CSC110 Supplemental Reading: Week02

Contents

1 Expressions

Now that we've learning about data and variables, we need to understand what an expression is:

An *expression* is any combination of *variables*, *constants*, *operators* and/or *function(method) calls* that, when evaluated, can be reduced to a single value. This is true for both string expressions and numeric expressions. A string expression will simplify to a single String; a numeric expression will simplify to a single number.

We've learned about variables and constants already. Now it's time to learn about operators.

1.1 String Expressions

String literals and string variables can be joined together to make longer strings by using the **string concatenation operator** -- a plus sign (+). The result is a **string expression**. Here are some examples:

```
temp = 'hot'
word = 'shot'
print  ('The coffee is ' + temp )
print ( 'That engineer is a real ' + temp + word )
ans = input( temp + " or cold?" )
```

In each of these examples, the part in blue is a **string expression**. Notice that some are string literals and others are combinations of string literals and variables using the concatenation operator. When using the concatenation operator, no extra space is added.

String expressions may be used wherever a string value is needed (why? because an expression always evaluates to a single value!) Try these examples in the Python shell to see what the results are.

Another interesting operator to use with a string is the **repetition operator** -- the star symbol(*). Just like in math, where multiplication gives us repeated additions, when using the * with strings, you get repeated concatenations. Here's an example:

```
word = 'taste'
longWord = word * 3
print ( longWord )
```

What is stored in longWord? The string 'tastetastetaste' Try this in the Python shell.

1.2 Numeric Operators and Operator Precedence

Here is a list of the mathematical operators in Python that we'll be using. Each of these may be applied to numbers (or variables that contain numbers) to construct a numeric expression.

Python Operator	Description
**	exponentiation
-	unary minus (gives the value with the opposite sign)
* / // %	Multiplication Division Integer division Remainder (also known as modulus)
+ -	Addition Subtraction

The book goes through these operators well, but seems to have missed the unary minus:

The **unary minus** operator will change the sign of the *expression* (it does not change the contents of the variable). For example:

```
x = 3
y = - x
```

This stores a -3 in y. Realize that the variable x hasn't changed.

Important point: **A Python assignment statement is not the same as an algebraic equality!** You could never write

```
x = x + 1
```

in algebra, but this is an extremely common construction in programming. We would read this statement "x **gets** x plus 1". That is, "calculate the value of x + 1 and store the result in the variable x". The Python '=' operator is so different from mathematical '=' operator that we use a different word (assignment instead of equals) when we read code containing this operator!

Statements that begin with a construction like 'x = x ...' are very common in programming because we often need to change the value of a variable to a new value that is based on its old value. The compound assignment operators reduce the amount of typing in situations like this.

2 More String operations

There are several string operations that programmers find useful. The first is to determine how many characters a string has; its length. There is a built-in Python function we can use: **len()**

For example, here's some Python code you can try in the shell:

```
someString = 'Tuesday is odd.'
length = len(someString)
```

The value 15 is stored in the variable length. That is the length of the string stored in someString. The basics of the len() function are as follows: pass in a string expression (in the parentheses) and the function returns the length of the string.

Here are some other example calls. Can you figure out what value would be returned?

Expression

```
len('cup')
len('how long is this string?')
len(someString + '123') # remember, any string expression
```

In some programs you want to work with substrings, or portions of strings. First, we need to understand that the characters in strings are located at indexes numbered 0 to len() -1. For example, take the string 'hello' It has length 5. The 'h' is at index 0, the 'e' is at index 1, and the 'o' is at index 4 (the length - 1) This type of counting, starting at 0, is very common in programming.

Ok, now that we know how the characters in a string are numbered, we can then access a single character by its index. The syntax is: ***str[index]*** Here are some example expressions and their values:

```
someString = 'Tuesday is odd.'
```

Expression	Value
someString[0]	'T'
someString[3]	's'
someString[4]	'd'

Notice how we use the square brackets, we put the index number that we want to access inside. This is an expression, the value is another string with one character.

We can expand on the square bracket[] syntax to ask for a substring, or a slice, of the original string. The syntax is: ***str[start:stop]*** Where start is the starting index and stop is the stopping index. The colon is required.

Here are some more example expressions and their values:

```
someString = 'Tuesday is odd.'
```

Expression	Value
`someString[0:3]`	'Tue'
`someString[4:6]`	'da'
`someString[11:15]`	'odd.'

Notice that the character at the stopping index isn't included; that is the index of where to stop.

Also, understand that the square bracket [] syntax does not change the original string. Once strings are created, they cannot be changed; they are immutable.

3 Numeric Data vs String Data

***Numeric data is not the same as string data*.** computers represent and store these two different kinds of data differently. So, the following two statements are **not the same**:

```
x = '14'
y = 14
```

The first stores in the variable x a 2-character string. The second stores a number with the value 14.

***It is important to always be aware of the type of data stored in a variable*.** Any type of data may be stored in any variable in Python (unlike some other programming languages). **If the wrong data type is used, expressions may not produce the correct results.** For example, consider the following:

```
str1 = '14'
num1 = 14
ans1 = str1 + str1
ans2 = num1 + num1
```

The value of ans1 is the 4-character string '1414'. The value of ans2 is the number 28. Do you understand why? In the first statement, ***the + is doing string concatenation***. In the second statement, ***the + is doing arithmetic***.

4 Mixed Data Type Expressions

When using the math operators with the same data types, the result value is always the same type. For example *int*/*int* will give an int result, where *float*/*float* will give you a float result.

When you have a **mixed-type expression**, like *float*/*int* or *int* + *float*, then the integer value will temporarily be converted to a float to complete the operation.

How about mixing numbers and strings? Python doesn't like that. If you tried to use the contenation operator like this: 'hello number ' + 3 You'll get an error. If you want Python to treat the numeric value as a string, you have to tell it that, using the **str()** function.

For example, the expression `'hello number ' + str(3)` will give you the string 'hello number 3'.

The same holds true, and will be more useful, when working with variables.

5 Built-in functions and the math module

Up to this point we've learned about a number of built-in functions:

- `input( )`
- `print( )`
- `int( )`
- `float( )`
- `len( )`
- `str( )`

You might want to review the usage for each of these. Let's also review the basic operation of functions:

- we **call** functions by typing their name and parentheses
- if a function has **parameters** -- required data that it needs -- then we type in arguments to supply that data. Arguments can be any expression as long as it simplifies to the correct data type
- If a function's job is to **return** a value (as all of these are) then we can use a function call as another expression. We can assign the return value into a variable, or use the function call as part of a larger expression.

Another useful function is **abs( )**, which returns the absolute value of the argument passed in. For example:

Expression	Value
`abs(-14)`	14
`x = 25` `abs(x - 100)`	75

One more useful, but complicated function, is **format( )**, added to Python in version 3. It gives you greater control of how output is displayed. **In every case, the result, or return value, of format( ) is a string!**

The basic syntax of the format( ) function is:

format(expression, format_spec), where **format_spec** ::= [[width] [,] [.precision] [type]]

In the format_spec, [ ] contain a parameter that is optional. Most of the time you are only using format( ) to display results, so it is usually used with the print( ) function. An example that uses all format_spec options would be **'10,.2f'**, which means a field of 10 characters, using , to separate orders of magnitude, 2 decimal places, and a floating point value. For more information, read the Python docs.

Code	Displays	Notes
`number = 47.123825` `print("The number is ", format( number, '.3f' ))`	47.124	rounds to 3 decimal places
`number = 6282097.3` `print("The number is ", format( number, '.2e'))`	6.28e5	rounds to 2 decimal places in scientific notation
`number = 6282097.3` `print("The number is ", format( number, ',.f'))`	The number is 6,282,097.30	inserts , to separate orders of magnitude. also notice 2 decimal places
`number = 382.59` `print("Answer is ", format( number, '8.1f' ))`	Answer is 382.6	notice leading spaces before 382.6 8.1 means 8 characters for entire number with 1 decimal place

5.1 The math module

There are a number of other mathematical operations that are useful to programmers, as well as numeric constants, like PI. These operations and constants are bundled together into a library, or module in Python. The particular module we'll want to use is the math module. To use a module in our code, first we must *import the math module*:

```
import math # you need this statement at the top of your Python program
```

Then, we can access the members of the module, using the math name. Here are some of the functions available:

Using functions and constants from math module	Description
`math.sqrt(x)`	returns the square root of x
`math.ceil(x)`	returns the smallest integer that is greater than or equal to x
`math.floor(x)`	returns the largest integer that is less than or equal to x
`math.pi`	This is the constant for PI

Here are some sample calls and their values:

Expression	Value
`math.sqrt(100)`	10.0
`y = 80` `math.sqrt( y + 1)`	9.0
`math.ceil(math.pi)`	4.0

You can find a lot more math module information from the Python documentation link under Python Resources at the top of the class homepage.

6 Software Development Lifecycle (different from the text)

"The sooner you start coding your program, the longer it's going to take." - Henry Ledgard

Writing Python programs ("coding") is only one part of the process of developing a computer program. This process is known as the **Software Development Lifecycle**. Here is one way to view the entire program development process:

1. **ANALYZE** the problem -- clearly define the problem and ensure that it is fully understood. Determine the required input and output.
2. **DESIGN** the solution to the problem -- that is, develop an algorithm that will solve the problem.
3. **CODE** -- translate the algorithm into code. This step includes the development of an effective user interface that obtains required inputs from the user and displays results (output).
4. **TEST** and **DEBUG** the program. How can you prove to yourself (and others) that the program works for all expected and unexpected inputs?
5. **DOCUMENT** the program. Documentation consists of all materials that describe the program, its purpose, and how it operates. Full documentation is an essential element of every programming project.

This process is often actually expressed as a *cycle*, because it may be necessary at any point to backtrack to an earlier step and then move forward again. For example, testing may prove that the algorithm is incorrect, necessitating a fall back to the design stage.

Let's take a closer look at each step in the process, especially as it applies to the work you will do this quarter.

6.1 Analyze the Problem

The goal of the first step in the process is to fully understand the objectives of the program. Most projects begin as a **specification** from a 'customer'. In general, the initial specification may be incomplete, inconsistent, or even impossible -- or, the programmer may simply not understand what is needed. An initial specification is just a starting point, and must be interpreted, refined and completed by the programmer in consultation with the customer. Once both parties are in agreement about what the program will do, the project can continue.

Also in this step, the programmer needs to convert the narrative specification into a list of specific programming tasks -- inputs that need to be obtained from the user and outputs that the program will produce. In the next step, we will consider the processing steps (e.g. calculations) that need to be performed in order to produce the required outputs from the inputs provided.

In this course, a homework assignment serves as a specification and your instructor is the customer! Fortunately, this 'customer' is experienced at writing specifications, so you should not have difficulties 🙂. Yet, there are times that I make mistakes. Your primary objective will be to make sure you *understand* what the assignment requires. If you are uncertain about the requirements or have questions about some aspect of the specification, or I've forgotten something, then you need to seek clarification (e.g. post a message to the forum). Sometimes I may have to clarify or change a specification after a homework assignment is posted.

6.2 Design an Algorithm to Solve the Problem

Think about building a house. the crew that does the building doesn't just make it up off the tops of their heads. They use a blueprint: a design that tells them what to build. The same holds true with programming: in order to program a computer to solve a problem, you need to have a plan -- ***a logical sequence of precise steps that solves the problem*** -- this is known as an **algorithm**! This may be as simple as a formula, or it may be a complex procedure involving decisions and repetition. Developing an algorithm that solves the problem is what the design step is all about!

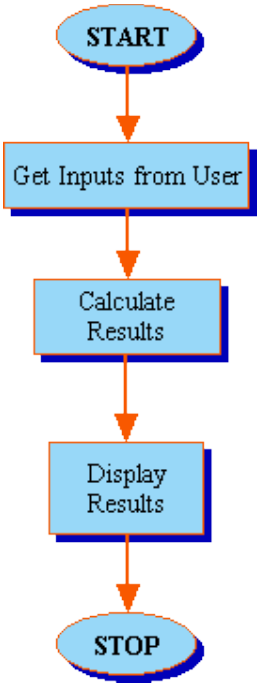
Your goal is to get your algorithm down on paper. To start, it will be helpful to have an outline, just as if you were writing a paper. For most programs, the outline will follow a general pattern that breaks the algorithm down into 3 or 4 regions. The primary regions are

- **Introduction** -- program introduction to the user
- **Input** -- get some information from the user
- **Processing** -- calculate something or perform other processing (*implement* the algorithm)
- **Output** -- display calculated results to the user

An additional region may be needed before any of these -- a section at the top of the program where values are assigned to **constants** for use throughout the program. **A constant *is simply a variable whose value is assigned one time and is never changed*.** By giving the constant a meaningful name, and then using that name wherever the special value is needed, a program is made more reliable and more readable.

(continues…)

For programs with any level of complexity, and especially for programs that depart from a simple 'linear' flow (something that will be possible as soon as we study selection statements), it is sensible to use some kind of design tool to help visualize the 'flow' of the program as it is executed. One such tool is a **flowchart** (see example at right).



A flow chart makes use of different kinds of boxes (or '**blocks**') to represent each step in the execution of a program, and **flow lines** to show the flow of control between blocks. Program flow is always in the direction of the arrows, so steps are executed in a particular order.

In general, **if the steps of a program are executed out of order, the program will not produce the same results**. **Order matters!** Flow charts help the programmer visualize the order of execution.

You will not be required to draw any flowcharts, but consider using a tool like this if it helps you visualize the operation of your algorithm. I will include a few on the web site throughout the quarter to illustrate the operation of control structures like selection structures and loops.

Another, more popular design tool is **pseudocode**: ***a blueprint for a program written in a form that can be easily converted into actual code***. There are no syntax rules for pseudocode. Rather, think of it as a shorthand description of the program steps written **in the order they will be performed**. Writing out the steps in shorthand before writing code actually saves time, because corrections to your logic can be made without struggling with error messages. Once the pseudocode seems correct, it can be converted into actual code.

One popular and valuable technique is to write pseudocode in the form of a series of short comments. **These comments then form a guide for writing the actual code**, and can remain in place as a reminder of what the code is supposed to do. This is a great way to avoid the problem of accidentally skipping a step in a program!

Once you have come up with your design, test it for correctness. Because if your design doesn't work, it doesn't make sense to turn it into code. How can you test your design? Pretend you are the computer, follow the steps by hand and see if the results match those determined during the Analysis phase. Or, give the algorithm to a friend and have them step through it.

No particular planning technique will be enforced in this course, but whichever approach you choose, be sure to plan and test your algorithms carefully before you begin writing code -- you'll be glad you did!

6.3 Write the Code

Coding is the process of translating the algorithm into a set of Python statements. Because of the possibility of introducing syntax errors, I recommend developing your programs progressively -- ***add just a few lines at a time, then test the program to make sure it still works***. This means you are always working from a known, good program. If adding just one line causes the program to generate an error, you have a good idea where to look for the problem!

6.4 Test and Debug

Every program needs to be tested! The purpose of **testing** is to prove that the program produces the correct results for given sets of input values. This means ***trying some inputs, then independently confirming that the results are correct*** (by performing calculations by hand, for example). When choosing input values to test, consider both 'typical' data and also any special cases. We will talk more about testing as we work with different kinds of programming problems.

Debugging is the process of locating and correcting any errors ("bugs") in the program. If testing turns up a problem, then the debugging process begins. One of the most important techniques in debugging is the ability to observe the values of variables at different points in the program. Consider adding temporary statements that displays the value of a variable at a place in the code where you suspect a problem. Once the problem is solved, these temporary statements can be removed (or maybe just 'commented out' so they can be used again in the future if needed).

6.5 Documentation

The purpose of documentation is to provide a record of how the problem was solved and how the resulting program works. You can think of documentation falling into two categories: internal and external.

Internal documentation is for people who are going to read your source code. This includes good variable names, named constants, and the comments within the program.

External documentation is for the users of your program. This can include an introduction that the program displays to the user, good prompts, and user manuals.

In this course, we will keep it simple. Your internal documentation should consist of the following:

- A standard block comment at the beginning of the program showing a **brief description** of the program (1 - 2 lines), your **name**, the **assignment number**, and the **date**.
- A small number of 'inline comments' (comments embedded among the lines of code) that provide information for only those things that really need an explanation. Don't include inline comments just for the sake of writing comments. Focus on saying ***why*** something is being done, and not what is being done (which will be obvious to other programmers anyway). Sometimes you don't need any inline comments!

For example, here is a line of code that is poorly commented:

```
x = x + 1 # add one to x
```

Why is this poor? Because it doesn't tell the reader anything new. The reader can look at the code and see that x is being incremented. The question is why.

Here is that same line of code with a better comment:

```
x = x + 1 # another latte sold
```

Now the reader has a better understanding of why x is incremented.

- A block comment at the beginning of a section of code. In the same way I've used headings on this page to keep track of the different sections, you can do that with comments in your code. This type of comment becomes more important as programs get longer (more than 1 page). Some sections might be input, data validation (we'll get to this soon), calculations, and output.
- A block comment at the beginning of each function describing what the function does, the parameters and the return value (if any). We'll revisit this when we learn to write our own functions.

- A block comment at the **end** of the program that shows **documented test cases**: a set of tests performed (inputs used and output produced) and an explanation of why you feel this testing proves that the program is working correctly. Some of the posted examples illustrate this.

The only external documentation we'll write this quarter will be program introductions to the user and helpful input prompts.

7 Errors and Testing

It is normal for errors to occur during the process of writing a program. These errors can be grouped into three categories:

1. A syntax error is present when a statement violates a Python language rule -- e.g. missing quotes around a string literal or missing parentheses in a method call. In general, the browser will signal an error message when a syntax error is encountered. This message does not always diagnose the problem correctly, but usually provides enough information to help you locate the part of the code containing the error. Program execution stops when a syntax error is encountered.

2. A logic error is a flaw in the design of a program. When a program runs with no syntax errors, but does not produce the correct result for the data provided, the problem may be a logic error. There may be

- ***an error in the algorithm being used*** to solve the problem, or there may be
- ***an error in the process of translating the algorithm into program code***.

Either way, locating and correcting logic errors is more difficult, because no error message is provided. That's why it is important to "solve the problem by hand." The programmer must figure out what results the program should produce, and then check that it does produce that result. In fact, **an important part of any programming project is thorough testing** with a variety of input data, because a program may produce correct results for some data but incorrect results in other cases.

In this course, you will **document your testing by adding comments at the end of your program that show the testing performed and the results obtained.** The circle.py program provides an example of this documentation.

3. A run-time error (or run-time exception) is an unexpected problem unrelated to syntax that causes the program to 'crash', or stop running. An example of a run-time error is trying to divide by 0.

Errors in computer programs are often called **bugs**, and the process of identify and correcting them is called **debugging**. These terms are thought to have originated half a century ago when the process involved removing actual bugs (insects) from the electrical switches and wiring of very early computers!

There are no tools that will find your logic errors. Good programmers must have the patience, diligence, and understanding of their code in order to find and fix them.