase version of the string
	replace(old, new)	Returns a string where every occurrence of the substring old is replaced with new.

IMPORTANT: For each method above, the original string object is not changed. A new modified string object is created and returned.

Format: stringObject.methodName(arg1, arg2, …)

Example: modifiedName = name.upper()

Code

```
01 school = "Iowa State"
02 print(school)
03 print(school.upper())
04 print(school.lower())
05 print(school)
06 school = school.upper()
07 print(school)
08 school = school.replace("IOWA", "KANSAS")
09 print(school)
10 school = school.replace("S", "Z")
11 print(school)
```

Output

```
Iowa State
IOWA STATE
iowa state
Iowa State
IOWA STATE
KANSAS STATE
KANZAZ ZTATE
```

7) Escape Sequences

What is the escape character?

A _____is embedded within a _____ and is followed by a character that has special meaning to Python when used after the backslash.

Common Escape Sequences	Semantics (Meaning)
	Newline. Advances output to the next line
	Horizontal tab: Advances output to the next tab stop.
	Double quote: Insert a double quote into a string delimited by double-quotes.
	Single quote: Insert a single quote into a string delimited by single-quotes.
	Backslash: Insert a backslash into a string.

Example:

Code

```
01 print("Line 1\nLine 2")
02 print("=" * 10)
03 print("Line 1\n\nLine 3")
04 print("column1\t\tcolumn2\tcolumn3")
05 print("She said, \"Touchdown Chiefs!\"")
06 print("C:\table")
07 print("C:\\table")
```

Output

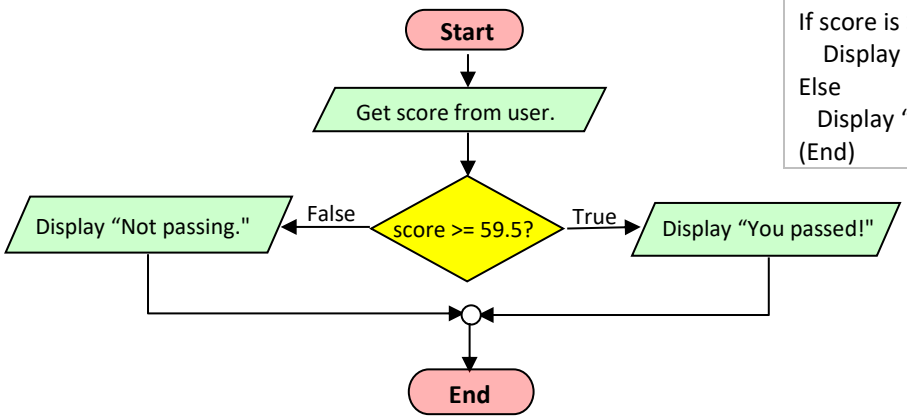
```
Line 1
Line 2
==========
Line 1

Line 3
column1     column2 column3
She said, "Touchdown Chiefs!"
C:  able
C:\table
```

A) Decisions: The if statement

1) Visual

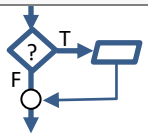
a) Problem/Task: Inform a user if they passed a test.

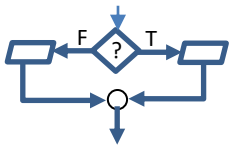
Pseudocode
(Start) Get score from user. If score is >= 59.5   Display "You passed!" Else   Display "Not passing." (End)



2) Syntax

Branch type	Syntax	Visual	Example

One-way branch	if condition:
	  statements
			`MIN_D = 59.5`
			`if score >= MIN_D:`
			`  print("You passed!")`

Two-way branch	if condition:
	  statements$_1$
	else :
	  statements$_2$
			`MIN_D = 59.5`
			`if score >= MIN_D:`
			`  print("You passed!")`
			`else:`
			`  print("Not passing.")`

condition: An expression that evaluates to True or False

Important: There is no condition after the else.

Warning: Remember the : after the condition and the else. After typing the : and pressing the enter key, the code on the next line is automatically indented. The indentation is critical to Python as it creates a block of code only associated with that branch.

Language Observation: Languages such as C++, C#, and Java require parentheses around the conditional expression. This is not required in Python, but paretheses may be used.

```
MIN_D = 59.5
if (score >= MIN_D) :
    print("You passed!")
```

3) Relational Operators and Boolean Expressions

a) What are the relational operators?

Relationship Tested	Math	Python
Equal to	=	
Not equal to	≠	
Greater than	>	>
Less than	<	<
Greater than or equal to	≥	
Less than or equal to	≤	

Coding Style Reminder: Insert one space on each side of relational (<, >, <=, >=, !=, ==), logical (and, or), and **assignment (=)** operators.

b) Evaluating simple relational expressions

Suppose that **`x = 5`** and **`y = -7`**

What are the values of the following relational expressions?

`x < y` → `5 < (-7)` → `False`

`x != y` → `5 != (-7)` → `True`

`x >= y` → `5 >= (-7)` → `True`

`x - y < 10` → `5 - (-7) < 10` → `12 < 10` →

4) If and If-Else Examples

Code

```
MIN_D = 59.5

# Get the score
score = float(input("Please enter the score: "))

# Display corresponding feedback
if score >= MIN_D:
    print("You passed!")
else:
    print("Not passing.")
...
```

… signifies that more code may occur as needed. The varied types of if statements may occur anywhere in your code.

Output (user input in **red**)

```
Please enter the score: 59.5
You passed!
```

Output (user input in **red**)

```
Please enter the score: 59.4
Not passing.
```

Code

```
# Need directions?
directionsNeeded = input("Directions needed? (Y or N): ")

# Display directions if needed
if directionsNeeded.upper() == "Y":
    print("From I-435, go south on Quivira past College")
    print("Take the first right into the JCCC campus")
```

Remember that an **else** block is <u>not</u> required for an if – only use when needed.

Output (user input in **red**)

```
Directions needed? (Y or N): y
From I-435, go south on Quivira past College
Take the first right into the JCCC campus
```

Output (user input in **red**)

```
Directions needed? (Y or N): N
```

Output (user input in **red**) How could we fix the following?

```
Directions needed? (Y or N): Yes
```

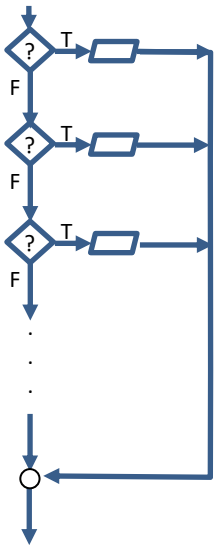
Fix:

5) Simplify code by removing duplication

Code	Simplified Code

price = float(input("Item Price? $"))
if price > 100:
  price = price * 0.8
  print(f"Discounted Price: ${price:,.2f}")
else:
  price = price * 0.9
  print(f"Discounted Price: ${price:,.2f}")
	price = float(input("Item Price? $"))
	if price > 100:
	  price = price * 0.8
	else:
	  price = price * 0.9
	print(f"Discounted Price: ${price:,.2f}")

6) Multi-way branching: More than two potential branches

General Syntax	Visual
if condition$_1$: statement(s) elif condition$_2$: statement(s) elif condition$_3$: statement(s) . . . elif condition$_n$: statement(s) *else :* *statement(s)*	
Note1: The italicized final **else** clause is **optional**. Note2: ()'s around each condition is optional.	

Example

MIN_A = 89.5
MIN_B = 79.5
MIN_C = 69.5
MIN_D = 59.5
 	
# Get the numeric grade 	
grade = float(input("Please enter a numeric grade: "))
 	
# Display corresponding letter grade 	
if grade >= MIN_A:
  print("A range.")
elif grade >= MIN_B:
  print("B range.")
elif grade >= MIN_C:
  print("C range.")
elif grade >= MIN_D:
  print("D range.")
else:
  print("Not Passing.")

Output (user input in **red**)
Please enter a numeric grade: 79.5 B range.
Output (user input in **red**)
Please enter a numeric grade: 59.49 Not Passing.

Demonstration: Illustrating code semantics with a debugger
Note: The previous example is really just nesting if-else statements within the trailing else blocks. The following code is also equivalent but results in too much rightward drift due to the nested indentation levels. Avoid this approach by using elif instead:

Nested If-else: rightward drift

```
# code for "constants" and input omitted

# Display corresponding letter grade
if grade >= MIN_A:
   print("A range.")
else:
   if grade >= MIN_B:
      print("B range.")
   else:
      if grade >= MIN_C:
         print("C range.")
      else:
         if grade >= MIN_D:
            print("D range.")
         else:
            print("Not Passing")
```

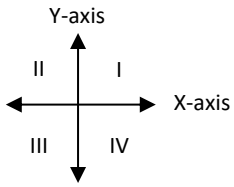
If possible, use **elif** to **avoid** situations like this where code unecessarily drifts right.

7) Nesting inside an if

a) Example 1

Code

```
xCoord = int(input("X coordinate? "))
yCoord = int(input("Y coordinate? "))

if xCoord > 0:
  if yCoord > 0:
    print"Quadrant I")
  elif yCoord < 0:
    print("Quadrant IV")
```



The above code is only a partial solution. Given only the code above,

Excluding the prompts, what is displayed if (5, 3) were the coordinates entered?

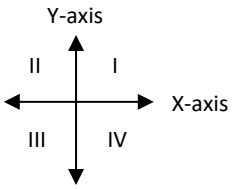
Excluding the prompts, what is displayed if (6, -3) were the coordinates entered?

Excluding the prompts, what is displayed if (2, 0) were the coordinates entered?

Excluding the prompts, what is displayed if (-3, -3) were the coordinates entered?

Careful: Indentation makes a difference in Python

```
xCoord = int(input("X coordinate? "))
yCoord = int(input("Y coordinate? "))

if xCoord > 0:
  if yCoord > 0:
    print("Quadrant I")
elif yCoord < 0:
  print("Quadrant IV")
```



Excluding the prompts, what is displayed if (-3, -5) were the coordinates entered?

b) Example 2

Code

```
##
#  This program computes income taxes, using a simplified tax schedule.
#

# Initialize constant variables for the tax rates and rate limits.
RATE1 = 0.10
RATE2 = 0.25
RATE1_SINGLE_LIMIT = 32000.0
RATE1_MARRIED_LIMIT = 64000.0

# Read income and marital status
income = float(input("Income: $"))
maritalStatus = input("Enter s for single, m for married: ").lower()

# Compute taxes due.
tax1 = 0 # tax due at first tax bracket income level
tax2 = 0 # remaining tax due if exceeding the first bracket income

if maritalStatus == "s" :
   if income <= RATE1_SINGLE_LIMIT :
      tax1 = RATE1 * income
   else :
      tax1 = RATE1 * RATE1_SINGLE_LIMIT
      tax2 = RATE2 * (income - RATE1_SINGLE_LIMIT)
else :
   if income <= RATE1_MARRIED_LIMIT :
      tax1 = RATE1 * income
   else :
      tax1 = RATE1 * RATE1_MARRIED_LIMIT
      tax2 = RATE2 * (income - RATE1_MARRIED_LIMIT)

totalTax = tax1 + tax2

# Display the tax.
print(f"The tax is ${totalTax:,.2f}")
```

Output (user input in **red**)

```
Income: $52000
Enter s for single, m for married: s
The tax is $8,200.00
```

Based on: Horstmann, C., and Necaise, D. Python for Everyone, 3rd edition.

B) Compound Conditions: Advanced Decision Making

1) Boolean (Logical) operators

Note: The expressions discussed in the table below are Boolean (AKA logical or conditional) expressions because they must evaluate to true or false (e.g., 2 < 3). They are not mathematical expressions (e.g., 2 + 3).

English	Python Logical Operator	In C, C++, C#, Java	Semantics
and	$expr_1$ **and** $expr_2$	$expr_1$ **&&** $expr_2$	Both expressions must be true for the result to be true.
or	$expr_1$ **or** $expr_2$	$expr_1$ **||** $expr_2$	One of the two expressions must be true for the result to be true.
not	**not** expr	**!** expr	Changes a true expression to false and vice versa.

Note: Remember that Python is case-sensitive. For example, **And** would produce a syntax error.

Coding Style Reminder: Insert one space on each side of relational (<, >, <=, >=, !=, ==), logical (and, or), and **assignment (=)** operators.

2) Examples

Given:

```
score1 = 5
score2 = 3
score3 = -1
response = 'Y'
```

Short-circuit evaluation: Once it can be determined if the entire condition is true or false, the rest of the condition is not evaluated. Thus, score2 > score3 would not be evaluated below.

```
score1 < score2
  5    <   3
       F
```

```
score1 < score2 and score2 > score3
```

```
score1 < score2 or  score2 > score3
     5 < 3      or       3 > (-1)
       F        or          T
```

```
not score1 <= score2
not   5    <=    3
not        F
           T
```

Short-circuit evaluation would apply here too. score3 > 1 would not need evaluation.

```
score2 > 0 or score3 > 1
```

```
response == 'y' or response == 'Y'
'Y'    == 'y' or    'Y'   == 'Y'
       F      or         T
              T
```

What if response held 'N' or 'X' instead?

Suppose y contains 0, x contains 10, and RATIO is a "named constant" holding some number. Why would the following code not produce a division by 0 error in the if statement condition?

```
if y > 0 and x / y < RATIO:
  print(x / y)
```

Language Observation: You may chain relational operators together in Python, but not in other languages like C++, C#, and Java

Chaining Works in Python:	Equivalent logic that also works in Python:	Does not work in Python :
`if x < y < z :` `  ...`	`if x < y and y < z :` `  ...`	`if x < y and < z :` `  ...`

Reminder: parentheses around the conditional expression is also permitted in Python

```
if (x < y and y < z) :
```

Does not work in C, C++, C#, Java (for primitive types):	Works in C, C++, C#, Java	Does not work in C, C++, C#, Java :
`if (x < y < z)` `  ...`	`if (x < y && y < z)` `  ...`	`if (x < y && < z)` `  ...`

3) More logic

a) Truth Tables

Suppose A and B represent logical (boolean) expressions (e.g x <= 3, y > 2)

A	B	A and B	A or B	not A	not B	not A or not B	not(A and B)	not (A or B)	not A and not B
T	**T**	T	T	F	F	F	F	F	F
T	**F**	F	T	F	T	T	T	F	F
F	**T**	F	T	T	F	T	T	F	F
F	**F**	F	F	T	T	T	T	T	T

b) De Morgan's Laws – Sometimes logical expressions may be simplified to make code easier to understand.

(i) The Laws

The Laws	Negation effects (opposite)

`not(expr1 and expr2)` → `not(expr1) or not(expr2)`
`not(expr1 or expr2)` → `not(expr1) and not(expr2)`
	`and` ⟷ `or`
	`!=` ⟷ `==`
	`<` ⟷ `>=`
	`>` ⟷ `<=`

(ii) Examples

```
not(response == 'n' or response == 'N')
```

```
not( a < b and not(c > d) )
```

The end goal is to remove each `not`.

(Also, as discussed below, the use of surrounding parentheses ensure that the outermost `not` is evaluated last due to operator precedence.)

(iii) Sample code with a compound condition

Code

```
01 LOWER = 0
02 UPPER = 10
03
04 # Get the number
05 print("Please guess a number between ", end = "")
06 guess = int(input(f"{LOWER} and {UPPER} inclusively: "))
07
08 # Check for an invalid number
09 if not(guess >= LOWER and guess <= UPPER) :
10     print("I'm sorry, you entered an invalid number.")
```

Output (user input in **red**)

```
Please guess a number between 0 and 10 inclusively: 12
I'm sorry, you entered an invalid number.
```

Note: Using De Morgan's Law, line 09 could also be written as:

C) Operator Precedence Updated

Precedence	Operators
Highest ↓	( )
	**
	unary + unary -
	* / // %
	+ -
	< > <= >= != == in not in
	not
	and
Lowest	or

Remember: logical and has higher precedence than logical or

Correct	Incorrect

`not False or True and False`
`True or True and False`
`True or False`
`True`
	`not False or True and False`
	`True or True and False`
	`True and False`
	`False`

D) FYI: The Conditional Operator (ternary operator)

1) Syntax

General Format	Semantics
`expr1 if condition else expr2`	**If** condition evaluates to true, the result is the value of expr1 **else** the result is the value of expr2.
Pattern: A conditional operator is useful when there is a two-way branch (if – else) with a similar statement in each branch (such as assignment to the same variable)	

2) Example

If-else statement

```
if grade >= PASS_LIMIT :
    message = "Passing"
else :
    message = "Not Passing"
```

Think of writing the repeated code in the parallel statements once, with a conditional operator for the rest.

Equivalent statement

```
message = "Passing" if (grade >= PASS_LIMIT) else "Not Passing"
```

E) Additional String Methods and Operations

1) Operation

Operation	Description
substring **in** s	Returns True if the string s contains substring; Else if returns False.

2) Methods

Methods	Description
s.count(substring)	Returns the number of non-overlapping occurrences of substring in the string s.
s.find(substring)	Returns the lowest index in the string s where substring begins, or –1 if substring is not found.
s.startswith(substring)	Returns True if the string s starts with substring; else it returns False.
s.endswith(substring)	Returns True if the string s ends with substring; else it returns False.
s.isalnum()	Returns True if string s consists of only letters or digits and len(s) >= 1; else it returns False.
s.isalpha()	Returns True if string s consists of only letters and len(s) >= 1; else it returns False.
s.isdigit()	Returns True if string s consists of only digits and len(s) >= 1; else, it returns False.
s.islower()	Returns True if all letters in string s are lowercase and len(s) >= 1; else it returns False.
s.isupper()	Returns True if all letters in string s are uppercase and len(s) >= 1; else, it returns False
s.isspace()	Returns True if string s consists of only white space characters (blank, newline, tab) and len(s) >= 1; else it returns False.

3) Practice

```
name = "Jack Jackson"
zipcode = "66210"
tempInput = "-2"
```

Expression	Value
`"Jack" == "jack"`	
`"Jack" in name`	
`"jack" not in name`	
`name.count("Jack")`	
`name.isalnum()`	
`"JackJohnson".isalnum()`	
`name.isalpha()`	
`zipcode.isdigit()`	
`tempInput.isdigit()`	
`name.isupper()`	

4) How are strings or individual characters compared?

a) The ASCII Table (**http://www.asciitable.com**) which is a subset of Unicode

Partial ASCII table

Decimal Value	Char		Decimal Value	Char		Decimal Value	Char		Decimal Value	Char
32	Space		56	8		80	P		104	h
33	!		57	9		81	Q		105	i
34	"		58	:		82	R		106	j
35	#		59	;		83	S		107	k
36	$		60	<		84	T		108	l
37	%		61	=		85	U		109	m
38	&		62	>		86	V		110	n
39	'		63	?		87	W		111	o
40	(		64	@		88	X		112	p
41	)		65	A		89	Y		113	q
42	*		66	B		90	Z		114	r
43	+		67	C		91	[		115	s
44	,		68	D		92	\		116	t
45	-		69	E		93	]		117	u
46	.		70	F		94	^		118	v
47	/		71	G		95	Underscore		119	w
48	0		72	H		96	`		120	x
49	1		73	I		97	a		121	y
50	2		74	J		98	b		122	z
51	3		75	K		99	c		123	{
52	4		76	L		100	d		124	|
53	5		77	M		101	e		125	}
54	6		78	N		102	f		126	~
55	7		79	O		103	g		127	Delete

b) Comparing characters (both strings of length 1)

Given:

```
response = 'y'
```

What are the values of the following expressions?

```
'A' < 'C' →                    →

'+' > '1' →                    →

response == 'y' →  'y' == 'y'  →   121 == 121   →

response < 'Y'  →   'y' < 'Y'  →   121 < _____   →
```

c) Comparing strings where at least one string has a length greater than 1

(i) Compare each string character by character left to right until either a mismatch is found, or the end of a string is reached.

(ii) If a character mismatch is found
the string containing the lower Unicode (ASCII) value of the two mismatched characters is less
Else if the strings are of different lengths
the shorter string is less
Else the strings are equal

Examples

Given:

```
string mascot = "Chief"
```

Expression	Mismatch?	Result
"KU" == "JCCC"	Yes K=75, J=74	False
"Olathe East" < "Olathe NW"	Yes E=69, N=78	True
mascot > "Chiefs"	No	False
mascot == "Chief"		
"mascot" == "Chief"		
"KState" > "KSTATE"		

A) Repetition Statements – Loops

1) Why loops?

Code

```
01 STATE_TAX_RATE = .065
02
03 # Get salary from user
04 salary = float(input("Please enter a salary: $"))
05
06 # Calculate income tax and display to user
07 incomeTax = salary * STATE_TAX_RATE
08 print(f"Your tax: ${incomeTax:,.2f}")
09
10 # Get salary from user
11 salary = float(input("Please enter a salary: $"))
12
13 # Calculate income tax and display to user
14 incomeTax = salary * STATE_TAX_RATE
15 print(f"Your tax: ${incomeTax:,.2f}")
16
17 # Get salary from user
18 salary = float(input("Please enter a salary: $"))
19
20 # Calculate income tax and display to user
21 incomeTax = salary * STATE_TAX_RATE
22 print(f"Your tax: ${incomeTax:,.2f}")
```

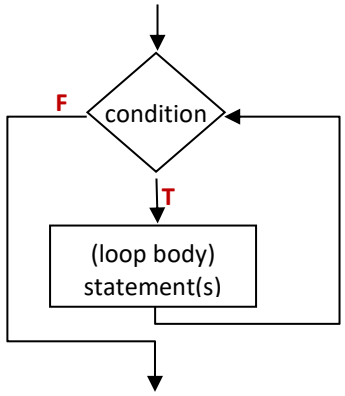
Output

```
Please enter a salary: $50000
Your tax: $3,250.00
Please enter a salary: $45500
Your tax: $2,957.50
Please enter a salary: $39250
Your tax: $2,551.25
```

What if the number of salaries isn't known until the program runs?

2) While statement

a) Visualization and Semantics

Flow Chart	Semantics
	If the condition is true, execute the statements inside the loop body. Then, return to test the loop condition. If the condition is still true, again execute the statements inside the loop body. Eventually, when the condition becomes false, jump to the next statement after the loop body.

b) Syntax

General Syntax

```
while condition :
    statement1
    statement2
    ...
    statementn
```

Language Observation: Parentheses around the while loop condition are not required in Python but are required in languages like C++, C#, and Java.

c) Counter-controlled while loops

Code

```
i = 1
LIMIT = 3
while i <= LIMIT :
  print(i)
  i = i + 1
print("finished")
```

Variable Trace

i:

Output

Iteration: Each time a single loop cycle completes
How many iterations occurred in the prior loop?

Code

```
01 i = 1
02 sum = 0
03
04 # Introduce the program and get the limit
05 print("This program will sum the integers " +
06       "from 1 to a user specified limit.\n")
07 limit = int(input("Please enter a limit less than 20: "))
08
09 # Calculate the sum
10 while i <= limit :
11   sum += i
12   i += 1
13
14 # Display the result
15 print(f"The sum of the integers from 1 to {limit} is {sum}")
```

```
limit: 5

i: 1 2 3 4 5 6

sum: 0 1 3 6 10 15
```

Output

```
This program will sum the integers from 1 to a user specified limit.

Please enter a limit less than 20: 5
The sum of the integers from 1 to 5 is 15.
```

Note: The variable i is known as a **counter** variable as it controls the number of loop iterations (how many times the loop body executes). Because single letter variable names of i, j, and k are commonly used for counter variables, they are considered descriptive of their intended use.

Coding Style: Single letter identifiers may only be used for counter variables (e.g., i, j, k).

d) Sentinel-controlled while loops

(i) What is a sentinel value?

Sentinel value: A value that signals the end of data processing. It lies outside the potential range of data values.

What might be potential sentinel values for data entry of bowling scores?

What might be potential sentinel values for processing Kansas temperature data?

(ii) Example: Fixing the previous tax program dilemma

Code

```
01 STATE_TAX_RATE = .065
02
03 # Get salary from user
04 salary = int(input("Please enter a salary or -1 to quit: $"))
05
06 # Compute taxes until the user quits
07 while salary >= 0 :
08     # Calculate income tax and display to user
09     incomeTax = salary * STATE_TAX_RATE
10     print(f"Your tax: ${incomeTax:,.2f}")
11
12     # Get another salary from user
13     salary = int(input("Please enter a salary or -1 to quit: $"))
```

Output

```
Please enter a salary or -1 to quit: $50000
Your tax: $3,250.00
Please enter a salary or -1 to quit: $45500
Your tax: $2,957.50
Please enter a salary or -1 to quit: $-1
```

Warning: Due to imprecise storage, do not use != or == when comparing floating-point values.

```
01 value = .1
02 while value != 1.0 :
03     print(f"Value: {value}")
04     value += .1
05 print("The end")
```

Output

```
Value:  0.1
Value:  0.2
Value:  0.30000000000000004
Value:  0.4
Value:  0.5
Value:  0.6
Value:  0.7
Value:  0.7999999999999999
Value:  0.8999999999999999
Value:  0.9999999999999999
Value:  1.0999999999999999
Value:  1.2
Value:  1.3
Value:  1.4000000000000001
.
```

Infinite Loop

```
.
.
```

Potential solution – change line 2 (See the strategy below for a better solution)

```
02  while (value < 1.0):
```

Output

```
Sum:  0.1
Sum:  0.2
Sum:  0.30000000000000004
Sum:  0.4
Sum:  0.5
Sum:  0.6
Sum:  0.7
Sum:  0.7999999999999999
Sum:  0.8999999999999999
Sum:  0.9999999999999999
The end
```

This change at least stops the loop, although it is inexact because .999 repeating (in other words 1) is still displayed.

Coding Strategy: In Python 3.5 and higher, math.isclose(floatNum1,floatNum2) can be used to compare for **floating point** equality. Not used for integers.

```
01  import math
02  value = .1
03  while not math.isclose(value, 1.0):
04      print(f"Value: {value}")
05      value += .1
06  print("The end")
```

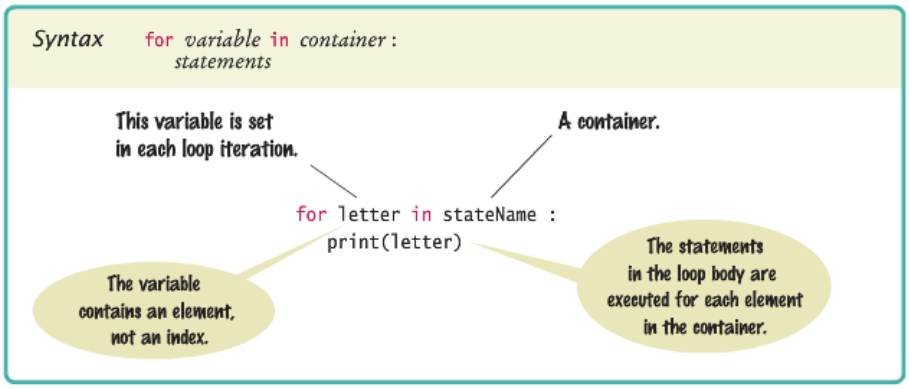
Output

```
Value:  0.1
Value:  0.2
Value:  0.30000000000000004
Value:  0.4
Value:  0.5
Value:  0.6
Value:  0.7
Value:  0.7999999999999999
Value:  0.8999999999999999
end
```

Better solution

Note: The key point of this is to be careful with comparing floating point numbers using != or == . The result may not be what you expect. You are not required to use math.isclose in this beginning course unless otherwise specified. Using an inequality such as <, <=, >, or >= is ok.

3) For statement
   a) Iterating (looping) through elements in a container
      (i) Syntax



A string is a type of a container because it is a collection of letters.

(ii) Examples

Code	Equivalent While Loop Code

`school = "JCCC"`
`for letter in school :`
`  print(letter)`
	`school = "JCCC"`
	`i = 0`
	`while (i < len(school)) :`
	`  print(school[i])`
	`  i += 1`

Output	
J C C C	

Code

`school = "University of Kansas"`
`numCapitalLetters = 0`
`for character in school :`
`  if character.isupper() :`
`  numCapitalLetters += 1`
 	
`print(f"Number of capital letters: {numCapitalLetters}")`

Output
Number of capital letters: 2

Code

`phoneNumber = "913-238-1544"`
`phoneNumberDigitsOnly = ""`
`for symbol in phoneNumber:`
`  if symbol.isdigit() :`
`  phoneNumberDigitsOnly += symbol`
`print(f"Modified phone number: {phoneNumberDigitsOnly}")`

Output
Modifed phone number: 9132381544

b) Iterating through a sequence of numbers

(i) The range function

First → Mathematical notation with two pairs of numbers

[or]	
(or)	
Example	Represents this sequence of numbers:
[0, 10)	
[-5, 5]	
(0, 11)	
(0, 10]	

Syntax	Semantics
range(n)	returns the sequence [0, n)
range(begin, end)	returns the sequence: [begin, end)
range(begin, end, step)	returns the sequence [begin, end) where the increment or decrement is the value of step rather than the default 1

Examples	Resulting sequence
range(5)	
range(10)	
range(-1, 3)	
range(1, 11)	
range(1, 10, 3)	
range(10, 2, -2)	

(ii) Examples

Code	Code	Code
`for i in range(4) :` `  print(i, end = " ")`	`for i in range(1, 4) :` `  print(i, end = " ")`	`for i in range(8, 1, -2) :` `  print(i, end = " ")`
Output	Output	Output
`0 1 2 3`	`1 2 3`	`8 6 4 2`

Code

`limit = int(input("Count from 1 through which number? "))`
 	
`for i in range(1, limit + 1) :`
`  print(i, end = " ")`

Output (user input shown in red)
`Count from 1 through which number? 8` `1 2 3 4 5 6 7 8`

Code

`balance = 1000.0`
`numChecks = int(input("Enter number of checks to process: "))`
 	
`for checkNumber in range(numChecks) :`
`  checkAmount = float(input("Check amount: $"))`
`  balance -= checkAmount`
 	
`print(f"Remaining balance: ${balance:,.2f}")`

Output (user input shown in red)

`Enter number of checks to process: 3`
`Check amount: $25`
`Check amount: $50.55`
`Check amount: $20`
`Remaining balance: $904.45`

B) Repetition Statements – Nested Loops

1) Example – Tracing by hand and by following program flow in a debugger

Code	Hand Trace

`for i in range(3) :`
`  for j in range(4) :`
`  print(i + j, end = " ")`
`  print()`
	`i:`
	
	`j:`

Output	

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

Code	Hand Trace

`for i in range(4) :`
`  for j in range(3) :`
`  print(i + j, end = " ")`
`  print()`
	i:
	
	j:

Output	
0 1 2 1 2 3 2 3 4 3 4 5	

2) What might this example display?

Code	
01	`for i in range(1, 3) :`
02	`  for j in range(1, 4) :`
03	`  print("*", end = " ")`

Output																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

How could we fix the code to print the second row under the first?

3) Example from Python for Everyone by Horstmann and Necaise:

Code	
01	`##`
02	`#  This program computes the average exam grade for multiple students.`
03	`#`
04	
05	`# Obtain the number of exam grades per student.`
06	`numExams = int(input("How many exam grades does each student have? "))`
07	
08	`# Initialize moreGrades to a non-sentinel value.`
09	`moreGrades = "Y"`
10	
11	`# Compute average exam grades until the user wants to stop.`
12	`while moreGrades == "Y" :`
13	
14	`  # Compute the average grade for one student.`
15	`  print("Enter the exam grades.")`
16	`  total = 0`
17	`  for i in range(1, numExams + 1) :`
18	`  score = int(input(f"Exam {i}: "))  # Prompt for each exam grade.`
19	`  total = total + score`
20	
21	`  average = total / numExams`
22	`  print(f"The average is {average:.2f}")`
23	
24	`  # Prompt as to whether the user wants to enter grades for another student.`
25	`  moreGrades = input("Enter exam grades for another student (Y/N)? ")`
26	`  moreGrades = moreGrades.upper()`

C) Random Numbers

1) A subset of the random module API (Application Programmer Interface)

The random module	
Function	Description
random()	Returns a random number in the range [0.0, 1.0)
randint(low, high)	Returns a random integer in the range [low, high]

2) Examples

Code

```
MAX_NUMBERS = 5
from random import random

for i in range(MAX_NUMBERS) :
    randomValue = random()
    print(randomValue)
```

Output

```
0.6420739708464852
0.1820310890218696
0.645708385722477
0.6089732902480582
0.9487110511651018
```

Code

```
MAX_NUMBERS = 5
from random import random

for i in range(MAX_NUMBERS) :
    print(random())
```

Output

```
0.9693200738744346
0.04593691230422503
0.016086962431010376
0.07029865839094651
0.841593362492279
```

Code

```
from random import randint
LOW = 1
HIGH = 100
MAX_NUMBERS = 5

for i in range(MAX_NUMBERS) :
    randomNumber = randint(LOW, HIGH)
    print(randomNumber)
```

Output

```
75
28
18
82
52
```

Code

```
from random import randint
LOW = -20
HIGH = 20
MAX_NUMBERS = 5

for i in range(MAX_NUMBERS) :
    print(randint(LOW, HIGH))
```

Output

```
-1
17
-17
5
16
```

Code

```
from random import randint
MAX_ITERATIONS = 5
NUM_DIE_SIDES = 6

for i in range(MAX_ITERATIONS) :
    die1 = randint(1, NUM_DIE_SIDES)
    die2 = randint(1, NUM_DIE_SIDES)
    print(die1, die2)
```

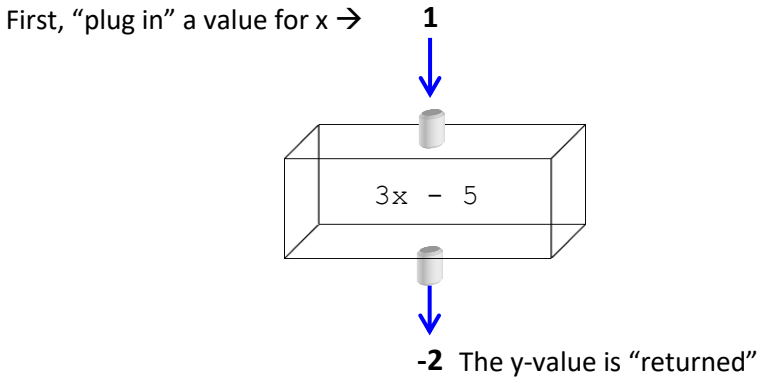
Output

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

A) Functions

1) Why use functions?
   a) To break a problem into small, manageable pieces.
   b) To allow reuse of code pieces which you or others have written.
   c) To reduce repeated code.

2) If you've learned algebra, then you've been using functions even before taking a programming class.
   Slope y-intercept form → y = mx + b → Example: y = 3x - 5



3) We've already used functions in Python.
   Which are the functions?

Code
01 day = input("Which day of the week is it? ") 02 print(f"Today is {day}.")
Output (user input in **red**)
Which day of the week is it? **Monday** Today is Monday.

B) Calling (Invoking) Python Functions

1) Syntax Pattern

Function Calling Syntax
functionName(arg_1, arg_2, ..., arg_N)

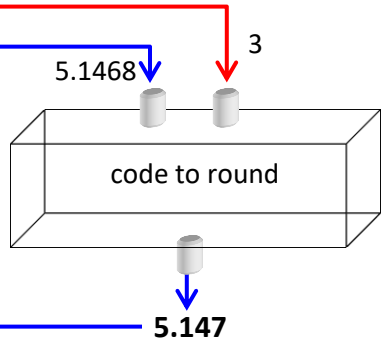
Note 1: The function **arguments** are also termed **actual parameters**: They are the values sent to a function or method.
Note 2: A function call requires a name, a pair of parentheses, and optionally arguments to send.

2) Visual example using the Python built-in **round** function

```
value = 5.1468
precision = 3

value = round(value, precision)


print(value)
```

Output

```
2.547
```



Question: What are the arguments (actual parameters) and the returned value for **round(3.14159, 2)**?

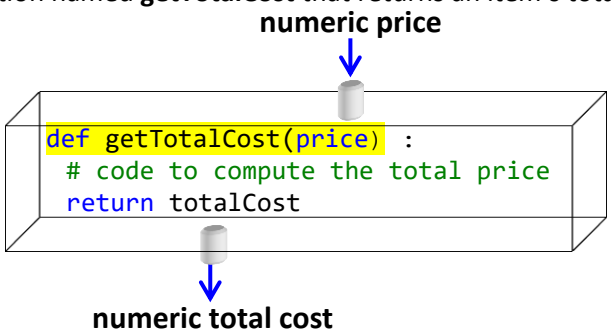
3) Recall that function calls may be nested:

```
price = float(input("Item price: $"))
print(round(value, precision))
```

```
import math
length = 3;
width = 4;
print(f"Diagonal length: {math.sqrt( pow(length, 2) + pow(width, 2) ):.1f}")
```

B) Programmer-defined Python Functions

1) Visualizing a function named **getTotalCost** that returns an item's total cost with tax



2) What pieces are needed to define (create) a Python function?

a) The `def` keyword.

b) A function name.

c) A set of parentheses with a trailing colon.

d) An optional comma separated list of parameters inside the parentheses.

Formal parameters: The name for the parameters in a function definition.

e) A function body consistsing of one or more indented statements.

Optionally, one or more return statements may exist inside the function body. When a return statement is encountered, the function exits immediately with the returned value.

None is returned if no return value is specified, or no return statement exists.

Function Definition Syntax
`def functionName(formalParam1, formalParam2, . . .) :` `  statement(s)`

Coding Style:

- Names for functions and formal parameters are in **lowerCamelCase**.
- Function names must be descriptive of the action performed and formal parameters are to be descriptive of the data stored.

3) Examples

Code

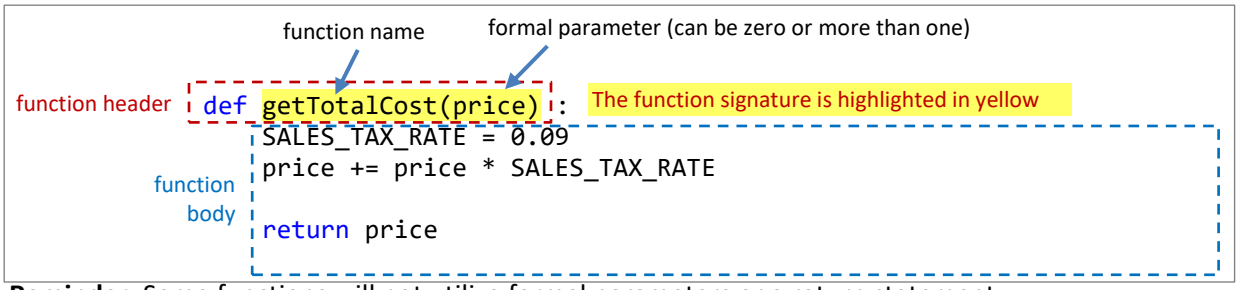
```
01 # getTotalCost computes the total cost of an item with tax
02 # @param price The price of an item
03 # @return The total price with tax
04 def getTotalCost(price) :
05     SALES_TAX_RATE = 0.09
06     price += price * SALES_TAX_RATE
07
08     return price
09
10 price = float(input("Item price? $"))
11 totalCost = getTotalCost(price)
12 print(f"Total cost: ${totalCost:.2f}")
```

The function header docs could instead be written with a doc string:

```
"""
getTotalCost computes the total cost of an item with tax.
@param price The price of an item
@return The total price with tax
"""
```

Output (input shown in **red**)

```
Item price? $49.95
Total cost: $54.45
```



Reminder: Some functions will not utilize formal parameters or a return statement

Coding Style: Function documentation is placed immediately above its definition.

Format:(List the description followed by any @param @preconndition, and/or @return tags as needed.)

```
# One or more sentences describing the purpose of the function.
# @param paramName Description of parameter 1     (Only if there are parameters)
# ...
# @param paramName Description of parameter n
# @precondition What must be true for the function to run correctly? (Only if required.)
# @return Description of the return value (Only if a value other than None is returned)
def functionName(...) :
```

or

```
“””
One or more sentences describe the purpose of the function.
@param paramName Description of parameter 1     (Only if there are parameters)
  ...
@param paramName Description of parameter n
@precondition What must be true for the function to run correctly? (Only if required)
@return Description of the return value (Only if a value other than one is returned)
“””
def functionName(...) :
```

Note that the prior code that calls getTotalCost can be written in various ways

Code

```
totalCost = getTotalCost(float(input("Item price? $")))
print(f"Total cost: ${totalCost:.2f}")
```

Code

```
price = float(input("Item price? $"))
print(f"Total cost: ${getTotalCost(price):.2f}")
```

Code

```
print(f"Total cost: ${getTotalCost(float(input('Item price? $'))):.2f}")
```

Warning: Be careful with too much nesting as the code can become less readable.

If a function is created in the same file as the code which calls it, it must be defined before any calls are made to the function. **The following will not work:**

Warning: Code that does not work

```
01 price = float(input("Item price? $"))
02 print(f"Total cost: ${getTotalCost(price):.2f}")
03
04 # getTotalCost computes the total cost of an item with tax
05 # @param price The price of an item
06 # @return The total price with tax
07 def getTotalCost(price) :
08     SALES_TAX_RATE = 0.09
09     price += price * SALES_TAX_RATE
10
11     return price
```

Output (User input in **red**)

```
Item price? $49.95
Traceback (most recent call last):
  File "C:\Users\vango\TotalCost.py", line 2, in <module>
    print(f"Total cost: ${getTotalCost(price):.2f}")
builtins.NameError: name 'getTotalCost' is not defined
```

One convention is to create a function named main and use that to start the function calls. When main() begins execution, all the prior functions have been defined.

Code

```
01 def main():
02     price = float(input("Item price? $"))
03     print(f"Total cost: ${getTotalCost(price):.2f}")
04
05 # getTotalCost computes the total cost of an item with tax
06 # @param price The price of an item
07 # @return The total price with tax
08 def getTotalCost(price) :
09     SALES_TAX_RATE = 0.09
10     price += price * SALES_TAX_RATE
11
12     return price
13
14 main()
```

There are no function header docs for the main function

The function header docs could instead be written with a doc string:

```
"""
getTotalCost computes the total cost of an item with tax.
@param price The price of an item
@return The total price with tax
"""
```

Output (User input in **red**)

```
Item price? $100
Total cost: $109.00
```

Larger function

Code

```
01 def main():
02     intensityValue = int(input("Earthquake intensity? "))
03     print("Earthquake intensity level " +
04           f"{intensityValue} -> {getEarthquakeShaking(intensityValue)} Shaking")
05
06 # getEarthquakeShaking determines the amount of shaking
07 # associated with a given earthquake intensity
08 # @param intensity The recorded intensity
09 # @return The shaking description
10 def getEarthquakeShaking(intensity):
11     shaking = "Unknown intensity"
12     if intensity == 2 or intensity == 3:
13         shaking = "Weak"
```

```
14      elif intensity == 4:
15          shaking = "Light"
16      elif intensity == 5:
17          shaking = "Moderate"
18      elif intensity == 6:
19          shaking = "Strong"
20      elif intensity == 7:
21          shaking = "Very Strong"
22      elif intensity == 8:
23          shaking = "Severe"
24      elif intensity == 9:
25          shaking = "Violent"
26      elif intensity == 10:
27          shaking = "Extreme"
28
29      return shaking
30
31  main()
```

Output (User input in **red**)

```
Earthquake intensity? 10
Earthquake intensity level 10 --> Extreme Shaking
```

Suppose the function is to instead show the shaking description rather than return it

Code

```
01  def main():
02      intensityValue = int(input("Earthquake intensity? "))
03      print(f"Earthquake shaking for intensity level {intensityValue} -> ", end = "")
04      showEarthquakeShaking(intensityValue)
05
06  # showEarthquakeShaking displays the amount of shaking
07  # associated with a given earthquake intensity
08  # @param intensity The recorded intensity
09  def showEarthquakeShaking(intensity):
10      shaking = "Unknown intensity"
11      if intensity == 2 or intensity == 3:
12          shaking = "Weak"
13      elif intensity == 4:
14          shaking = "Light"
15      elif intensity == 5:
16          shaking = "Moderate"
17      elif intensity == 6:
18          shaking = "Strong"
19      elif intensity == 7:
20          shaking = "Very Strong"
21      elif intensity == 8:
22          shaking = "Severe"
23      elif intensity == 9:
24          shaking = "Violent"
25      elif intensity == 10:
26          shaking = "Extreme"
27
28      print(shaking)
29
30  main()
```

Because there is no return statement, there is no @return in the function header docs

Output (User input in **red**)

```
Earthquake intensity? 8
Earthquake shaking for intensity level 8 -> Severe
```

More than one parameter

Code

```
01 from datetime import datetime
02 def main():
03     print(getReceipt("Baseball", 4.95))
04
05 # getReceipt builds a sales receipt based upon an item and its price
06 # @param item The name of the item
07 # @param price The price of the item
08 # @return The receipt
09 def getReceipt(item, price) :
10     SALES_TAX_RATE = 0.09
11     tax = price * SALES_TAX_RATE
12     timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
13
14     # Build receipt
15     receipt = "*" * 25
16     receipt += "\n" + timestamp
17     receipt += "\n" + f"{item:<15s}${price:.2f}"
18     receipt += "\n" + f'{"Tax":<15s}${tax:.2f}'
19     receipt += "\n\n" + f'{"Total":<15s}${price + tax:.2f}'
20
21     return receipt
22
23 main()
```

Two formal parameters, so two @param tags are needed.

Output

```
*************************
2022-09-17 17:34:30
Baseball       $4.95
Tax            $0.45

Total          $5.40
```

Note : A function may contain multiple returns; however, be careful as multiple exit points may make a function more difficult to test.

Code

```
01 def getEarthquakeShaking(intensity):
02     if intensity == 2 or intensity == 3:
03         return "Weak"
04     elif intensity == 4:
05         return "Light"
06     elif intensity == 5:
07         return "Moderate"
08     elif intensity == 6:
09         return "Strong"
10     elif intensity == 7:
11         return "Very Strong"
12     elif intensity == 8:
13         return "Severe"
14     elif intensity == 9:
15         return "Violent"
16     elif intensity == 10:
17         return "Extreme"
18
19     return "return intensity"
```

4) Function stubs

Programmers can outline their code with function stubs that can be called. They can then fill in the function body later.

Code

```
01  def main():
02      showEarthQuakeScale()
03      shaking = getEarthQuakeShaking(1)
04      showFormattedEarthQuakeReport("")
05
06  def getEarthQuakeShaking(intensity):
07      return ""
08
09  def showEarthQuakeScale():
10      return
11
12  def showFormattedEarthQuakeReport(report):
13      pass
14
15  main()
```

Either the `pass` keyword or an empty `return` could be used for an unimplemented funciton that will not return a specified value. In both cases, `None` is returned. Note that `pass` could even used in other situations: for example, as placeholders for future code in the body of selection or looping statements.

C) Note about **Function Overloading**: Functions with the same name but different parameter lists.

Language Observation: Python does not permit function/method overloading. Other languages such as C++, C#, and Java allow this.

Warning: Code that does not work (function overloading)

```
def main() :
    displayWelcome("Logan")
    print()
    displayWelcome()

# displayWelcome displays a customized, introductory welcome message
# to the game.
# @param name The name of the player
def displayWelcome(name) :
    print(f"{name}, welcome to Blackjack. In this game, you will ")
    print("compete against the computer. The winner is ")
    print("the closest to 21 without going over.")

# displayWelcome displays an introductory welcome message to the game.
def displayWelcome() :
    print("Welcome to Blackjack. In this game, you will ")
    print("compete against the computer. The winner is ")
    print("the closest to 21 without going over.")

main()
```

Attempt to overloaded functions

Notice that if a function has no parameters, there would be no @param in the documentation

FYI: Python can simulate method overloading by utilizing optional parameters. Other languages do also allow optional parameters.

Code that does work – Using optional parameters

```
def main() :
    displayWelcome("Logan")
    print()
    displayWelcome()

# displayWelcome displays a customized, introductory welcome message
# to the game.
# @param name The name of the player
```

```
def displayWelcome(name = "") :
    if (len(name) > 0):
        print(f"{name}, ", end = "")
    print("Welcome to Blackjack. In this game, you will ")
    print("compete against the computer. The winner is ")
    print("the closest to 21 without going over.")

main()
```

Output

```
Logan, Welcome to Black Jack. In this game, you will
compete against the computer. The winner is
the closest to 21 without going over.

Welcome to Black Jack. In this game, you will
compete against the computer. The winner is
the closest to 21 without going over.
```

D) Parameter Passing and Stack Frames

Note 1: To obtain a beginning understanding of how function calling works, the examples simplify parameter passing by showing values as stored on the stack for numbers, strings, and Booleans. Recall, however, that immutable variables such as these essentially refer to (contain the location of) the object that stores that value.

Note 2: Stack Frame is synonymous with Activation Record.

Note 3: The entire stack on the right may not be used when tracing the calls.

1) Example 1

Code

```
01 def main():
02   price = 10
03   print(f"Total cost: " +
04         $"{getTotalCost(price):.2f}")
05
06 def getTotalCost(price) :
07   SALES_TAX_RATE = 0.09
08   price += price * SALES_TAX_RATE
09
10   return price
11
12 main()
```

One stack frame (activation record). If there are more than 3 local variables, more than 3 rows will be used to track the variables.

Stack Info	Stack Var Names	Stack Var Values
Function Name		
Return Line #		
Return Value		
Function Name		
Return Line #		
Return Value		
Function Name		
Return Line #		
Return Value		

Python App

Output

```
Total cost: $10.90
```

2) Example 2

```
01 def main():
02   low = int(input("Low? "))
03   high = int(input("High? "))
04   print(getSummation(low, high))
05
06 def getSummation(low, high):
07   sum = 0
08   for i in range(low, high + 1):
09     sum += i
10
11   return sum
12
13 main()
```

More than 3 local variables in getSummation (4 total), so a 4th row is added for the getSummation activation record

Stack Info	Stack Var Names	Stack Var Values
Function Name		
Return Line #		
Return Value		
Function Name		
Return Line #		
Return Value		
Python App		

Output (User input in **red**)

```
Low? 1
High? 5
15
```

Example 3

```
01 def main():
02     LIMIT = 2
03     for i in range(1, LIMIT + 1):
04         animalCount(i, "Woof")
05
06 def animalCount(count, sound):
07     for i in range(count):
08         print(sound, end = " ")
09     print()
10
11 main()
```

Stack Info	Stack Var Names	Stack Var Values
Function Name		
Return Line #		
Return Value		
Function Name		
Return Line #		
Return Value		
Function Name		
Return Line #		
Return Value		
Python App		

Output
Woof Woof Woof

E) Variable scope

1) Terms

a) **Scope** – The scope of an identifier is the part of the program in which it is _____.

b) **Local variable** – A variable whose scope is limited to a function (or method).

- Note: A local variable is visible (accessible) from the point it is defined until the end of the function. In other words, its scope is from the place it is created until the end of the function.

2) Examples with local variables

Code

```
def main() :
  LIMIT = 11
  sum = 0
  for i in range(1, LIMIT):
    square = i * i
    sum += square

  print(i, sum)

# Start the program
main()
```

Output

```
10 385
```

1. Will this code completely run?

```
def main() :
  LIMIT = 11
  sum = 0
  for i in range(1, LIMIT):
    square = i * i
    sum += square

  print(i, sum)
  print(LIMIT, square)

# Start the program
main()
```

2. Will this code completely run?

```
def main() :
  LIMIT = 11
  sum = 0
  for i in range(1, LIMIT):
    print(square)
    square = i * i
    sum += square

# Start the program
main()
```

3. Will this code completely run?

```
def main() :
  sideLength = 10
  print(cubeVolume())

def cubeVolume() :
  return sideLength ** 3

main()
```

4. Will this code completely run?

```
def main() :
  result = cubeVolume(10)
  print(result)

def cubeVolume(sideLength) :
  result = sideLength ** 3
  return result

main()
```

Be careful: Example "1. Will this code completely run?" would not completely run if the for loop were not executed (e.g., LIMIT = 0). Why?

Language Observation: Example "1. Will this code completely run?" would not run in languages like C++, C#, and Java: The definition of square inside the for-loop would cause its scope to be limited to the for-loop.

3) Global variables
   a) Notes
      (i) **Global variable**: A variable that is defined outside a function or class.
      (ii) The scope of a global variable is the point at which it is defined to all code in the file that runs after the definition. However, if a function wishes to modify a global variable, it must include the `global` declaration.
      (iii) It is best to avoid global variables.
   b) Examples

Would this code run?

```
def main() :
  print(cubeVolume())

def cubeVolume() :
  return sideLength ** 3

main()
sideLength = 10
```

Would this code run?

```
def main() :
  print(cubeVolume())

def cubeVolume() :
  return sideLength ** 3

sideLength = 10
main()
```

Code	What happened here?

`balance = 10000`
 	
`def main() :`
`  checkAmount = 100`
`  withdraw(checkAmount)`
`  print("Balance in main: " +`
`  f"${balance:,.2f}")`
 	
`def withdraw(amount) :`
`  global balance`
`  if balance >= amount :`
`  balance -= amount`
`  print("Balance in withdraw: " +`
`  f"${balance:,.2f}")`
 	
`main()`
	`balance = 10000`
	
	`def main() :`
	`  checkAmount = 100`
	`  withdraw(checkAmount)`
	`  print("Balance in main: " +`
	`  f"${balance:,.2f}")`
	
	`def withdraw(amount) :`
	`  balance = 9000`
	`  if balance >= amount :`
	`  balance -= amount`
	`  print("Balance in withdraw: " +`
	`  f"${balance:,.2f}")`
	
	`main()`

Output	**Output**
`Balance in withdraw: $9,900.00` `Balance in main: $9,900.00`	`Balance in withdraw: $8,900.00` `Balance in main: $10,000.00`

Code

`balance = 10000`
 	
`def main() :`
`  checkAmount = 100`
`  withdraw(checkAmount)`
`  print("Balance in main: " +`
`  f"${balance:,.2f}")`
 	
`def withdraw(amount) :`
`  global balance`
`  balance = 9000`
`  if balance >= amount :`
`  balance -= amount`
`  print("Balance in withdraw: " +`
`  f"${balance:,.2f}")`
 	
`main()`

Output
Balance in withdraw: $8,900.00 Balance in main: $8,900.00

Note: Elements contained inside a global dictionary or global list can be modified inside a function without specifying the global keyword.

Language Observation: In C++, the global keyword is not needed to change the value of a global variable. Global variables are essentially not possible in Java or C# but can be somewhat simulated with the use of public static variables defined within a public static class.

c) Self-check Exercises (Source: Horstmann, C., and Necaise, D. Python for Everyone, 3rd edition.)

• 1. In the program below, what are all the variables that can be legally displayed using the incomplete call to the `print` function?

```
def main() :
   limit = 5
   for k in range(1, limit + 1)
      print(computeResult(k))

   print(_______)

def computeResult(value) :
   result = value * 2
   return result

main()
```

- ☐ result
- ☐ limit
- ☐ value, result
- ☐ limit, k

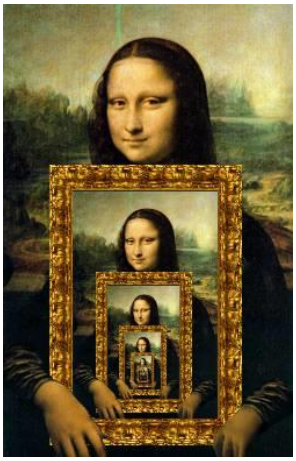
• 2. In the program below, what variable name in the program cannot legally be used in the incomplete assignment statement?

```
def main() :
   number = 2
   print(computeResult(number))

def computeResult(value) :
   result = 1
   for k in range(1, value + 1) :
      temp = _______
      result = result * k
   return result

main()
```

- ☐ value
- ☐ result
- ☐ k
- ☐ number

F) Recursion – When a Method Calls Itself

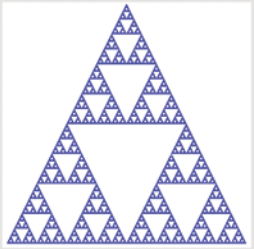
1) Recursion involves a pattern that repeats



www.dominiek.eu/blog/?m=200711

http://www.wired.com/images_blogs/cult_of_mac/recursion300400.jpg

FIGURE 7-3: A Sierpinski gasket



Herbert, C.W. (2007) An Introduction to Programming using Alice, p. 190

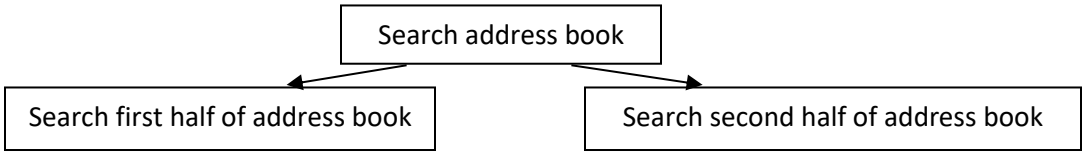
FIGURE 7-4: The Sierpinski algorithm through five levels of recursion

Herbert, C.W. (2007) An Introduction to Programming using Alice, p. 190

2) How is it used in problem-solving?

a) Searching a cell phone address book sorted by last name

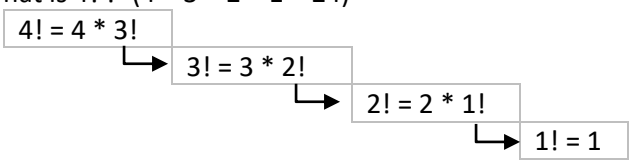
Break a problem into smaller mirror images of the same problem.



Once the problem is small enough (base case), solve it, and then backtrack to solve the remaining problems if needed.

b) Calculating a factorial

What is 4! ? (4 * 3 * 2 * 1 = 24)



n! = n * (n – 1) * (n – 2) * … * 1

c) Note: Recursion is helpful if it is difficult to solve a problem using iteration (a loop).

2) The basics

a) A recursive method is a method that calls itself.

b) A recursive method has:

- A _____ case that stops the recursive calls. (It may return a value.)
- A _____ case where the method calls itself. **This call must decrease the size of the problem to solve.**

G) Recursive Code Examples

1) Factorial – recursive solution

4! = 4 * 3 * 2 * 1 = 24

5! = 5 * (4 * 3 * 2 * 1) → 5 * 4! = 120

Code	Stack Info	Var Names	Var Values

```
01 def main() :
02   print(factorial(4))
03
04 # @precondition facLimit >= 0
05 def factorial(facLimit):
06   if facLimit <= 1:
07     return 1
08   else:
09     return facLimit * \
10       factorial(facLimit - 1)
11 main()
```

"Function name", "Return Line #", and "Return Value" are not written in this time, but the procedure is the same as before. main has no local variables, and facLimit has 1 local variable, so counting up 3 for each frame (activation record) will suffice.

Output (One symbol per cell)																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

2) Factorial – iterative solution

```
01 def main() :
02   print(factorial(4))
03
04 # @precondition facLimit >= 0
05 def factorial(facLimit):
06   result = 1
07   for i in range(1, facLimit + 1):
08     result *= i
09   return result
10
11 main()
```

3) Summation – recursive solution

summation(3) → 3 + 2 + 1 = 6

summation(4) → 4 + 3 + 2 + 1 = 4 + summation(3)

Code	Stack Info	Var Names	Var Values
01 def main() : 02 print(summation(3)) 03 04 # @precondition upper >= 1 05 def summation(upper): 06 if (upper == 1): 07 return 1; 08 else: 09 return upper + \ 10 summation(upper - 1) 11 12 main()			

Output (One symbol per cell)																		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

4) Summation – iterative solution

```
01 def main() :
02   print(summation(3))
03
04 # @precondition upper >= 1
05 def summation(upper):
06   sum = 0
07   for i in range(1, upper + 1):
08       sum += i
09   return sum
10
11 main()
```

5) Fibonacci numbers

Code	Stack Info	Var Names	Var Values
01 def main() : 02 print(fib(4)) 03 04 def fib(n) : 05 if n <= 2 : 06 return 1 07 else : 08 return fib(n - 1) \ 09 + fib(n - 2) 10 11 main()			

Output (One symbol per cell)																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

6) An example showing where the recursive call is not last.

Code	Stack Info	Var Names	Var Values
01 def main() : 02 count(2) 03 print() 04 05 def count(limit): 06 if limit > 0: 07 count(limit - 1) 08 print(limit, end = " ") 09 10 main()			

Output (One symbol per cell)																			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

H) Storing Functions into a Module – Simple Illustration

MyMathFunctions.py

```
def factorial(facLimit):
  if (facLimit <= 1):
    return 1
  else:
    return facLimit * \
      factorial(facLimit - 1)

def summation(upper):
  if (upper == 1):
    return 1;
  else:
    return upper + \
      summation(upper - 1)
```

TestProgram.py

```
import MyMathFunctions

print(MyMathFunctions.factorial(5))
print(MyMathFunctions.factorial(6))
```

Output

```
120
720
```

A) List Introduction

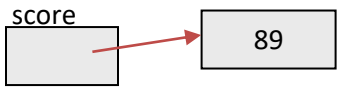
1) Rationale for using a list

Suppose your client wants to input an unknown number of names (not in alphabetic order) into a cell phone and 2) display those names in alphabetic order. Can you easily accomplish this with what we currently know?

2) Visualizing lists

a) Recall how single variables work

```
score = 89
```

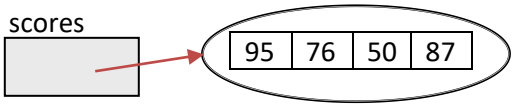


score refers to an object containing 89

(Reminder: to simplify the notes, we have been treating a value such as 89 stored inside score.)

b) How do lists work?

```
scores = [95, 76, 50, 87]
```



scores refers to a new list object that has 4 elements.

c) Summary

- A list is a container holding a changeable sequence of values.
- List variable names are typically plural because they often hold more than one value.
- Generally, each list has values all of the same type (all floating-point numbers, all strings, etc.)
- A list is a mutable type in Python – meaning that values inside an existing list can be changed instead of a new list object being built for each change that occurs.

3) List defintion (creation)

a) Syntax/Semantics

Syntax	Semantics
listVar = [] --or— listVar = list()	Defines (Creates) a new empty list that is accessed by listVar
listVar = [val_1, val_2, … val_n]	Defines (Creates) a new list, accessed by listVar, that is initialized with n elements. Each element may be a literal value or even a larger expression.

b) Examples

(i) Define a list of strings named schools initialized to Kansas, Colorado, and Iowa State.

(ii) Define an empty list that will eventually hold batting averages

4) Accessing and modifying existing list elements

a) How to access individual elements

(i) Include the list name and an index (position) inside brackets → listVar[i]

(ii) The index must evaluate to an integer value (could be a larger expression such as i + 1 inside brackets).

b) Examples

scores = [0, 0, 0, 0, 0]

scores[0]	0	**scores[-5]**	first element
scores[1]	0	**scores[-4]**	second element
scores[2]	0	**scores[-3]**	third element
scores[3]	0	**scores[-2]**	fourth element
scores[4]	0	**scores[-1]**	fifth element

Each integer inside the [ ] is called an index

The negative indexes range from -1 down through –(len(theList))

- Set the first list element to 4.
- Write a line of code to change the element at index 3 to 5.
- Display the second list element.
- Decrease the value of the fifth element by 1.
- Change the value stored at index 2 to the value stored at index 3.

Warning: The indexes which are 0 or positive are always one less than the list element indicated. Given the scores list above, what happens if the following code were executed?

```
scores[5] = 10
```

Python would raise a **list index out of range** exception, and the program would stop.

5) Basic list methods

Methods	Semantics
listVar.append(value)	Appends value to the end of the list
listVar.insert(i, value)	Inserts value into the list at index i. The original list values beginning at index i are shifted right (down) to make room.
listVar.pop(i)	Removes and returns the value at index i in the list.
listVar.pop()	Removes and returns the last value in the list.
listVar.remove(value)	Removes value from the list and shifts all elements following it left (up) one position.
listVar.index(value)	Returns the index of value in the list. **An error results if value is not in the list.**
listVar.count(value)	Returns the number of occurrences of value in the list
listVar.sort()	Sorts the values in the list from least to greatest
listVar. reverse()	Sorts the values in the list from greatest to list

Note: The **del** statement is recommended instead of pop if not needing the value that was deleted. (e.g., del myList[2] instead of myList.pop(2))

Code	Output shown across from each print
01 schools = ["Kansas"]	
02 print(schools)	['Kansas']
03 schools.append("JCCC")	
04 print(schools)	['Kansas', 'JCCC']
05 schools.append("Kansas State")	
06 print(schools)	['Kansas', 'JCCC', 'Kansas State']
07 schools.insert(0, "Missouri")	
08 print(schools)	['Missouri', 'Kansas', 'JCCC', 'Kansas State']
09 schools.pop(0)	
10 print(schools)	['Kansas', 'JCCC', 'Kansas State']
11 schools.pop()	
12 print(schools)	['Kansas', 'JCCC']
13 schools.insert(1, "UMKC")	
14 print(schools)	['Kansas', 'UMKC', 'JCCC']
15 schools.sort()	
16 print(schools)	['JCCC', 'Kansas', 'UMKC']
17 index = schools.index("UMKC")	
18 print(index)	2
19 schools.append("JCCC")	

```
20 print(schools)                          ['JCCC', 'Kansas', 'UMKC', 'JCCC']
21 print(schools.count("JCCC"))            2
22 del schools[0]
23 print(schools)                          ['Kansas', 'UMKC', 'JCCC']
24 schools.reverse()
25 print(schools)                          ['JCCC', 'UMKC', 'Kansas']
26 while (len(schools) > 0):
27     print(schools.pop(), end = " ")     Kansas UMKC JCCC
28 print()
29 print(schools)                          []
```

6) List values may be split into separate lines to avoid long lines as needed. The line continuation operator is not needed because the values are inside [ ]

Code (line wrap and/or supassing coding style line length)

```
01 scores = [88, 100, 90, 78, 85, 95, 90, 85, 92, 67, 83, 53, 100, 45,
02 75, 94, 99, 88, 71, 68, 34, 92, 91]
```

Code (fixed)

```
01 scores = [88, 100, 90, 78, 85, 95, 90, 85, 92,
02           67, 83, 53, 100, 45, 75, 94, 99, 88,
03           71, 68, 34, 92, 91]
```

7) Some Other functions and operations that work on sequence types such as Lists

dailyLows = [40, 42, 42, 51, 39]
dailyHighs = [55, 58, 61, 58, 47]

Function / Operation	Result
min(dailyLows)	39
max(dailyLows)	51
sum(dailyLows)	214
len(dailyLows)	5
dailyLows + dailyHighs	[40, 42, 42, 51, 39, 55, 58, 61, 58, 47]
dailyLows * 2	[40, 42, 42, 51, 39, 40, 42, 42, 51, 39]
[-1] * 10	[-1, -1, -1, -1, -1, -1, -1, -1, -1, -1]
dailyHighs[1 : 3]	[58, 61]
dailyHighs[1 :]	[58, 61, 58, 47]
dailyHighs[: 3]	[55, 58, 61]
dailyHighs[: -1]	[55, 58, 61, 58]

B) List Processing Examples

1) Data access, display, and modification

Code

```
01 from random import randint
02 testScores = [92, 56, 72, 68, 86, 98, 46]
03
04 # Display all scores
05 for score in testScores:
06     print(score, end = " ")
07 print()
08
09 # Add 2 points extra credit
10 for i in range(len(testScores)):
11     testScores[i] += 2
12
13 # Display all scores
```

92	56	72	68	86	98	46

```
14 print(testScores)
15
16 # Create a list of 5 random integers
17 randomData = []
18 NUM_LIST_VALUES = 5
19 LOW_VALUE = 1
20 HIGH_VALUE = 100
21 for i in range(NUM_LIST_VALUES):
22     randomData.append(randint(LOW_VALUE, HIGH_VALUE))
23
24 # Display the list values
25 for i in range(len(randomData)):
26     print(randomData[i], end= " ")
27 print()
28
29 # Create a list and partially fill it.
30 partialList = [0] * NUM_LIST_VALUES
31 NUM_VALUES = 3
32 for i in range(NUM_VALUES):
33     partialList[i] = randint(LOW_VALUE, HIGH_VALUE)
34
35 # Show the list
36 print(partialList)
37
38 # Show only the desired values
39 for i in range(NUM_VALUES):
40     print(partialList[i], end = " ")
41 print()
```

Output

```
92 56 72 68 86 98 46
[94, 58, 74, 70, 88, 100, 48]
53 12 77 14 48
[16, 8, 37, 0, 0]
16 8 37
```

2) Solve the initial problem of entering names and then displaying in alphabetic order

Code

```
01 names = []
02
03 # Get the names
04 name = input("Enter name or press only the enter key to quit: ")
05 while name != "" :
06     names.append(name)
07     name = input("Enter name or press only the enter key to quit: ")
08
09 # Sort the names
10 names.sort()
11
12 # Display the names
13 for name in names :
14     print(name, end = " ")
15 print()
```

Output (user input shown in **red**)

```
Enter name or press only the enter key to quit: Jack
Enter name or press only the enter key to quit: Adam
Enter name or press only the enter key to quit: Jocelyn
Enter name or press only the enter key to quit: Davonte
Enter name or press only the enter key to quit: Jasmine
```

```
Enter name or press only the enter key to quit:
Adam Davonte Jack Jasmine Jocelyn
```

FYI: If not wanting to change the original list order, but only sort the output:

Code

```
01 names = ["Jack", "Lacy", "Desmond", "Adam"]
02
03 # Display the names
04 for name in sorted(names) :
05     print(name, end = " ")
06 print()
07 print(names)
```

Output (user input shown in **red**)

```
Adam Desmond Jack Lacy
['Jack', 'Lacy', 'Desmond', 'Adam']
```

C) Common List Algorithms

1) Fill a list –See prior examples where the append method was used. The insert method may be used too.
2) Add all numbers in a list / concatenate all strings in a list with delimiters (list traversals)

Code

```
01 prices = [19.99, 5.99, 49.99, 35]
02 items = ["Basketball", "Baseball", "Tent", "Fishing Pole"]
03
04 # Find the total of all the prices
05 total = 0
06 for price in prices :
07   total = total + price
08 print(f"Total: ${total:.2f}")
09
10 # Create one string of items separated by commas
11 if (len(items) > 0) :
12   itemNames = items[0] # Initialize to the first item
13
14   # Tack on the rest with comma delimiters
15   for i in range(1, len(items)) :
16     itemNames += f", {items[i]}"
17
18   # Display comma-separated string
19   print(itemNames)
20
21 # Create one string of items separated by commas
22 itemNames = items[0] if len(items) > 0 else ""
23 for i in range(1, len(items)) :
24     itemNames += f", {items[i]}"
25 print(itemNames)
```

Can simplify lines 11 – 19 if displaying the empty string is permitted

Output

```
Total: $110.97
Basketball, Baseball, Tent, Fishing Pole
Basketball, Baseball, Tent, Fishing Pole
```

Note: The textbook would write lines 21 – 25 above similar to the code below. This also works, but the code is less efficient because the if condition is checked each loop iteration:

Code

```
21 # Create one string of items separated by commas
22 itemNames = ""
```

```
for i in range(len(items)) :
   if i > 0 :
      itemNames = itemNames + ", "
   itemNames = itemNames + items[i]
print(itemNames)
```

3) List searching
   a) Linear search
      (i) Could potentially do this:

Code

```
items = ["Basketball", "Baseball", "Tent", "Fishing Pole"]

# Where is the Tent?
index = items.index("Tent")
print(index)
```

Why might this not always be the best? Hint: What happens if a value isn't found?

      (ii) Could do this:

Code

```
itemToFind = "Tent"

# Search for the item
i = 0
found = False
while i < len(items) and not found :
  if items[i] == itemToFind :
    found = True
  else :
    i += 1

if (found) :
  print("{itemToFind} found at index {i}")
else :
  print("Item not found")
```

Output

```
Tent found at index 2
```

Suppose itemToFind contains "Tents"

```
Item not found.
```

Turning into a **function** and simplifying the code with two returns

Code

```
def main() :
  items = ["Basketball", "Baseball", "Tent", "Fishing Pole"]
  item = input("Enter an item to find: ")
  print(f"The index is {linearSearch(items, item)}")

# linearSearch searches for a given value in a list
# @param theList The list to search
# @param target The value to match
# @return The index of value if found or -1 if not found.
def linearSearch(theList, target) :
  for i in range(len(theList)) :
    if theList[i] == target :
      return i
  return -1
```

Be careful: -1 is a valid index. However, in this case we using it to represent an unfound value.

```
15
16 main()
```

Output

```
Enter an item to find: Tent
The index is 2.
```

Output

```
Enter an item to find: Tents
The index is -1.
```

FYI: A modified function below also works, but is less efficienct – there are likely two loops that run behind the scenes. One for **in**, and one for **index.** This would also not work in situations such as searching for a number > target or searching for an object whose attribute matches the target.

Code

```
01 # linearSearch searches for a given value in a list
02 # @param theList The list to search
03 # @param target The value to match
04 # @return The index of value if found or -1 if not found.
05 def linearSearch(theList, target) :
06   if target in theList:
07       return theList.index(target)
08   return -1
```

A modified linear search function using try/catch that works with a perfect size (completely full) list:

Code

```
01 # linearSearch searches for a given value in a list
02 # @param theList The list to search
03 # @param value The value to match
04 # @return The index of value if found or -1 if not found.
05 def linearSearch(theList, value) :
06   try:
07     index = theList.index(value)
08   except:
09     index = -1
10   return index
```

Using an optional parameter to get LinearSearch to work with oversize (partially filled) and perfect size (completely full) lists. A partially filled list could be a list that is treated as fixed size where a default value such as 0 could be stored to represent an open slot. This is more common with other languages that use arrays.

Code

```
01 # linearSearch searches for a given value in a list
02 # @param theList The list to search
03 # @param value The value to match
04 # @param numValues The number of values to search. If not
05 #        specified, the entire list is searched.
06 # @return The index of value if found or -1 if not found.
07 def linearSearch(theList, value, numValues = -1) :
08   # Search the entire list?
09   if numValues == -1:
10     numValues = len(theList)
11
12   for i in range(numValues) :
13     if theList[i] == value :
14       return i
15   return -1
```

b) Binary search – a faster search

What must be true of the data for binary search to properly work (precondition) ?

Code

```
01 def main():
02     data = [-40, -10, -2, 5, 26, 42, 72]
03     print("Index of -10: "
04           f"{binarySearch(data, -10)}")
05     print("Index of 43: "
06           f"{binarySearch(data, 43)}")
07
08 def binarySearch(data, key):
09     low = 0
10     high = len(data) - 1
11
12     while (high >= low):
13         mid = (low + high) // 2
14         if (key > data[mid]):
15             low = mid + 1
16         elif (key < data[mid]):
17             high = mid - 1
18         else:
19             return mid # key found
20
21     # key's insertion index would be
22     # abs(return value from binary search) - 1
23     return -low - 1
24
25 main()
```

Output

```
Index of -10: 1
Index of 43: -7
```

-40	-10	-2	5	26	42	72
0	1	2	3	4	5	6

key: **-10**

low:

high:

mid:

return:

key: **43**

low:

high:

mid:

return:

c) A recursive binary search version

Code	Stack Info	Var Names	Var Values

```
01 def main():
02   data = [-40, -10, -2, 5, 26, 42, 72]
03   print("Index of 26: "
04       f"{binarySearch(data, 0, len(data) - 1, 26)}")
05
06 def binarySearch(data, low, high, key) :
07   if low <= high :
08     mid = (low + high) // 2
09
10     if data[mid] == key :
11       return mid
12     elif data[mid] < key :
13       return binarySearch(data, mid + 1, high, key)
14     else :
15       return binarySearch(data, low, mid - 1, key)
16
17   else :
18     return -low - 1
19 main()
```

Suppose the list object referred to by data is stored at heap address 0x6B44

Output

```
Index of 26: 4
```

d) Simple search analysis – linear vs binary search

In linear search, given an array of n elements, _____ elements are check in the worst case.

In binary search, given an array of n elements, about **$\log_2 n + 1$** elements are checked in the worst case

$\text{Log}_2 n$ → 2 to the what power gives n?

# elements	Number of elements checked in worst case by **Linear Search**	Number of elements checked in worst case by **Binary Search**
4		
8		
16		
32		
64		
1024		

4) Swap values in a list

a) Example 1: works in all languages

Code

```
01 items = ["Baseball", "Basketball", "Tent", "Fishing Pole"]
02
03 # Swap the tent with the fishing pole
04 tmp = items[2]                          tmp =
05 items[2] = items[3]                     items[2] =
06 items[3] = tmp                          items[3] =
07
08 # Display the result
09 print(items)
```

Output

```
['Baseball', 'Basketball', 'Fishing Pole', 'Tent']
```

b) Example 2: Python shortcut

Code

```
01 items = ["Baseball", "Basketball", "Tent", "Fishing Pole"]
02
03 # Swap the tent with the fishing pole
04 items[2], items[3] = items[3], items[2]
05
06 # Display the result
07 print(items)
```

Output

```
['Baseball', 'Basketball', 'Fishing Pole', 'Tent']
```

5) Insertion and deletion without modifying the list size

a) Deletion (Left-shift)

Code

```
01 testScores = [92, 56, 46, 67, 86, 98, 72]
02 numScores = len(testScores)
03
04 # delete item at index 2
05 deletionIndex = 2;
06 for i in range(2, numScores - 1):
07   testScores[i] = testScores[i + 1];
08
09 numScores -= 1
```

92	56	46	67	86	98	72

b) Insertion (right-shift)

Code

```
01 testScores = [ 56, 76, 81, 92, 0, 0, 0 ]
02 numScores = 4
03
04 # Insert a score and maintain sorted order
05 newScore = 80
06 insertionIndex = 2
07
08 # Make room by shifting values right
09 # Assumption: there is room to insert a new number
10 for i in range(numScores, insertionIndex, -1):
11   testScores[i] = testScores[i - 1]
12
13 # Store the value
14 testScores[insertionIndex] = newScore
15 numScores += 1
```

56	76	81	92	0	0	0

D) List Parameters and Return Values

1) As a parameter to a method

Code

```
01 def main():
02     data = [0, 0, 0]
03
04     showList(data);
05     fillList(data);
06     showList(data);
07
08 def fillList(theList):
09     for i in range(len(theList)):
10         theList[i] = i + 1
11
12 def showList(theList):
13     for value in theList:
14         print(f"{value} ",
15               end = "")
16     print()
17
18 main()
```

Trace

Stack Info	Stk Var Names	Stk Var Values
Function Name		
Return Line #		
Return Value		

Python App

Heap

0x4AB2

0	
1	
2	

Output

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

Remember: All Python objects are stored on the heap, but for illustrative purposes we are showing values for strings, Booleans, and numbers stored directly in the local variable on the stack.

2) As a return value from a method

Code

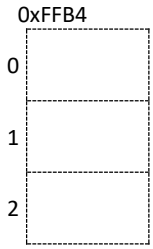
```
01 def main():
02   MAX_CAPITALS = 3
03   capitals = \
04     getCapitals(MAX_CAPITALS)
05
06   for capital in capitals:
07     print(capital, end = " ")
08   print()
09
10 def getCapitals(numCapitals):
11   print(f"Enter {numCapitals}" +\
12     " state capitals.")
13   capitals = []
14   for i in range(numCapitals):
15     capital = \
16       input(f"Capital {i + 1}: ")
17     capitals.append(capital)
18
19   return capitals
20
21 main()
```

Trace

Stack Info	Stk Var Names	Stk Var Values
Function Name		
Return Line #		
Return Value		

Python App

Heap



Output

```
Enter 3 state capitals.
Capital 1: Denver
Capital 2: Lincoln
Capital 3: Jackson
Denver Lincoln Jackson
```

Another way to write getCapitals:

Code

```
10 def getCapitals(numCapitals):
11   print(f"Enter {numCapitals} state capitals.")
12   capitals = [""] * numCapitals
13   for i in range(numCapitals):
14     capitals[i] = input(f"Capital {i + 1}: ")
15
16   return capitals
```

E) Elementary List Sorting Algorithms

1) Selection Sort (selecting the smallest)

a) Visualization (URL subject to change)

The smallest remaining value is selected each loop iteration and swapped.
http://liveexample.pearsoncmg.com/liang/animation/web/SelectionSort.html
(There are other variations where the largest is selected.)

b) Practice

Note: “Pass” refers to the state of the array after an iteration of the outer loop has completed.

Pass	8 4 5 1 3

Pass	5 4 3 1 9 2

c) Code

Selection Sort Code

```
def main():
    testScores = [83, 45, 65, 75, 86, 92, 68]
    selectionSort(testScores)
    print(testScores)

def selectionSort(theList) :
    for i in range(len(theList) - 1) :
        indexOfSmallest = i
        # Find the index of the smallest remaining items
        for j in range(indexOfSmallest + 1, len(theList)):
            if (theList[j] < theList[indexOfSmallest]):
                indexOfSmallest = j
        # Swap data if smaller item is found
        if (indexOfSmallest != i):
            temp = theList[indexOfSmallest]
            theList[indexOfSmallest] = theList[i]
            theList[i] = temp

main()
```

Output

```
[45, 65, 68, 75, 83, 86, 92]
```

2) Insertion Sort
   a) Visualization (URL subject to change)
      Make a sorted list of size 2, then size 3, then size 4, … , then size n
      http://liveexample.pearsoncmg.com/liang/animation/web/InsertionSort.html

   b) Sorting an array manually

Initial Array	10 8 7 9 6
Pass 1	[8 10] 7 9 6 (sorted list of size 2)
Pass 2	[7 8 10] 9 6 (sorted list of size 3)
Pass 3	[7 8 9 10] 6 (sorted list of size 4)
Pass 4	[6 7 8 9 10] (sorted list of size 5)

Practice:

Initial Array	8 4 5 1 3
Pass 1	
Pass 2	
Pass 3	
Pass 4	

Initial Array	5 4 3 1 9 2
Pass 1	
Pass 2	
Pass 3	
Pass 4	
Pass 5	

c) Code

Insertion Sort Code

```
def main():
    testScores = [83, 45, 65, 75, 86, 92, 68]
    insertionSort(testScores)
    print(testScores)

def insertionSort(theList) :
    for i in range(1, len(theList)) :
        itemToMove = theList[i]

        j = i - 1 # Start just to the left
        while( (j >= 0) and (itemToMove < theList[j]) ):
            theList[j + 1] = theList[j]
            j -= 1

        # Place item into correct location
        theList[j + 1] = itemToMove
main()
```

3) Bubble Sort

a) Visualization (URL subject to change)

As the largest value falls to the bottom each loop iteration, the smaller values "bubble up".
http://liveexample.pearsoncmg.com/liang/animation/web/BubbleSort.html
(There are other variations where the largest value moves to the top each loop iteration.)

b) Practice

Initial Array	**10 8 7 9 6**
Pass 1	**8 10** 7 9 6 (compare 1st pair and switch if needed) 8 **7 10** 9 6 (compare next pair and switch if needed) 8 7 **9 10** 6 (compare next pair and switch if needed) 8 7 9 **6 10** (compare next pair and switch if needed) **8 7 9 6 [10]** (largest element bubbled to bottom)
Pass 2	**7 8** 9 6 [10] (compare 1st pair and switch if needed) 7 **8 9** 6 [10] (compare next pair and switch if needed) 7 8 **6 9** [10] (compare next pair and switch if needed) **7 8 6 [9 10]** (2nd largest now in place)
Pass 3	**7 8** 6 [9 10] (compare 1st pair and switch if needed) 7 **6 8** [9 10] (compare next pair and switch if needed) **7 6 [8 9 10]** (3rd largest now in place)
Pass 4	**6 7** [8 9 10] (compare 1st pair and switch if needed) **6 [7 8 9 10]** (4th largest now in place)
Sorted result	**[6 7 8 9 10]**

Try these:

Pass	8 4 5 1 3

Pass	5 4 3 1 9 2

c) Code

Bubble Sort Code

```
def main():
    testScores = [83, 45, 65, 75, 86, 92, 68]
    bubbleSort(testScores)
    print(testScores)

def bubbleSort(theList) :
    for i in range(1, len(theList)) :
        for j in range(0, len(theList) - i) :
            if (theList[j] > theList[j + 1]) :
                theList[j + 1], theList[j] = theList[j], theList[j + 1]
main()
```

Traditional Swap would be:

```
tmp = theList[j + 1]
theList[j + 1] = theList[j]
theList[j] = tmp
```

Bubble Sort Version 2 (Stops early if data is known to be sorted; i.e. no swaps)

```
def bubbleSort(theList) :
    sorted = False
    i = 1
    while ( i < len(theList) and not sorted ) :
        sorted = True # Assume sorted
        for j in range(0, len(theList) - i) :
            if (theList[j] > theList[j + 1]) :
                theList[j + 1], theList[j] = theList[j], theList[j + 1]
                sorted = False # swap occurred
        i += 1
```

F) Tuples – Immutable Lists

1) Comparison to a List

List	Tuple
• ordered*, index access • items may be changed • items may be added/removed • duplicates allowed	• ordered*, index access • items may **not** be changed • items may **not** be added/removed • duplicates allowed

* ordered does not imply sorted order. It only means that items can be accessed in order from the beginning to the end.

2) Examples

Code

```
01 jcccCoordinates = (38.922180, -94.732550)
02 print(f"JCCC latitude: {jcccCoordinates[0]}")
03 print(f"JCCC longitude: {jcccCoordinates[1]}")
```

Output

```
JCCC latitude: 38.92218
JCCC longitude: -94.73255
```

The surrounding parentheses in line 01 are optional. If brackets were used, then it would be a list.

Code

```
01 def main():
02     countDown = (3,2,1)
03     showTuple(countDown)
04     stats = getStats()
05     showTuple(stats)
06
07 def showTuple(newTuple):
08     for value in newTuple:
09         print(f"{value} ", end = "")
10     print()
11     for i in range(len(newTuple)):
12         print(f"{newTuple[i]} ", end = "")
13     print()
14
15 def getStats():
16     low = 1
17     median = 4
18     high = 7
19
20     return low, median, high # or return (low, median, high)
21
22 main()
```

Output

```
3 2 1
3 2 1
1 4 7
1 4 7
```

G) List Comprehensions

1) A shorter syntax that may be used if desiring to build a **new** list

2) Syntax:

newList = [*expression* for *item* in *iterable* if *condition* == True]

Note1: if *condition* == True is optional

Note2: A conditional expression may also be used:

newList = [expr1 if condition else expr2 for *item* in *iterable*]

3) Examples

Traditional Loop

```
01 PASSING_LIMIT = 59.5
02 testScores = [92, 56, 72, 68, 86, 98, 46]
03
04 passingScores = []
05 for score in testScores:
06     if score >= PASSING_LIMIT:
07         passingScores.append(score)
08
09 print(passingScores)
```

List Comprehension

```
01 PASSING_LIMIT = 59.5
02 testScores = [92, 56, 72, 68, 86, 98, 46]
03
04 passingScores = [score for score in testScores if score >= PASSING_LIMIT]
05 print(passingScores)
```

Output (Same for each example)

```
[92, 72, 68, 86, 98]
```

Traditional Loop

```
01 states = ["iowa", "north dakota", "utah", "florida", "new mexico"]
02
03 statesInTitleCase = []
04 for state in states:
05     statesInTitleCase.append(state.title())
06
07 print(statesInTitleCase)
```

List Comprehension

```
01 states = ["iowa", "north dakota", "utah", "florida", "new mexico"]
02
03 statesInTitleCase = [state.title() for state in states]
04 print(statesInTitleCase)
```

Output (Same for each example)

```
['Iowa', 'North Dakota', 'Utah', 'Florida', 'New Mexico']
```

Traditional Loop

```
01 LOW = -10
02 HIGH = 10
03
04 evens = []
05 for num in range(LOW, HIGH + 1):
06     if num % 2 == 0:
07         evens.append(num)
08
09 print(evens)
```

List Comprehension

```
01 LOW = -10
02 HIGH = 10
03
04 evens = [num for num in range(LOW, HIGH + 1) if num % 2 == 0]
05
06 print(evens)
```

Output (Same for each example)

```
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]
```

Traditional Loop

```
01 numbers = [92, -86, 72, -68, 86, -98, 46]
02 newNumbers = []
03 for number in numbers:
04     if number >= 0:
05         newNumbers.append(number)
06     else:
07         newNumbers.append(0)
08 print(newNumbers)
```

Traditional Loop using a Conditional Expression

```
01 numbers = [92, -86, 72, -68, 86, -98, 46]
02 newNumbers = []
03 for number in numbers:
04     newNumbers.append(number if number >= 0 else 0)
05 print(newNumbers)
```

List Comprehension

```
01 numbers = [92, -86, 72, -68, 86, -98, 46]
02 newNumbers = [number if number >= 0 else 0 for number in numbers]
03 print(newNumbers)
```

Output (Same for each example)

```
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]
```

4) Practice

a) Use a list comprehension to build a new list named **negatives** consisting of those numbers less than 0 from a list named **numbers**.

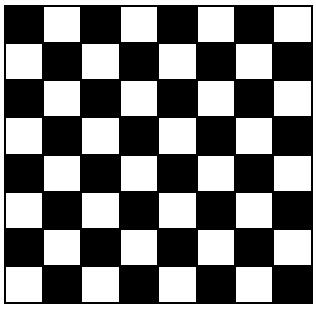
b) Use a list comprehension to build a new list named **bSchools** consisting of all the schools beginning with the letter B (or b) from a list named **schools**.

c) Use a list comprehension to build a new list named **numbers** that consists of every other number from 0 through 1000.

H) Two-dimensional Lists (Tables)

1) Examples of M x N tables (Each table row has the same length.)

Checkers



Tic Tac Toe

X	O	X
	O	

	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Figure 11 Figure Skating Medal Counts

2) Creation and Visualization

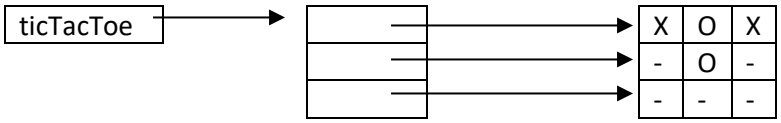
Code

```
01 ticTacToe = [
02     ["X","O","X"],
03     ["-","O","-"],
04     ["-","-","-"]
05 ]
06
07 print(ticTacToe)
```

Output

```
[['X', 'O', 'X'], ['-', 'O', '-'], ['-', '-', '-']]
```

A list where each element is also a list (A list of lists)



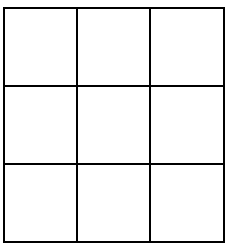
Can simply think of it like this:

	[0]	[1]	[2]
[0]	X	O	X
[1]	-	O	-
[2]	-	-	-

What is the value of **ticTacToe[0][1]**? What about **ticTacToe[1][0]**?

What would an empty board contain if the following two instructions were executed?

ticTacToe[2][1] = "X"
ticTacToe[1][0] = "O"



3) Formatting table output

Code

```
01 ROWS = 3
02 COLS = 3
03 ticTacToe = [
04     ["X","O","X"],
05     ["-","O","-"],
06     ["-","-","-"]
07 ]
08
09 for row in range(ROWS):
10     for col in range(COLS):
11         print(ticTacToe[row][col], end=" ")
12     print()
```

Output

```
X O X
- O -
- - -
```

Code

```
ticTacToe = [
    ["X","O","X"],
    ["-","O","-"],
    ["-","-","-"]
]

for row in range(len(ticTacToe)):
    for col in range(len(ticTacToe[row])):
        print(ticTacToe[row][col], end=" ")
    print()
```

What if we know all rows are the same length?

Output

```
X O X
- O -
- - -
```

4) Different ways to display tables with varying row sizes (also works if rows all the same size)

Code

```
01 calendar = [[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
02               19,20,21,22,23,24,25,26,27,28,29,30,31],
03              [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
04               19,20,21,22,23,24,25,26,27,28],
05              [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,
06               19,20,21,22,23,24,25,26,27,28,29,30,31]
07              ]
08
09 for month in calendar:
10     print(month)
11
12 print()
13 for month in calendar:
14     for day in month:
15         print(day, end = " ")
16     print()
17
18 print()
19 for month in calendar:
20     for i in range(len(month)):
21         print(month[i], end = " ")
22     print()
23
24 print()
25 for row  in range(len(calendar)):
26     for col in range(len(calendar[row])):
27         print(calendar[row][col], end = " ")
28     print()
```

Three months (rows) stored

Display each row as a list

Can pull out each row and then use an index to access each value in the row. This index could also be used to change a row value if desired. (Recall that indexes are needed to modify existing values)

Output

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
```

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

5) Creating larger tables

25x100 table initialized to 0's

```
01 ROWS = 25
02 COLS = 100
03
04 matrix = []
05 for row in range(ROWS):
06     nextRow = [0] * COLS
07     matrix.append(nextRow)
```

50x50 table initialized to empty strings

```
ROWS = 50
COLS = 50

table = []
for row in range(ROWS):
    nextRow = [""] * COLS
    table.append(nextRow)
```

6) Sending tables to and returning tables from functions

Code

```
01 from random import randint
02
03 def main():
04     ROWS = 4
05     COLS = 5
06
07     matrix = getTable(ROWS, COLS)
08     displayMatrix(matrix)
09     print()
10     fillMatrix(matrix)
11     displayMatrix(matrix)
12
13 # getTable creates a rows x cols table initialized to 0's
14 # @param rows The number of rows
15 # @param cols The number of columns
16 # @return the rectangular table
17 def getTable(rows, cols):
18     table = []
19     for row in range(rows):
20         nextRow = [0] * cols
21         table.append(nextRow)
22     return table
23
24 # displayMatrix displays a matrix in table format with a cell width of 4
25 # @param matrix The matrix to display
26 def displayMatrix(matrix):
27     for row in range(len(matrix)):
28         for col in range(len(matrix[row])):
29             print(f"{matrix[row][col]:4d}", end = "")
30         print()
31
32 # fillMatrix fills a matrix with random integers in the range -10 through 10
33 # @param matrix The matrix to fill
34 def fillMatrix(matrix):
35     LOW = -10
36     HIGH = 10
37     for row in range(len(matrix)):
38         for col in range(len(matrix[row])):
39             matrix[row][col] = randint(LOW, HIGH)
40
41 main()
```

Also works:

```
for rowList in matrix:
  for col in range(len(rowList)):
     rowList[col] = randint(LOW, HIGH)
```

Output

```
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0
    0    0    0    0    0

    9   10    7    2    0
   -1   -1   -7  -10    7
   -4    0    1    0    9
    0    5    3    9    3
```

7) Slightly modified example from Python for Everyone by Horstmann and Necaise

Code

```
01 #  This program prints a table of medal winner counts with row totals.
02 #
03
04 MEDALS = 3
05 COUNTRIES = 8
06
07 # Create a list of country names.
08 countries = [ "Canada",
09              "Italy",
10              "Germany",
11              "Japan",
12              "Kazakhstan",
13              "Russia",
14              "South Korea",
15              "United States" ]
16
17 # Create a table of medal counts.
18 counts = [
19            [ 0, 3, 0 ],
20            [ 0, 0, 1 ],
21            [ 0, 0, 1 ],
22            [ 1, 0, 0 ],
23            [ 0, 0, 1 ],
24            [ 3, 1, 1 ],
25            [ 0, 1, 0 ],
26            [ 1, 0, 1 ]
27          ]
28
29 # Print the table header.
30 print(f'{"Country":>15s}{"Gold":>8s}{"Silver":>8s}{"Bronze":>8s}{"Total":>8s}')
31
32 # Print countries, counts, and row totals.
33 for i in range(COUNTRIES) :
34    print(f"{countries[i]:>15s}", end = "")
35
36    # Print each row element and update the row total.
37    total = 0
38    for j in range(MEDALS) :
39       print(f"{counts[i][j]:>8d}", end = "")
40       total = total + counts[i][j]
41
42    # Display the row total and print a new line.
43    print(f"{total:>8d}")
```

Output

```
        Country    Gold  Silver  Bronze   Total
```

```
          Canada      0       3       0       3
           Italy      0       0       1       1
         Germany      0       0       1       1
           Japan      1       0       0       1
      Kazakhstan      0       0       1       1
          Russia      3       1       1       5
     South Korea      0       1       0       1
   United States      1       0       1       2
```

8) FYI

a) Lists may be more than two dimensions

Three Dimensional List

```
01 threeDim = [
02     [[1,2],[3,4],[5,6]],
03     [[7,8],[9,10],[11,12]],
04     [[13,14],[15,16],[17,18]]
05 ]
06
07 print(threeDim[2][1][0])
```

Output

```
15
```

b) Tuples may also be multi-dimensional

A) Beginning File Processing

1) File open syntax:

fileObject = open(*filename* [, mode])

where:

The bracket notation [] signifies that syntax contained inside is optional. It is not Python syntax (i.e. a list) in this context.

- *filename* is a string representing the name of the file to open. The file path could also be part of the string.
- *mode* is a string indicating the type of file processing. A subset of modes include:

mode	description
"r"	Default. Opens filename for reading. Results in an error if filename does not exist.
"w"	Opens filename for writing. Creates filename if it does not exist.
"a"	Opens filename for appending. Creates filename if it does not exist.
The following can be appended to "r", "w", or "a" above (e.g., "rt", "wb")	
"t"	Default. Opens filename in text mode.
"b"	Opens filename in binary mode.

If open is only called with a filename (no mode), the default mode is "rt".

"r+" opens a text file for updating (reading and writing)

2) Reading text from a file

By default, the **open** function looks for Cities.txt in the same location as the source code file that accesses it.

a) Line by line

Code

```
01 inFile = open("Cities.txt", "r")
02 for line in inFile:
03     print(line)
04 inFile.close()
05
06
```

```
inFile = open("Cities.txt", "r")
line = inFile.readline()
while line != "":
    print(line)
    line = inFile.readline()
inFile.close()
```

Cities.txt

```
Overland Park
Olathe
Lenexa
Lees Summit
```

Output

```
Overland Park

Olathe

Lenexa

Lees Summit
```

Note: Why the blank lines?

Removing the blank lines – Example 1

```
01 inFile = open("Cities.txt", "r")
02 for line in inFile:
03     print(line.rstrip())
04 inFile.close()
05
```

Output

```
Overland Park
Olathe
Lenexa
Lees Summit
```

Removing the blank lines – Example 2

```
inFile = open("Cities.txt", "r")
for line in inFile:
    line = line.rstrip()
    print(line)
inFile.close()
```

Output

```
Overland Park
Olathe
Lenexa
Lees Summit
```

b) If the file is small, it could be read all at once

Code using read

```
01 #Get the file name
02 filename = input("Which file to display? ")
03
```

```
04 # Read the data
05 inFile = open(filename, "r")
06 data = inFile.read()
07 inFile.close()
08
09 # Display the data
10 print(data)
```

Output (user input in **red**)

```
Which file to display? Cities.txt
Overland Park
Olathe
Lenexa
Lees Summit
```

Note: read() returns one string for the entire file

data → `"Overland Park\nOlathe\nLenexa\nLees Summit"`

Similar code using readlines

```
01 #Get the file name
02 filename = input("Which file to display? ")
03
04 # Read the data
05 inFile = open(filename, "r")
06 data = inFile.readlines()
07 inFile.close()
08
09 # Display the data
10 print(data)
11 print()
12 for city in data:
13     print(city.rstrip())
```

Output (user input in **red**)

```
Which file to display? Cities.txt
['Overland Park\n', 'Olathe\n', 'Lenexa\n', 'Lees Summit']

Overland Park
Olathe
Lenexa
Lees Summit
```

Note: readlines() returns a list. Each list element contains a string representing a line

data →

Overland Park\n	Olathe\n	Lees Summit\n	Lenexa

What is another way to write line 13 using the **end** named parameter?

3) Reading numeric data

a) Line by line

Code

```
01 inFile = open("Numbers.txt", "r")
02
03 # Sum the integers on each line in the file
04 sum = 0
05 for line in inFile:
06     sum += int(line)
07
08 # Display the sum
09 print(f"Sum: {sum}")
10 inFile.close()
```

Since a mathematical calculation is performed, the number must first be converted from text (e.g. "3") into an integer (e.g. 3). **int(line.rstrip())** could have been used in line 6, but **int** ignores the end of line marker.

Output

```
Sum: 1
```

Numbers.txt

```
3
9
-2
-8
-4
3
```

b) Getting the file name from the user and modularizing the prior code

Code

```
01 def main():
02     inFile = open(getFileName(), "r")
03     print(f"Sum: {getSum(inFile)}")
04     inFile.close()
05
06 # getFileName retrieves a filename from the user
07 # @return the filename
08 def getFileName():
09     filename = input("Input filename? ")
10     return filename
11
12 # getSum sums the integers on each line in the file
13 # @param inFile File object already opened for reading
14 # @return The sum
15 def getSum(inFile):
16     sum = 0
17     for line in inFile:
18         sum += int(line)
19     return sum
20
21 main()
```

Output (user input in **red**)

```
Input filename? Numbers.txt
Sum: 1
```

c) If the file is small, another way to write the prior code using **readlines**

Code

```
01 def main():
02     inFile = open(getFileName(), "r")
03     lineList = inFile.readlines()
04     inFile.close()
05     print(f"Sum: {getSum(lineList)}")
06
07 # getFileName retrieves a filename from the user
08 # @return the filename
09 def getFileName():
10     fileName = input("Input filename? ")
11     return fileName
12
13 # getSum sums the integers from a list
14 # @param numbers A list of numbers as strings
15 # @return The sum
16 def getSum(numbers):
17     sum = 0
18     for number in numbers:
19         sum += int(number)
20     return sum
21
22 main()
```

Output (user input in **red**)

```
Input filename? Numbers.txt
Sum: 1
```

d) If with is used, file closing occurs automatically when the with block finishes execution

Code

```
01 def main():
02     with open(getFileName(), "r") as inFile:
03         lineList = inFile.readlines()
04     print(f"Sum: {getSum(lineList)}")
05
06 # rest of code omitted
```

inFile is automatically closed, so no need for inFile.close()

Output (user input in **red**)

```
Input filename? Numbers.txt
Sum: 1
```

e) Suppose the numbers are scattered on numerous lines but are all separated by white space?

Code

```
01 inFile = open("Numbers2.txt", "r")
02
03 # Sum the integers on each line in the file
04 sum = 0
05 for line in inFile:
06     numbers = line.split()
07     print(numbers)
08
09     # Add the numbers
10     for number in numbers:
11         sum += int(number)
12
13 # Display the sum
14 print(f"Sum: {sum}")
15 inFile.close()
```

Numbers2.txt

```
1  12   3
4
-4  3
```

split() returns a list of elements that were delimited by the white space characters.

Line 07 is only inserted for instructional purposes -- so each created list can be seen.

Output

```
['1', '12', '3']
['4']
['-4', '3']
Sum: 19
```

If the file is small and because new line characters are also whitespace….

```
01 # Read all tokens (numbers) into a list
02 with open("Numbers2.txt", "r") as inFile:
03     numbers = inFile.read().split()
04 print(numbers)
05
06 # Convert each number in the list and add to the sum
07 sum = 0
08 for number in numbers:
09     sum += int(number)
10
11 # Display the sum
12 print(f"Sum: {sum}")
```

Reads the file as one string, and then split returns a list of the white space separated values.

Line04 is only inserted for instructional purposes.

Output

```
['1', '12', '3', '4', '-4', '3']
Sum: 19
```

4) Writing text to a file

a) Example – line by line

Code

```
01 inFile = open("Cities.txt", "r")
02 outFile = open("CitiesCopy.txt", "w")
03
04 # Copy each line to new file
05 for line in inFile:
06     outFile.write(line)
07
08 inFile.close()
09 outFile.close()
10
11 print("File copy completed.")
```

Output

```
File copy completed.
```

Cities.txt

```
Overland Park
Olathe
Lenexa
Lees Summit
```

Newly created CitiesCopy.txt

```
Overland Park
Olathe
Lenexa
Lees Summit
```

Note: **write** does not output an end of line marker like print

b) Example – as a whole (Assume the file is relatively small)

Code (using **with**)

```
01 # Read the data
02 with open("Cities.txt", "r") as inFile:
03   data = inFile.read()
04
05 # Write the data
06 with open("CitiesCopy.txt", "w") as outFile:
07   outFile.write(data)
08
09 print("File copy completed.")
```

Output

```
File copy completed.
```

Code (**not** using **with**)

```
01 # Read the data
02 inFile = open("Cities.txt", "r")
03 data = inFile.read()
04 inFile.close()
05
06 # Write the data
07 outFile = open("CitiesCopy.txt", "w")
08 outFile.write(data)
09 outFile.close()
10
11 print("File copy completed.")
```

Output

```
File copy completed.
```

c) Similar example, except writing out the names to the same line in sorted order

Code

```
01 # Read the names
02 inFile = open("Cities.txt", "r")
03 lines = inFile.readlines()
04 inFile.close()
05
06 # Write the names in sorted order onto one line
07 outFile = open("SortedCities.txt", "w")
08 for line in sorted(lines):
09     outFile.write(f"{line.rstrip()} ")
10 outFile.close()
11
12 print("New file created.")
```

Output

```
New file created.
```

SortedCities.txt

```
Lees Summit Lenexa Olathe Overland Park
```

B) Beginning CSV (comma separated values) File Processing

1) Example 1

FYI: A .csv extension is not required but double-clicking a file with a .csv extension will launch Excel in MS Windows.

a) Reading a csv file using **split**

Code

```
01 with open("CityNames.csv", "r") as inFile:
02     # Get the names line by line
03     for line in inFile:
04         line = line.rstrip()
05         cities = line.split(",")
06
07         # Display the names on the line
08         for name in cities:
09             print(name)
```

CityNames.csv

```
Overland Park,Olathe,Lenexa,Shawnee,Mission
Roeland Park,Leawood,Gardner,De Soto
```

Output

```
Overland Park
Olathe
Lenexa
Shawnee
Mission
Roeland Park
Leawood
Gardner
De Soto
```

b) Reading a csv file using the **csv module**

Code

```
01 import csv
02
03 with open("CityNames.csv", "r") as inFile:
04     csvReader = csv.reader(inFile, delimiter = ',')
05
06     # Get the names line by line
07     for line in csvReader:
08         # Display the names on the line
09         for name in line:
10             print(name)
```

For each loop iteration, the csvReader will read the next line from the file and strip off the end of line marker. It assigns to line a list of cities for the line just read. **No need for split (or rstrip).**

Output

```
Overland Park
Olathe
Lenexa
```

```
Shawnee
Mission
Roeland Park
Leawood
Gardner
De Soto
```

2) Example 2

a) Reading and processing a csv file using **split**

Code

```
01 inFile = open("Grades.txt", "r")
02
03 # Bypass the file header (first line)
04 inFile.readline()
05
06 # Process each student
07 for line in inFile:
08     studentData = line.split(",")
09     print(studentData)
10
11     # Reset test score data
12     testScoreTotal = 0
13     numTestScores = len(studentData) - 1
14
15     # Total the student's test scores and display results
16     for i in range(1, numTestScores + 1):
17         testScoreTotal += float(studentData[i])
18     print(f"{studentData[0]} average: {testScoreTotal / numTestScores:.2f}")
19
20 inFile.close()
```

Grades.txt

```
ID,Test 1,Test 2
00300095,85.3,95
30412373,72,89.1
```

Line09 is only inserted for instructional purposes

Output

```
['00300095', '85.3', '95\n']
00300095 average: 90.15
['30412373', '72', '89.1']
30412373 average: 80.55
```

Note: Line04 could also be changed to **`next(inFile)`**

b) Reading and processing a csv file using the **csv module**

Code

```
01 import csv
02
03 inFile = open("Grades.txt", "r")
04 csvReader = csv.reader(inFile, delimiter = ',')
05
06 #Bypass the file header
07 next(csvReader)
08
09 # Process each student
10 for studentRow in csvReader:
11     print(studentRow)
12
13     # Reset test score data
14     testScoreTotal = 0
15     numTestScores = len(studentRow) - 1
16
17     # Total the student's test scores and display results
18     for i in range(1, numTestScores + 1):
19         testScoreTotal += float(studentRow[i])
```

Grades.txt

```
ID,Test 1,Test 2
00300095,85.3,95
30412373,72,89.1
```

Line 11 is only inserted for instructional purposes. studentRow is already a list – no need for split.

```
20      print(f"{studentRow[0]} average: {testScoreTotal / numTestScores:.2f}")
21
22  inFile.close()
```

Output

```
['00300095', '85.3', '95']
00300095 average: 90.15
['30412373', '72', '89.1']
30412373 average: 80.55
```

Note: No end of line marker after 95 like in the prior example. The csvReader removed it.

3) Example 3

a) Creating and writing data to a CSV file

```
01  MAX_HOLES = 9
02
03  # Get the golfer's name
04  name = input("Golfer name: ")
05
06  # Get the golfer's scores
07  scores = []
08  print(f"{name}'s scores for {MAX_HOLES} holes:")
09  for i in range(1, MAX_HOLES + 1):
10      scores.append(int(input(f"Hole {i}: ")))
11
12  # Store the golfer data
13  with open("GolfScores.csv", "w") as outFile:
14      #Create the header row
15      outFile.write(f"Name,1,2,3,4,5,6,7,8,9\n")
16      outFile.write(f"{name}")
17
18      # Write each score
19      for score in scores:
20          outFile.write(f",{score}")
21      outFile.write("\n")
22
23  print("Data written to GolfScores.csv")
```

Comma separators (delimiters) must be written to separate the data fields.

Output (user input in **red**)

```
Golfer name: Jasmine
Jasmine's scores for 9 holes:
Hole 1: 4
Hole 2: 6
Hole 3: 5
Hole 4: 3
Hole 5: 4
Hole 6: 7
Hole 7: 5
Hole 8: 6
Hole 9: 4
Data written to GolfScores.csv
```

GolfScores.csv

```
Name,1,2,3,4,5,6,7,8,9
Jasmine,4,6,5,3,4,7,5,6,4
```

b) Writing to a CSV file using the **csv module**

```
01  import csv
02  MAX_HOLES = 9
03
04  # Get the golfer's name
05  name = input("Golfer name: ")
06
07  # Get the golfer's scores
```

```
08 scores = []
09 print(f"{name}'s scores for {MAX_HOLES} holes:")
10 for i in range(1, MAX_HOLES + 1):
11     scores.append(int(input(f"Hole {i}: ")))
12
13 # Store the golfer data
14 with open("GolfScores.csv", "w", newline = "") as outFile:
15     csvWriter = csv.writer(outFile, delimiter = ',')
16
17     #Create the header row and store the data
18     csvWriter.writerow(["Name", "1", "2", "3", "4",
19                         "5", "6", "7", "8", "9"])
20     csvWriter.writerow([name] + scores)
21
22 print("Data written to GolfScores.csv")
```

Need to send a list to **writerow**. List elements are automatically written with comma separators.

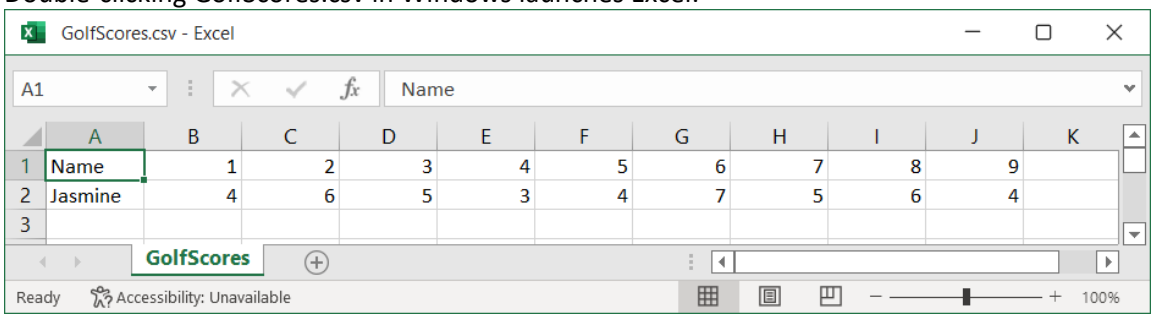
Output (user input in **red**)

```
Golfer name: Jasmine
Jasmine's scores for 9 holes:
Hole 1: 4
Hole 2: 6
Hole 3: 5
Hole 4: 3
Hole 5: 4
Hole 6: 7
Hole 7: 5
Hole 8: 6
Hole 9: 4
Data written to GolfScores.csv
```

GolfScores.csv

```
Name,1,2,3,4,5,6,7,8,9
Jasmine,4,6,5,3,4,7,5,6,4
```

Double-clicking GolfScores.csv in Windows launches Excel:



C) Exceptions

1) Terminology

Exception: An object that represents an error or a condition that prevents execution from proceeding normally. If an exception is not handled, the program will terminate abnormally.

2) Handling Exceptions with try-except

Syntax	Semantics

`try :`
`  statement(s)`
`except [ExceptionType1 [as identifier]] :`
`  statement(s)`
 	
`. . .`
 	
`except [ExceptionTypen [as identifier]] :`
`  statement(s)`
 	
`[finally :`
`  statement(s)]`
	Execute the statements in the try block. If a statement causes an exception to be raised, immediately transfer to the first except block. Proceed downward through each except until the exception thrown matches the exception type. If a match is found, execute the statements in that except block.
	
	The finally block is included if there is some code that must be executed last, whether or not an exception is thrown.
	
	Note: After code is executed in the catch and potentially a finally, program execution continues in a linear fashion after those blocks unless a statement such as a return or exit was executed. Control does not return to where the exception was initially raised in the try.

Note: The brackets above are notation meaning optional. They are not part of the Python syntax in this context

3) Examples

Halloween.py

```
01 numCandyBars = int(input("How many candy bars? "))
02 numKids = int(input("How many kids? "))
03 barsPerKid = numCandyBars / numKids
04 print(f"Each kid gets {barsPerKid:.1f} candy bars.")
```

Output (user input in **red**)

```
How many candy bars? 10
How many kids? 4
Each kid gets 2.5 candy bars.
```

Output (user input in **red**)

```
How many candy bars? 10
How many kids? 0
Traceback (most recent call last):
  File "C:\Users\vango\source\repos\PythonTesting\PythonTesting.py",
line 3, in <module>
    barsPerKid = numCandyBars / numKids
ZeroDivisionError: division by zero
```

HalloweenWithExceptionHandling.py

```
01 numCandyBars = int(input("How many candy bars? "))
02 numKids = int(input("How many kids? "))
03 try:
04     barsPerKid = numCandyBars / numKids
05     print(f"Each kid gets {barsPerKid:.1f} candy bars.")
06 except ZeroDivisionError as exc:
07     print(exc)
08 print("Continuing with program...")
```

Output (user input in **red**)

```
How many candy bars? 10
How many kids? 0
division by zero
Continuing with program...
```

SummationWithExceptionHandling.py

```
01 MAX_NUMBERS = 4
02
03 print(f"Enter {MAX_NUMBERS:d} integers to add.")
04 sum = 0
05 i = 0
06 while (i < MAX_NUMBERS):
07     try:
08         number = int(input(f"Integer {i + 1:d}: "))
09         sum += number
10         i += 1
11     except ValueError as exc:
12         print("Input ignored. Please enter a valid integer. ")
13 print(f"Sum: {sum}")
```

Because exc is not used, line 11 can instead be written as
`except:`
or
`except ValueError:`

Output (user input in **red**)

```
Enter 4 integers to add.
Integer 1: 2
Integer 2: e
Input ignored. Please enter a valid integer.
Integer 2: 4
Integer 3: -2
Integer 4: 3.4
Input ignored. Please enter a valid integer.
Integer 4: 3
Sum: 7
```

ExceptionPropogationExample.py

```
01 def main():
02     try:
03         print("First stmt in try")
04         m1()
05     except Exception as exc:
06         print("First stmt in except block")
07         print(exc)
08
09 def m1():
10     print("First stmt in m1")
11     m2()
12     print("Last stmt in m1")
13
14 def m2():
15     print("First stmt in m2")
16     m3()
17     print("Last stmt in m2")
18
19 def m3():
20     print("First stmt in m3")
21     raise Exception("Test message in exception")
22
23 main()
```

Output

```
First stmt in try
First stmt in m1
First stmt in m2
First stmt in m3
First stmt in except block
Test message in exception
```

InputFilesWithExceptions.py

```
01 def main():
02     filename = input("Which data file? ")
03     try:
04         inFile = open(filename, "r")
05         print(f"Sum: {getSum(inFile):d}")
06     except FileNotFoundError as exc:
07         print("Did you enter the correct filename?")
08         print(exc)
09     except ValueError as exc:
10         print("Input mismatch. Check the data file")
11     except Exception as exc:
12         print(exc)
13     finally:
14         if ("inFile" in locals()):
15             inFile.close()
16
17 # getSum adds all the integers in a file
18 # @param inFile An opened input file
19 # @return The sum of the integers
20 def getSum(inFile):
21     sum = 0
22     for line in inFile:
23         numbers = line.split()
24
25         # Add the numbers
26         for number in numbers:
27             sum += int(number)
28     return sum
29
30 main()
```

FYI: The textbook suggests to nest the try catch in main instead of using locals():

```
filename = input("Which data file? ")
try:
  inFile = open(filename, "r")
  try:
    print(f"Sum: {getSum(inFile):d}")
  except FileNotFoundError as exc:
    print("Did you enter the correct filename?")
    print(exc)
  except ValueError as exc:
    print("Input mismatch. Check the data file")
  except Exception as exc:
    print(exc)
finally:
  inFile.close()
```

Output (user input in **red**)
Integers.txt contains: **1 2 3 4 a 9 11**

```
Which data file? Integers.txt
Input mismatch. Check the data file
```

Output (user input in **red**)

```
Which data file? Ints.txt
Did you enter the correct filename?
[Errno 2] No such file or directory: 'Ints.txt'
```

Output (user input in **red**)
Integers.txt contains: **1 2 3 4 5 9 11**

```
Which data file? Integers.txt
Sum: 35
```

Output (user input in **red**)
Integers.txt contains: **1 2 3 4 5 9.3 11**

```
Which data file? Integers.txt
Input mismatch. Check the data file
```

Suppose
- Integers.txt contains: **1 2 3 4 5 9 11**
- Permission was denied to read the file.

Output (user input in **red**)

```
Which data file? Integers.txt
[Errno 13] Permission denied: 'Integers.txt'
```

4) Python Exception Hierarchy

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

Source → https://docs.python.org/3/library/exceptions.html

A) Sets

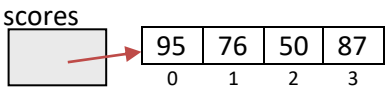
1) Introduction

a) A set is a collection of unique items (no duplicates) that are unordered and unchangeable. Unchangeable means that a set item cannot be directly changed to a different item; however items may be added and removed.

b) Visual including review of prior built-in Python containers (list, tuple)

(i) **List** named scores:

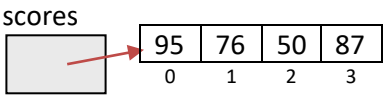
```
scores = [95, 76, 50, 87]
```



- ordered*, index access
- items may be changed
- items may be added/removed
- duplicates allowed

(ii) **Tuple** named scores:

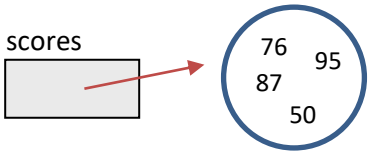
```
scores = (95, 76, 50, 87)
```



- ordered*, index access
- items may not be changed
- items may not be added/removed
- duplicates allowed

(iii) **Set** named scores:

```
scores = {95, 76, 50, 87}
```



- unordered, no index access
- items may not be changed
- items may be added/removed
- duplicates not allowed

* ordered does not imply sorted order. It only means that items can be accessed in order from the beginning to the end.

c) Uses:

(i) Quickly determine item membership in a collection (and order is not needed)

(ii) Prevent duplicate items in a collection.

(iii) Mimic mathematical set operations.

2) Examples

SetBasics.py

```
01 def main():
02     mySet = {5,-4,3,10,1}
03     showSet(mySet)
04
05     mySet.add(-3)
06     showSet(mySet)
07
08     mySet.discard(5)
09     showSet(mySet)
10
11     mySet.add(2)
12     showSet(mySet)
13
14 def showSet(theSet):
15     for item in theSet:
16         print(item, end = " ")
17     print()
18
19 main()
```

Because set items are unordered, a loop like this might not show the items in the original coded order. As stated above, utilizing an index to access an item is prohibited.
Note: A set could be sent to the **sorted** function, and it would return a list of items in sorted order.

Output (user input in **red**)

```
1 3 5 10 -4
1 3 5 10 -4 -3
1 3 10 -4 -3
1 2 3 10 -4 -3
```

WhoseTurn.py

```
01 def main():
02     names = getRoster("Roster.txt")
03
04     moreTurns = "Y"
05     while(moreTurns == "Y"):
06         name = input("Whose turn is it? ").title()
07         if (name in names):
08             print(f"{name}, your turn.")
09             names.discard(name)
10         else:
11             print(f"{name} does not exist or already took a turn.")
12
13         moreTurns = input("Another turn (Y or N)? ")[0].upper()
14
15 # getRoster reads a given file containing one comma-separated line
16 # of roster names
17 # @param filename The name of the file
18 # @return A set of names
19 def getRoster(filename):
20     import csv
21     with open(filename, "r") as inFile:
22         csvReader = csv.reader(inFile, delimiter = ',')
23         names = next(csvReader)
24     return set(names)
25
26 main()
```

Roster.txt

```
Jackson,Abby,Caelyn,Desmond,Ally,Andrew
```

Output (user input in **red**)

```
Whose turn is it? Jackson
Jackson, your turn.
Another turn (Y or N)? y
Whose turn is it? Caelyn
Caelyn, your turn.
Another turn (Y or N)? y
Whose turn is it? Jackson
Jackson does not exist or already took a turn.
Another turn (Y or N)? n
```

Python for Everyone textbook example -- UniqueWords.py

```
01 ##
02 #  This program counts the number of unique words contained in a text document.
03 def main() :
04    uniqueWords = set()
05
06    filename = input("Enter filename (default: NurseryRhyme.txt): ")
07    if len(filename) == 0 :
08       filename = "nurseryrhyme.txt"
09    inputFile = open(filename, "r")
10
11    for line in inputFile :
12       theWords = line.split()
13       for word in theWords :
14          cleaned = clean(word)
15          if cleaned != "" :
16             uniqueWords.add(cleaned)
17
18    print(f"The document contains {len(uniqueWords)}"
19          + " unique words.")
20
21 ## Cleans a string by making letters lowercase and
22 #  removing characters that are not letters.
23 #  @param string the string to be cleaned
24 #  @return the cleaned string
25 def clean(string) :
26    result = ""
27    for char in string :
28       if char.isalpha() :
29          result = result + char.lower()
30
31    return result
32
33 main()
```

NurseryRhyme.txt

```
Mary had a little lamb,
whose fleece was white as snow.

And everywhere that Mary went,
the lamb was sure to go.

It followed her to school one day
which was against the rules.

It made the children laugh and play,
to see a lamb at school.

And so the teacher turned it out,
but still it lingered near,

And waited patiently about,
till Mary did appear.

"Why does the lamb love Mary so?"
the eager children cry.

"Why, Mary loves the lamb, you know."
 the teacher did reply.
```

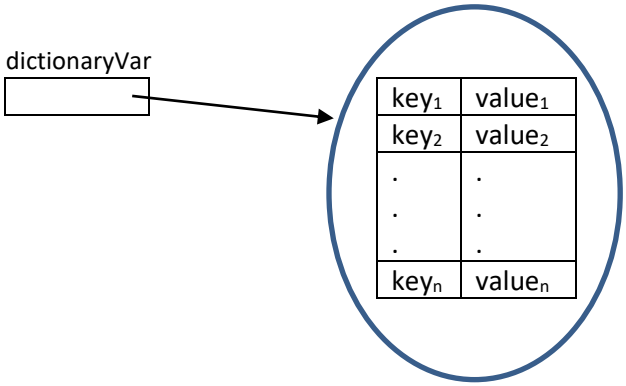
Output (The enter key was pressed after the prompt)

```
Enter filename (default: nurseryrhyme.txt):
The document contains 57 unique words.
```

B) Dictionaries – A Fourth Python built-in Container

1) What are dictionaries?

A **dictionary** associates _____ with _____.



- ordered (Python 3.7 and newer)
- key access
- items may be changed
- items may be added/removed
- duplicate keys not allowed

2) Dictionary creation

Syntax	Semantics
dictionaryVar = {} —or— dictionaryVar = dict()	Creates a new empty dictionary that is accessed by dictionaryVar
dictionaryVar = { key_1 : val_1, key_2 : val_2, . . ., key_n : val_n }	Creates a new dictionary, accessed by dictionaryVar, that is initialized with n key-value pairs

Note: Unlike list indexes, the keys are not required to be integers.

3) Dictionary Operations

Operations	Semantics
`dictionaryVar[key]`	Access the dictionary value associated with key. A KeyError occurs if key does not exist.
`dictionaryVar[key] = newVal`	If key exists in the dictionary, change its associated value to newVal; else add key to the dictionary with a value of newVal
`dictionaryVar.update({key:newVal})`	If key exists in the dictionary, change its associated value to newVal; else add key to the dictionary with a value of newVal. **Note**: the dictionary sent to update may consist of multiple key-value pairs.
`del dictionaryVar[key]`	Delete key and its associated value from the dictionary. A KeyError occurs if key does not exist.
`list(dictionaryVar)`	Returns a new list containing only the dictionary keys
`len(dictionaryVar)`	Returns the number of dictionary entries

4) Dictionary Methods

Methods	Semantics
`dictionaryVar.clear()`	Removes all entries from the dictionary
`dictionaryVar.get(key)` `dictionaryVar.get(key, default)`	Returns the value associated with key, else return None if key not found. If default is specified, then return default rather than None if key not found.
`dictionaryVar.pop(key)` `dictionaryVar.pop(key, default)`	Removes and returns the value associated with key. If key not found, a key error results. If default is specified, an unfound key does not result in an error, but rather in the return of default.
`dictionaryVar.keys()`	Returns a 'view' of all the keys
`dictionaryVar.values()`	Returns a 'view' of all the values
`dictionaryVar.items()`	Returns a 'view' of all the key-value tuples

5) Examples

Code

```
01 # Create a dictionary of cities with their populations
02 cityPopulation = {}
03 cityPopulation["Olathe"] = 142841
04 cityPopulation["Overland Park"] = 196636
05 print(cityPopulation)
06
07 city = input("Which city population would you like to see? ")
08 print(f"{city}'s population: {cityPopulation[city]:,d}")
09 del cityPopulation["Olathe"]
10 print(cityPopulation)
```

Output (user input in **red**)

```
{'Olathe': 142841, 'Overland Park': 196636}
Which city population would you like to see? Olathe
Olathe's population: 142,841
{'Overland Park': 196636}
```

Code

```
01 contacts = { "Andrew": "913-723-5591", "Rachel": "816-384-1212",
02              "David": "816-254-7844", "Sarah": "913-221-3278" }
03
04 name = input("Whose number do you want? ")
05 print(f"{name}'s number: {contacts[name]}")
06
07 name = input("\nWhose number do you want to change? ")
08 contacts[name] = input(f"Enter {name}'s new number NNN-NNN-NNNN: ")
09
10 name = input("\nWhose name do you want to add? ")
11 contacts[name] = input(f"Enter {name}'s number NNN-NNN-NNNN: ")
12
13 name = input("\nWhose number do you want to remove? ")
14 del contacts[name]
15
16 print(f"Contacts: {contacts}")
```

Output (user input in **red**)

```
Whose number do you want? Rachel
Rachel's number: 816-384-1212

Whose number do you want to change? Sarah
Enter Sarah's new number NNN-NNN-NNNN: 816-221-3278

Whose name do you want to add? Biruk
Enter Biruk's number NNN-NNN-NNNN: 515-628-0415

Whose number do you want to remove? David
Contacts: {'Andrew': '913-723-5591', 'Rachel': '816-384-1212',
'Sarah': '816-221-3278', 'Biruk': '515-628-0415'}
```

Code

```
01 contacts = { "Andrew": 9137235591, "Rachel": 8163841212,
02              "David": 8163841212, "Sarah": 9132213278 }
03
04 for name in contacts:
05     print(f"{name}'s number: {contacts[name]}")
```

contacts.keys() may also be used.

Output

```
Andrew's number: 9137235591
Rachel's number: 8163841212
David's number: 8163841212
Sarah's number: 9132213278
```

Before Python 3.7, dictionaries were unordered. Thus, the order displayed for earlier versions of Python may not match the same order as entered into the dictionary.

Code

```
01 contacts = { "Andrew": 9137235591, "Rachel": 8163841212,
02              "David": 8163841212, "Sarah": 9132213278 }
03
04 for name, phoneNumber in contacts.items():
05     print(f"{name}'s number: {phoneNumber}")
```

Output

```
Andrew's number: 9137235591
Rachel's number: 8163841212
David's number: 8163841212
Sarah's number: 9132213278
```

Code

```
01 contacts = { "Andrew": 9137235591, "Rachel": 8162997453,
02              "David": 8163841212, "Sarah": 9132213278 }
03 DEFAULT_NUMBER = 411
04
05 print()
06 if ("Jackson" in contacts) :
07     print(f'Jackson\'s number: {contacts["Jackson"]}')
08 else:
09     print("No contact information for Jackson")
10
11 if ("Sarah" in contacts) :
12     print(f'Sarah\'s number: {contacts["Sarah"]}')
13 else:
14     print("No contact information for Sarah")
15
16 print()
17 print(f'Jackson\'s number: {contacts.get("Jackson", DEFAULT_NUMBER)}')
18 print(f'Sarah\'s number: {contacts.get("Sarah", DEFAULT_NUMBER)}')
19 print()
20
21 for name in contacts:
22     print(f"{name}'s number: {contacts[name]}")
```

Output

```
No contact information for Jackson
Sarah's number: 9132213278

Jackson's number: 411
Sarah's number: 9132213278

Andrew's number: 9137235591
Rachel's number: 8162997453
David's number: 8163841212
Sarah's number: 9132213278
```

Code

```
01 contacts = { "Andrew": 9137235591, "Rachel": 8162997453,
02              "David": 8163841212, "Sarah": 9132213278 }
03
04 print(contacts.pop("Andrew"))
05
06 for key in contacts:
07     print(f"{key}: {contacts[key]}")
08 print()
09
10 # Sort the keys first
11 for key in sorted(contacts):
12     print(f"{key}: {contacts[key]}")
13
14 # Get a list of values using a list comprehension
15 phoneNumbers = [number for number in contacts.values()]
16 print(phoneNumbers)
17
```

```
18 # Get a list of keys using another means of building a list
19 keyList = list(contacts.keys())
20 print(keyList)
```

Output

```
9137235591
Rachel: 8162997453
David: 8163841212
Sarah: 9132213278

David: 8163841212
Rachel: 8162997453
Sarah: 9132213278
[8162997453, 8163841212, 9132213278]
['Rachel', 'David', 'Sarah']
```

Code

```
01 def main():
02     lexicon = getLexicon("Lexicon.txt")
03     showLexicon(lexicon)
04
05     word = input("\nNew word? ")
06     if (word not in lexicon):
07         lexicon[word] = input("Definition: ")
08     else:
09         print(f"{word} already in dictionary")
10     print()
11     showLexicon(lexicon)
12
13 def getLexicon(filename):
14     import csv
15     lexicon = {}
16     with open(filename, "r") as inFile:
17         csvReader = csv.reader(inFile, delimiter = ':')
18         for (word, definition) in csvReader:
19             lexicon[word] = definition
20     return lexicon
21
22 def showLexicon(lexicon):
23     for (word, definition) in sorted(lexicon.items()):
24         print(f"{word}: {definition}")
25
26 main()
```

Lexicon.txt

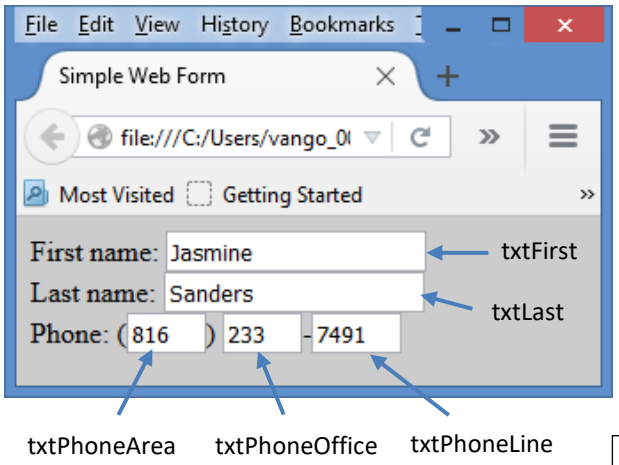
School:An institution where instruction is given.
Night:The period of darkness between sunset and sunrise.
Joke:Something said or done to provoke laughter or cause amusement.
Computer:A programmable electronic device designed to accept data, perform prescribed mathematical and logical operations at high speed, and display the results of these operations.

```
Line 07 can also be written as:
definition = input("Definition: ")
lexicon.update({word:definition})
```

Output (user input in **red**)

```
Computer: A programmable electronic device designed to accept data, perform
prescribed mathematical and logical operations at high speed, and display the
results of these operations.
Joke: Something said or done to provoke laughter or cause amusement.
Night: The period of darkness between sunset and sunrise.
School: An institution where instruction is given.

New word? Student
Definition: A person formally engaged in learning, especially one enrolled in a
school or college.

Computer: A programmable electronic device designed to accept data, perform
prescribed mathematical and logical operations at high speed, and display the
results of these operations.
Joke: Something said or done to provoke laughter or cause amusement.
```

```
Night: The period of darkness between sunset and sunrise.
School: An institution where instruction is given.
Student: A person formally engaged in learning, especially one enrolled in a school
or college.
```




```
...
   First name: <input name="txtFirst"><br />
   Last name: <input name="txtLast"><br />
   Phone: (<input name="txtPhoneArea" size="3">)
        <input name="txtPhoneOffice" size="3">-
        <input name="txtPhoneLine" size="4"><br />
...
```

Supposed Web form information has already been retrieved and placed by the Web server into a dictionary named **fields**.

Code

```
01 for formFieldName in fields:
02     print(f"{formFieldName}'s value: {fields[formFieldName]}")
```

Output

```
txtPhoneLine's value: 7491
txtFirst's value: Jasmine
txtPhoneArea's value: 816
txtPhoneOffice's value: 233
txtLast's value: Sanders
```

Code

```
01 def main():
02     gradebook = getGradebook("GradesNames.txt")
03     showGradebook(gradebook)
04
05 def getGradebook(filename):
06     import re
07     gradebook = {}
08     with open(filename, "r") as inFile:
09         for line in inFile:
10             studentInfo = re.split("[:,]", line.rstrip())
11             gradebook[studentInfo[0]] = studentInfo[1:]
12     return gradebook
13
14 def showGradebook(gradebook):
15     for (name, scores) in sorted(gradebook.items()):
16         scoresStr = ""
17         for score in scores:
18             scoresStr += f"{score:>6s}"
19         print(f"{name + ':':10s} {scoresStr}")
20
21 main()
```

GradeNames.txt

```
Darius:85.3,95,87
Callum:72,89.1,94.3
Lexi:100,78.2,95
```

Output (user input in **red**)

```
Callum:        72  89.1  94.3
Darius:      85.3    95    87
Lexi:         100  78.2    95
```

A) Object-oriented programming: A popular programming paradigm

1) What kinds of things can be objects? (Think of Nouns)
   It might represent something physical

   It might represent something conceptual or abstract (Something that is not tangible)

2) An object consists of attributes (sometimes termed properties) and behaviors
   a) **Attributes**: Characteristics of an object. What an object has. (An object "has-a/has-an" attribute)
      Implemented as **Instance Variables** (also sometimes referred to as fields or member variables):
      Variables defined inside a class where each instantiated object of that class has its own copy.**

      - **State of an object**: The current values of the _____.

      Consider a/an _____

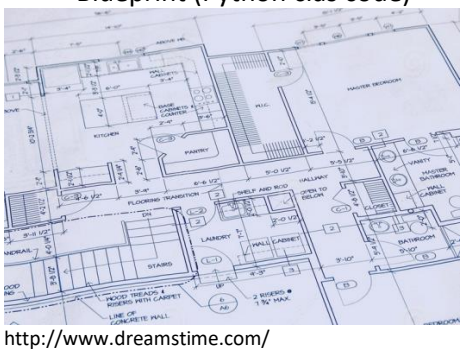
Attributes	Values (state)

   b) Behaviors (actions): Implemented as **methods**. **Methods** are functions belonging to a class and are also known as **member functions**. Think of a verb – what an object can do or be instructed to do.

      - What might be methods for a _____

** Python instance variables (non-static) are created inside a method. Static "class" variables are created outside a method, but still inside the class.

3) Objects are constructed from a class
   Blueprint (Python clas code)



http://www.dreamstime.com/
royalty-free-stock-images-blueprint-house-image1979949

Blueprint
(Python class code)

Constructing the house
(instantiation)

http://www.dreamstime.com/stock-images-luxury-american-house-image37471204

The house
(The object in memory. It is an instance of a class.)

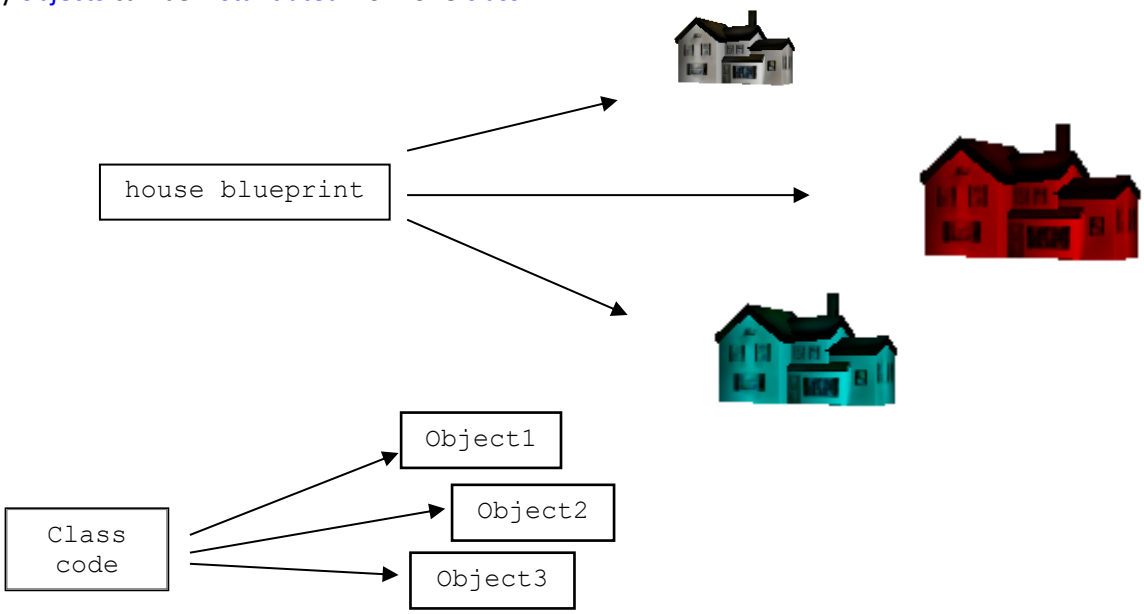
Class – code from which an object is created. A group of objects with the same attributes and methods can be created from one class.

Creating an object from a class is known as _____.

When _____ occurs, _____ is reserved for that object.

Many houses can be built from one blueprint

Many objects can be instantiated from one class



4) We have already been working with several Python built-in classes/objects. Examples include:

Class/Object	A subset of behaviors (methods)
String (The str class)	name = "JCCC Cavaliers" name = name.upper() name = name.lower() index = name.find("Cav")
List	aList = [5,3,7,5,-2,-5] aList.pop() aList.insert(2, 10)

B) Defining Beginning Programmer-Defined Classes

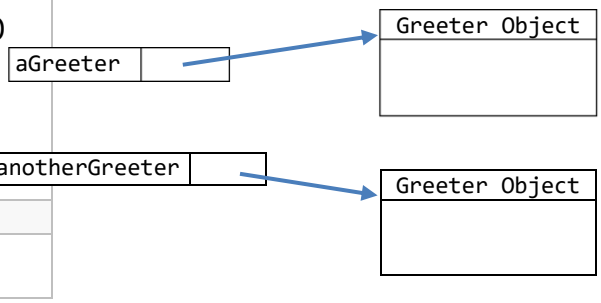
1) A Greeter Class

Code

```
01 class Greeter:
02     def speak(self, name):
03         print(f"Welcome {name}!")
04
05 aGreeter = Greeter()
06 aGreeter.speak("Tyson")
07 anotherGreeter = Greeter()
08 anotherGreeter.speak("Elise")
```

Output

```
Welcome Tyson!
Welcome Elise!
```



Note: There is only one speak method for all Greeter objects. self tracks which object calls the method.

2) A Dog class with its own attribute (also known as an instance variable or field)

Code

```
01 class Dog :
02     def setBreed(self, newBreed):
03         self.breed = newBreed
04
05     def getBreed(self):
06         return self.breed
07
08
09 bigDog = Dog()
10 bigDog.setBreed("St. Bernard")
11 smallDog = Dog()
12 smallDog.setBreed("Yorkshire Terrier")
13 print(bigDog.getBreed())
14 print(smallDog.getBreed())
15 bigDog.setBreed("Great Dane")
16 print(bigDog.getBreed())
```

Output

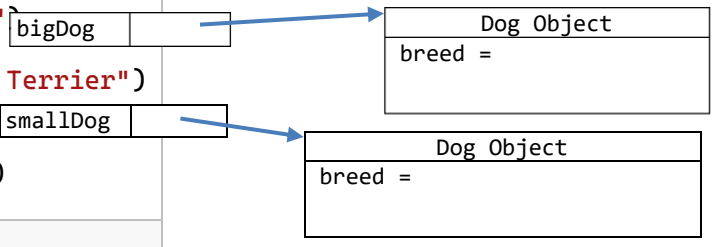
```
St. Bernard
Yorkshire Terrier
Great Dane
```

Mutator: A method that only changes the value of an attribute.
- Also known as a **setter** or **set method** (setInstanceVarName)
- Header documentation is optional in our course.

Accessor: A method that only returns the value of an attribute.
- Also known as a **getter** or **get method** (getInstanceVarName)
- Header documentation is optional in our course.

Note: The parameter and attribute names can be the same for any method (but they are different variables). Example:

```
def setBreed(self, breed):
    self.breed = breed
```



Coding Style:
- Programmer-defined class names are written in **UpperCamelCase**. Examples: Dog, BankAccount

3) Umpire clicker for baseball and softball

src: https://www.walmart.com/ip/MacGregor-4-Way-Umpire-s-Indicator/...

4) The code for a beginning umpire clicker – only modeling and testing the inning attribute

Code

```
01 # Class UmpClicker models an umpire clicker where innings
02 # can be tracked.
03 # @author First Last
04 class UmpClicker :
05     def setInning(self, newInning) :
06         self.inning = newInning
07
08     def getInning(self) :
09         return self.inning
10
11     # nextInning adds 1 to the current inning. If it was
12     # the 9th inning, then the current inning resets to 1.
13     def nextInning(self) :
14         MAX_INNINGS = 9
15         if (self.inning == MAX_INNINGS)
16             self.setInning(1)
17         else:
18             self.inning += 1
19
20 # Test the clicker
21 clicker = UmpClicker()
22 clicker.setInning(1)
23 clicker.nextInning()
24 clicker.nextInning()
25
26 curInning = clicker.getInning()
27 print(f"Current Inning: {curInning}")
28
29 MAX_INNINGS = 9
30 for i in range(curInning, MAX_INNINGS + 1):
31     clicker.nextInning()
32 print(f"Current Inning: {clicker.getInning()}")
```

The if-else shows how a method can call another method. However, note that lines 15 – 18 could be reduced to:

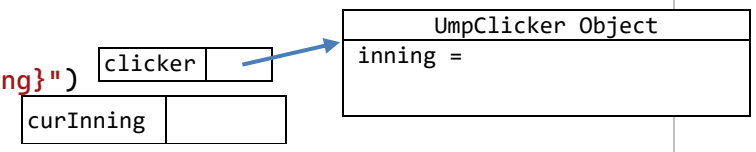
```
self.inning = self.inning % MAX_INNINGS + 1
```

If only utilizing accessors and mutators, line 18 could be written as:

```
self.setInning(self.getInning() + 1)
```

or lines 15 – 18 could be changed to

```
self.setInning(self.getInning() %
                MAX_INNINGS + 1)
```



Output

```
Current Inning: 3
Current Inning: 1
```

Reminder: The notes are simplified somewhat in that types such as numbers and strings are shown as being stored directly into the variable; however, it really is the location of a new numeric or string object that is stored.

What is the class name?

What is the class attribute (i.e. instance variable, field)?

Which are the class behaviors (methods)?

Which line of code instantiates a class object?

What is the object name (the name of the variable that refers to the object)?

What operator (symbol) is needed to call a given object's method?

5) Code Reuse: Separating the class into a different file (module) so it may be reused by other code

UmpClicker.py

```
01 # Class UmpClicker models an umpire clicker where innings
02 # can be tracked.
03 # @author First Last
04 class UmpClicker :
05     def getInning(self) :
06         return self.inning
07
08     def setInning(self, newInning) :
09         self.inning = newInning
10
11     # nextInning adds 1 to the current inning. If it was
12     # the 9th inning, then the current inning resets to 1.
13     def nextInning(self) :
14         MAX_INNINGS = 9
15         if (self.inning == MAX_INNINGS) :
16             self.setInning(1)
17         else:
18             self.setInning(self.getInning() + 1)
```

Remember, this is the same as `self.inning += 1`

Driver.py

```
01 # This program tests the umpire clicker
02 # @author First Last
03 from UmpClicker import UmpClicker
04
05 # Test the clicker
06 clicker = UmpClicker()
07 clicker.setInning(1)
08 clicker.nextInning()
09 clicker.nextInning()
10
11 curInning = clicker.getInning()
12 print(f"Current Inning: {curInning}")
13
14 MAX_INNINGS = 9
15 for i in range(curInning, MAX_INNINGS + 1):
16     clicker.nextInning()
17 print(f"Current Inning: {clicker.getInning()}")
```

class name

file name (no .py at the end)

Output

```
Current Inning: 3
Current Inning: 1
```

6) Constructors

a) Be careful with the prior code. What happens if clicker.setInning() is not invoked (called) before clicker.nextInning() or clicker.getInning()? Why?

b) **Constructor**: A special method that creates and prepares a new object for use.

(i) Syntax is somewhat different from other languages

(ii) Adding a constructor to the UmpClicker class

UmpClicker.py

```
01 # Class UmpClicker models an umpire's clicker where innings
02 # can be tracked.
03 # @author First Last
04 class UmpClicker :
05     def __init__(self) :
06         self.setInning(1)
07
08     def getInning(self) :
09         return self.inning
10
11     def setInning(self, newInning) :
12         self.inning = newInning
13
14     # nextInning adds 1 to the current inning. If it was
15     # the 9th inning, then the current inning resets to 1.
16     def nextInning(self) :
17         MAX_INNINGS = 9
18         if (self.inning == MAX_INNINGS) :
19             self.setInning(1)
20         else:
21             self.setInning(self.getInning() + 1)
```

Note: There are 2 consecutive underscore characters on each side of **init** even though certain fonts may make it appear as 1.

The inning instance variable is now defined automatically upon object instantiation.

Driver.py

```
01 # This program tests the umpire clicker
02 # @author First Last
03 from UmpClicker import UmpClicker
04
05 # Test the clicker
06 clicker = UmpClicker()
07 clicker.setInning(1)
08 clicker.nextInning()
09 clicker.nextInning()
10
11 curInning = clicker.getInning()
12 print(f"Current Inning: {curInning}")
13
14 MAX_INNINGS = 9
15 for i in range(curInning, MAX_INNINGS + 1):
16     clicker.nextInning()
17 print(f"Current Inning: {clicker.getInning()}")
```

No longer need to call setInning in the driver before calling nextInning() or getInning() in a newly instantiated object.

Output

```
Current Inning: 3
Current Inning: 1
```

7) Revised UmpClicker class

Language Observation: In languages like C++, C#, and Java, a class may contain more than one constructor. Python permits only one constructor per class.

UmpClicker.py

```
01 # Class UmpClicker models an umpire's clicker where innings
02 # can be tracked.
03 # @author First Last
04 class UmpClicker :
05     MAX_INNINGS = 9
06
07     # Constructor to set initial innings
08     # @param inning The initial inning
09     def __init__(self, newInning = 1) :
10         self.setInning(newInning)
11
12     def getInning(self) :
13         return self.inning
14
15     def setInning(self, newInning) :
16         if newInning >= 1 and newInning <= UmpClicker.MAX_INNINGS :
17             self.inning = newInning
18         else :
19             self.inning = 1
20
21     # nextInning adds 1 to the current inning. If it was
22     # the 9th inning, then the current inning resets to 1.
23     def nextInning(self) :
24         self.setInning(self.getInning()
25                        % UmpClicker.MAX_INNINGS + 1
```

Class variable: Shared by all instances (objects) of the class. It is treated as a named constant for the UmpClicker class because its value will always be 9; however, a class variable could also be a variable with changing values.

Formal parameter added that defaults to one. Can now directly start a previously postponed game in an inning other than one.

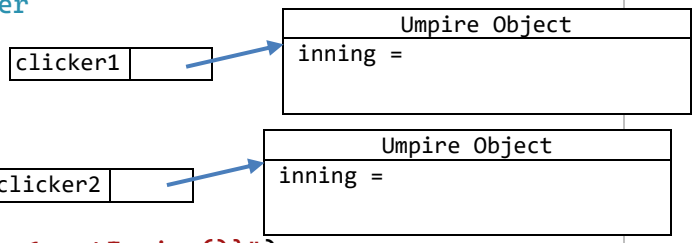
The mutator is updated to guard against invalid data: If the incoming parameter data is not in a valid range, then the inning attribute is set to 1.

Remember, in Python, line 16 could be:
`if 1 <= newInning <= UmpClicker.MAX_INNINGS :`

Best to access class variables by using the name of the class.

Driver.py

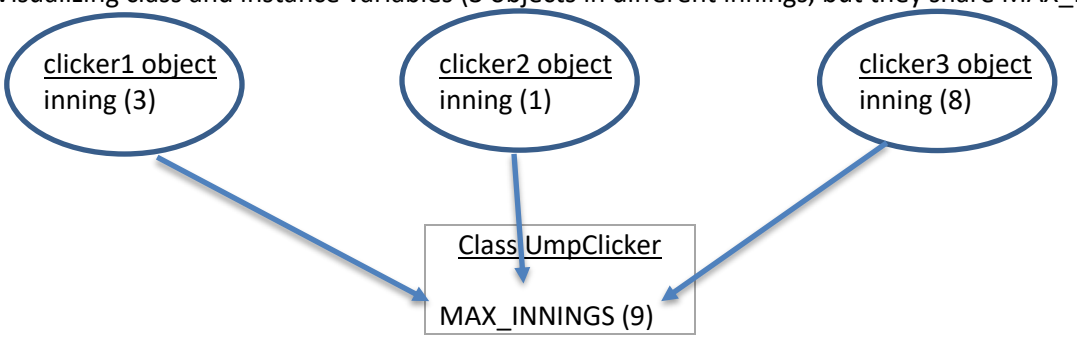
```
01 # This program tests the umpire clicker
02 # @author First Last
03 from UmpClicker import UmpClicker
04
05 # Test the clicker
06 clicker1 = UmpClicker()
07 clicker2 = UmpClicker(3)
08 clicker1.nextInning()
09 clicker2.nextInning()
10
11 print(f"clicker1 Inning: {clicker1.getInning()}")
12 print(f"clicker2 Inning: {clicker2.getInning()}")
13
14 for i in range(clicker1.getInning(),
15                UmpClicker.MAX_INNINGS + 1):
16     clicker1.nextInning()
17 print(f"clicker1 Inning: {clicker1.getInning()}")
18
19 clicker2.setInning(-5)
20 print(f"clicker2 Inning: {clicker2.getInning()}")
```



Output

```
clicker1 Inning: 2
clicker2 Inning: 4
clicker1 Inning: 1
clicker2 Inning: 1
```

8) Visualizing class and instance variables (3 objects in different innings, but they share MAX_INNINGS)



9) More Definitions

Method (or function) signature: The method name together with its parameter list. (e.g. getInning())
Public interface of a Class: Set of all public class method signatures together with a description of their behaviors.

Class UmpClicker Interface

```
# Constructor to set initial innings
# @param newInning The initial inning
__init__(newInning = 1)

# Get the current inning.
# @return the current value
getInning()

# Set the current inning. Invalid inning set to 1.
# @param newInning the desired inning
setInning(newInning)

# nextInning adds 1 to the current inning. If it was
# the 9th inning, then the current inning resets to 1.
def nextInning() :
```

Encapsulation: Combining the data (instance variables) and the code (methods) that manipulate that data into a single unit. The intent is to provide a public interface while hiding the data and code details.
For class UmpClicker: The value of the inning is meant to be hidden and only modified/accessed by the UmpClicker methods.

10) A note about Python class member privacy

Python class attributes (instance variables) may be directly accessed from outside the class by using objectName.attribute. However, it is safest to only use the accessors/mutators to access and modify instance variables. **Unless otherwise stated, only access/modify instance variables through the accessors and mutators. Exception: If writing code inside the class, this course will permit accessing the instance variables directly.**

Example: clicker1.inning = -10 is valid code, but it results in an invalid inning (using setInning() with its built-in if statement guards against this.)

> **Language Observation**: In languages like C++, C#, and Java, instance variables and methods can be preceded by the private keyword to ensure that access/modification is only permitted within the class.

Note 1: Pseudo-privacy can be achieved in Python by inserting two consecutive underscores before an identifier in a class (name-mangling). These notes do not utilize this approach.
Note 2: The textbook prepends an underscore to the beginning of an instance variable that is meant to be private (e.g., self._inning). The notes/coding style don't do this, but you are permitted to do so.

C) A Simple Person Class

person.py

```
01 # Class Person models a person with a name and age
02 # @author First Last
03 class Person :
04     # Constructor
05     # @param name The person's name
06     # @param age The person's age
07     def __init__(self, name = "", age = 0) :
08         self.setName(name)
09         self.setAge(age)
10
11     def getName(self) :
12         return self.name
13
14     def setName(self, newName) :
15         self.name = newName
16
17     def getAge(self) :
18         return self.age
19
20     def setAge(self, newAge) :
21         if newAge >= 0:
22             self.age = newAge
23         else:
24             self.age = 0
25
26 defaultPerson = Person()
27 aPerson = Person("Derrick", 25)
28
29 print(f"{defaultPerson.getName()}, {defaultPerson.getAge()}")
30 print(f"{aPerson.getName()}, {aPerson.getAge()}")
```

FYI: An instance variable is preceded by self dot inside a method. If a variable is created inside a method and is not preceded by self dot, then it is only a local variable used by that method

```
class Person :
    def count(self):
        limit = int(input("How high? "))
        for i in range(1, limit + 1):
            print(i, end = " ")
```

limit and i are local variables known only by count. A method could utilize instance, local, and class variables (and even global variables). For example, in setName, newName is a local variable, but name (self.name) is an instance variable.

Output

```
, 0
Derrick, 25
```

Which method is the constructor?

Which methods are the accessors?

Which methods are the mutators?

D) Python Feature – Including a/an `__str__(self)` method (There are two underscores before and after str)

Person.py

```
01 # Class Person models a person with a name and age
02 # @author First Last
03 class Person :
04     def __init__(self, name = "", age = 0) :
05       self.setName(name)
06       self.setAge(age)
07
08     def getName(self) :
09         return self.name
10
11     def setName(self, newName) :
12         self.name = newName
13
14     def getAge(self) :
15         return self.age
```

```
16
17      def setAge(self, newAge) :
18          if (newAge >= 0):
19              self.age = newAge
20          else:
21              self.age = 0
22
23      def __str__(self):
24          return f"{self.getName()}, {self.getAge()}"
```

Driver

```
01 defaultPerson = Person()
02 aPerson = Person("Derrick", 25)
03
04 print(defaultPerson)
05 print(aPerson)
06
07 objectState = str(aPerson)
08 print(objectState)
```

Output

```
, 0
Derrick, 25
Derrick, 25
```

Note: Python programmers may also utilize `__repr__(self)`. This can be written to produce a more thorough string representation of the object that is intended for programmers rather than end-users.

E) Method Chaining – The dot operator links together multiple method calls in one statement

Person.py

```
01 # Class Person models a person with a name and age
02 # @author First Last
03 class Person :
04     def __init__(self, name = "", age = 0) :
05         self.setName(name).setAge(age)
06
07     def getName(self) :
08         return self.name
09
10     def setName(self, newName) :
11         self.name = newName
12         return self
13
14     def getAge(self) :
15         return self.age
16
17     def setAge(self, newAge) :
18         if (newAge >= 0):
19             self.age = newAge
20         else:
21             self.age = 0
22         return self
23
24     def __str__(self):
25         return f"{self.getName()}, {self.getAge()}"
```

Two method calls chained together.

To implement method chaining, a method needs to return an object of the class. Quite often, this is self.

Driver

```
01 father = Person()
02 father.setName("Diego").setAge(55)
03 son = Person("Julian", 25)
04
05 print(father)
```

```
06 print(son)
07
08 father.setAge(56)
09 print(father)
```

Can still call the method separately and ignore the object returned.

Output

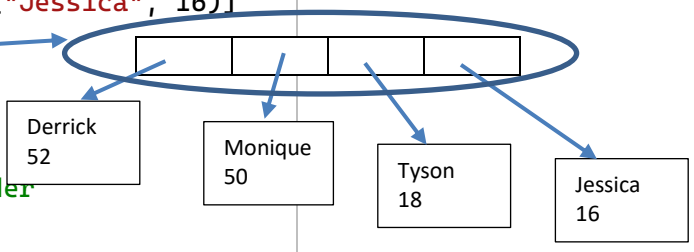
```
, 0
Derrick, 25
Derrick, 25
```

F) A List of (mutable) Person Objects

1) Driver without functions

Driver

```
01 # This program creates a family and increases the
02 # age of family members
03 # @author First Last
04 from Person import Person
05
06 # Create the family
07 family = [Person("Derrick", 52), Person("Monique", 50),
08           Person("Tyson", 18), Person("Jessica", 16)]
09
10 # Show the family members
11 for member in family:
12     print(member)
13
14 # Make all family members one year older
15 for member in family:
16     member.setAge(member.getAge() + 1)
17
18 # Show the family members
19 print()
20 for member in family:
21     print(member)
```



Output

```
[Derrick, 52]
[Monique, 50]
[Tyson, 18]
[Jessica, 16]

[Derrick, 53]
[Monique, 51]
[Tyson, 19]
[Jessica, 17]
```

2) Modularizing the prior program with functions

Driver

```
01 # This program creates a family and increases the
02 # age of family members
03 # @author First Last
04
05 from Person import Person
06
07 def main():
08     # Create the family
09     family = [Person("Derrick", 52), Person("Monique", 50),
10               Person("Tyson", 18), Person("Jessica", 16)]
11     showFamily(family)
12     increaseAge(family, 1)
13     print()
14     showFamily(family)
15
16 # showFamily displays information for each family member
17 # @param newFamily A list of family members
18 def showFamily(newFamily) :
19     for member in newFamily:
20         print(member)
21
22 # increaseAge increases the age for each family member
23 # by a given amount
24 # @param newFamily A list of family members
25 # @param amount The amount by which to increase the ages
26 def increaseAge(newFamily, amount):
27     for member in newFamily:
28         member.setAge(member.getAge() + amount)
29
30 main()
```

A) Inheritance Hierarchies

Create a hierarchy for the following terms ranging from the most generic to the most specific.

Bird, Cat, Tiger, Animal, Falcon, Lion, Eagle

Each class becomes more specific as the hierarchy is traversed _____.

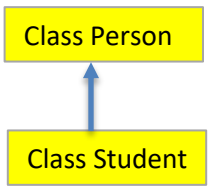
Conversely, each class becomes more general hierarchy is traversed _____.

Terms: **Superclass** (parent class, base class), **Subclass** (child class, derived class), "**Is-a**"

B) Example 1

studentPerson.py

```
01 # Class Person models a person with a name and age
02 # @author First Last
03 class Person :
04     def __init__(self, name = "", age = 0) :
05         self.setName(name)
06         self.setAge(age)
07
08     def getName(self) :
09         return self._name
10
11     def setName(self, name) :
12         self._name = name
13
14     def getAge(self) :
15         return self._age
16
17     def setAge(self, age) :
18         self._age = age if (age > 0) else 0
19
20     def __str__(self):
21         return f"{self.getName()}, {self.getAge()}"
22
23 # Class Student models a student with a name, age, and tuition
24 # @author First Last
25 class Student(Person):
26     # Constructor
27     def __init__(self, name = "", age = 0, tuition = 0.0) :
28         super().__init__(name, age)
29         self.setTuition(tuition)
30
31     def getTuition(self) :
32         return self.tuition
33
34     def setTuition(self, tuition) :
```



How inheritance is implemented: **class childClass(parentClass) :**

```
35          self.tuition = tuition if (tuition > 0) else 0
36
37      def __str__(self):
38          return f"{super().__str__()}, ${self.getTuition():,.2f}"
39
40  # displayPerson displays a person and differentiates
41  # between a person and a student
42  # @param person The person to display
43  def displayPerson(person):
44      print(f'{"Person:" if person is Student else "Student:"}')
45      print(person)
46      print()
47
48  defaultStudent = Student()
49  aStudent = Student("Derrick", 25, 1200)
50  aPerson = Person("Amy", 21)
51
52  displayPerson(defaultStudent)
53  displayPerson(aStudent)
54  displayPerson(aPerson)
55  displayPerson(aStudent.getName())
56
```

Method overriding: When a method in a child class has the same signature as a method in a parent class: `__str__(self)` in this case

Print called the correct __str__ method even though it was sent two students and one person.

Polymorphism: "Many Forms". In this case the `__str__` method performed differently depending upon which object was sent.
Also – think how the + operator behaves differently if it is utilized by a string (concatenation) or a number (addition0.

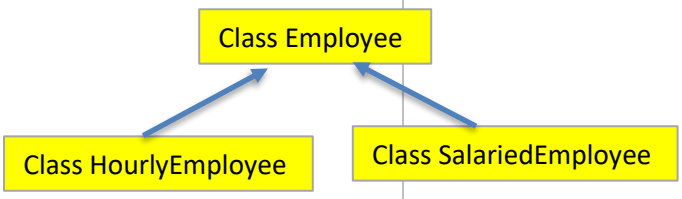
Output

```
Student:
, 0, $0.00

Student:
Derrick, 25, $1,200.00

Student:
Amy, 21

Student:
Derrick
```

C) Example 2

Employee.py

```
01  # Class Employee represents an employee with and id.
02  # @author First Last
03  class Employee :
04      def __init__(self, id = "") :
05          self.setId(id)
06
07      def getId(self) :
08          return self.id
09
10      def setId(self, id) :
11          self.id = id
12
```



HourlyEmployee.py

```
01  from Employee import Employee
02  # Class HourlyEmployee represents an hourly employee with a wage.
03  # @author First Last
04  class HourlyEmployee(Employee) :
05      def __init__(self, id = "", wage = 0.0) :
06          super().__init__(id)
```

```
07          self.setWage(wage)
08
09      def getWage(self) :
10          return self.wage
11
12      def setWage(self, wage) :
13          self.wage = wage if (wage > 0) else 0
14
15      def getWeeklyPay(self):
16          HOURS_IN_WEEK = 40
17          return self.getWage() * HOURS_IN_WEEK
18
```

SalariedEmployee.py

```
01 from Employee import Employee
02 # Class SalariedEmployee represents a salaried employee with a salary.
03 # @author First Last
04 class SalariedEmployee(Employee) :
05     def __init__(self, id = "", salary = 0.0) :
06         super().__init__(id)
07         self.setSalary(salary)
08
09     def getSalary(self) :
10         return self.salary
11
12     def setSalary(self, salary) :
13         self.salary = salary if (salary > 0) else 0
14
15     def getWeeklyPay(self):
16         WEEKS_IN_YEAR = 52
17         return self.getSalary() / WEEKS_IN_YEAR
18
```

testDriver.py

```
01 # This program tests employee objects
02 # @author First last
03
04 from HourlyEmployee import HourlyEmployee
05 from SalariedEmployee import SalariedEmployee
06
07 employees = []
08 employees.append(HourlyEmployee("0012377", 12.50))
09 employees.append(SalariedEmployee("4578107", 50200))
10 employees.append(HourlyEmployee("8765290", 13.25))
11 employees.append(SalariedEmployee("5647389", 45120))
12
13 for emp in employees:
14     print(f"{emp.getId()} -> ${emp.getWeeklyPay():,.2f}")
```

Output

```
0012377 -> $500.00
4578107 -> $965.38
8765290 -> $530.00
5647389 -> $867.69
```

The correct getWeeklyPay was called depending upon which object was referred to by emp.

D) "Abstract" Classes

Python does not implement abstract classes like other languages do. These are classes that will not be directly instantiated but contain methods that are intended to be overriden by sub classes which are to be instantiated. For example, this could be done for getWeeklyPay in the prior Employee class:

Employee.py

```
01 # Class Employee represents an employee with and id.
02 # @author First Last
03 class Employee :
04     def __init__(self, id = "") :
05         self.setId(id)
06
07     def getId(self) :
08         return self.id
09
10     def setId(self, id) :
11         self.id = id
12
13     def getWeeklyPay(self) :
14         raise NotImplementedError
```

Subclasses are to override getWeeklyPay(self) if they are to be instantiated. In this way, subclasses will contain a consistent interface for their implementation of getWeeklyPay(self). class Employee may be too generic to implement a method for determining pay.