

DSC 510

WEEK 2 VARIABLES AND MATH OPERATIONS

Python Variables

A variable is a name (identifier) that is associated with a value and is a way of referring to a memory location used by a computer program. A variable is a symbolic name for this physical memory location. This memory location contains values, like numbers, text or more complicated types.

A variable can be seen as a container values. While the program is running, variables are accessed and sometimes changed, i.e. a new value will be assigned to the variable.

One of the main differences between Python and strongly-typed languages like C, C++ or Java is the way it deals with types. In strongly-typed languages every variable must have a unique data type. E.g. if a variable is of type integer, solely integers can be saved in the variable. In Java or C, every variable has to be declared before it can be used. Declaring a variable means binding it to a data type.

Python just like all languages has multiple types of variables that can be used including numbers (integers and floats), strings, and lists.

Integers

Integer numbers (or simply, *integers*) are counting numbers like 1, 2, 3, but also include 0 and negative numbers. The following are examples of data that are expressed as integers:

- Number of people in a room
- Personal or team score in a game
- Course number
- Date in a month
- Temperature (in terms of number of degrees)

Floats

Floating-point numbers (or simply, *floats*) are numbers that have a decimal point in them. The following are examples of data that are expressed as floating-point numbers:

- Grade point average
- Price of something
- Percentages
- Irrational numbers, like pi

Strings

Strings (also called *text*) are any sequences of characters. Examples of data that are expressed as strings include the following:

- Name
- Address
- Course name
- Title of a book, song, or movie
- Sentence
- Name of a file on a computer

Booleans

Booleans are a type of data that can only have one of two values: True or False. Booleans are named after an English mathematician named George Boole, who created an entire field of logic based around these two-state data items. The following are some examples of data that can be expressed as Booleans:

- The state of a light switch: True for on, False for off
- Inside or outside: True for inside, False for outside
- Whether someone is alive or not: True for alive, False for dead
- If someone is listening: True for listening, False for not listening

Python Variables

Declaration of variables is not required in Python. If there is need of a variable, you think of a name and start using it as a variable.

Another remarkable aspect of Python: Not only the value of a variable may change during program execution but the type as well. You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable.

In the following line of code, we assign the value 42 to a variable:

```
i = 42
```

The equal "=" sign in the assignment shouldn't be seen as "is equal to". It should be "read" or interpreted as "is set to", meaning in our example "the variable `i` is set to 42". Now we will increase the value of this variable by 1:

Variable Overview

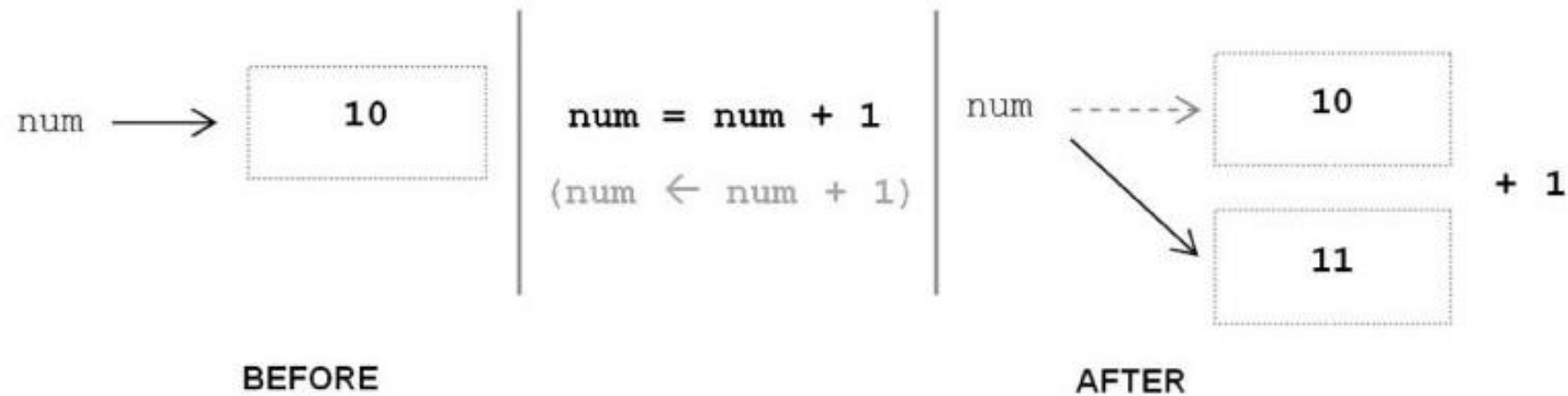
A variable can be assigned different values during a program's execution—hence, the name "variable." Wherever a variable appears in a program (except on the left-hand side of an assignment statement), it is the value associated with the variable that is used, and not the variable's name,

$\text{num} + 1 \rightarrow 10 + 1 \rightarrow 11$

Variables are assigned values by use of the assignment operator, =,
 $\text{num} = 10$ $\text{num} = \text{num} + 1$

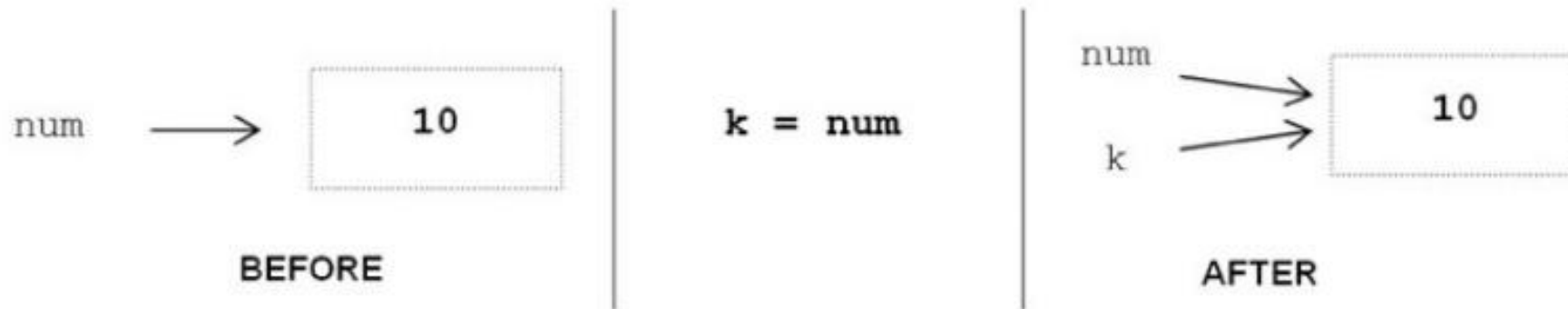
Variable Assignment Statements

Assignment statements often look wrong to novice programmers. Mathematically, $\text{num} = \text{num} + 1$ does not make sense. In computing, however, it is used to increment the value of a given variable by one. It is more appropriate, therefore, to think of the $=$ symbol as an arrow symbol,



Variable Assignment Statements

When thought of this way, it makes clear that the right side of an assignment is evaluated first, then the result is assigned to the variable on the left. An arrow symbol is not used simply because there is no such character on a standard computer keyboard. Variables may also be assigned to the value of another variable (or expression, discussed below) as depicted below.



Variable Assignment Statements Continued...

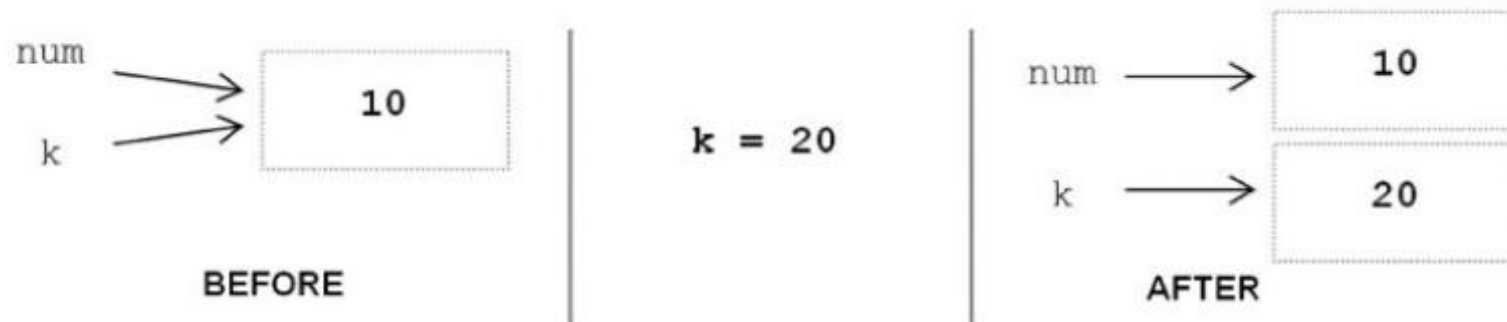
Variables num and k are both associated with the same literal value 10 in memory. One way to see this is by use of built-in function id,

```
>>> id(num)  
505494040
```

```
>>> id(k)  
505494040
```

ID Function

The id function produces a unique number identifying a specific value (object) in memory. Since variables are meant to be distinct, it would appear that this sharing of values would cause problems. Specifically, if the value of num changed, would variable k change along with it? This cannot happen in this case because the variables refer to integer values, and integer values are immutable. An immutable value is a value that cannot be changed. Thus, both will continue to refer to the same value until one (or both) of them is reassigned, as depicted below:



Variable References

If no other variable references the memory location of the original value, the memory location is deallocated (that is, it is made available for reuse).

Finally, in Python the same variable can be associated with values of different type during program execution, as indicated below.

```
var = 12 integer
```

```
var = 12.45 float
```

```
var = 'Hello' string
```

Working with Python Variables

Let's try using a variable in `hello.py`. Add a new line at the beginning of the file, and modify the second line:

```
message = "Hello my name is Mike"  
print(message)
```

Run this program to see what happens. You should see the same output you saw previously:

```
Hello my name is Mike
```

We've added a variable named **message**. Every variable holds a value, which is the information associated with that variable. In this case the value is the text "Hello my name is Mike".

Adding a variable makes a little more work for the Python interpreter. When it processes the first line, it associates the text "Hello my name is Mike" with the variable `message`. When it reaches the second line, it prints the value associated with `message` to the screen.

Identifiers

An identifier is a sequence of one or more characters used to provide a name for a given program element. Variable names `line`, `num_credits`, and `gpa` are each identifiers. Python is case sensitive, thus, `Line` is different from `line`. Identifiers may contain letters and digits, but cannot begin with a digit. Additional rules for defining variable names are provided on the following slide.

Python Variable Naming Rules

- Variable names can contain only letters, numbers, and underscores. They can start with a letter or an underscore, but not with a number. For instance, you can call a variable `message_1` but not `1_message`.
- Spaces are not allowed in variable names, but underscores can be used to separate words in variable names. For example, `greeting_message` works, but `greeting message` will cause errors.
- Avoid using Python keywords and function names as variable names; that is, do not use words that Python has reserved for a particular programmatic purpose, such as the word `print`. (See "Python Keywords and Built-in Functions" on page 489.)
- Variable names should be short but descriptive. For example, `name` is better than `n`, `student_name` is better than `s_n`, and `name_length` is better than `length_of_persons_name`.
- Be careful when using the lowercase letter `l` and the uppercase letter `O` because they could be confused with the numbers `1` and `0`.

Print Statements

The print statement is very "general purpose." You ask it to print something and it prints whatever you ask it to print into the Shell window. The general print statement looks like this:

```
print <whatever you want to see>
```

Here are some examples in the Shell:

```
>>> eggsInOmllette = 3
>>> print eggsInOmllette
3
>>> knowsHowToCook = True
>>> print knowsHowToCook
True
>>>
```

Assignment and Print Statements

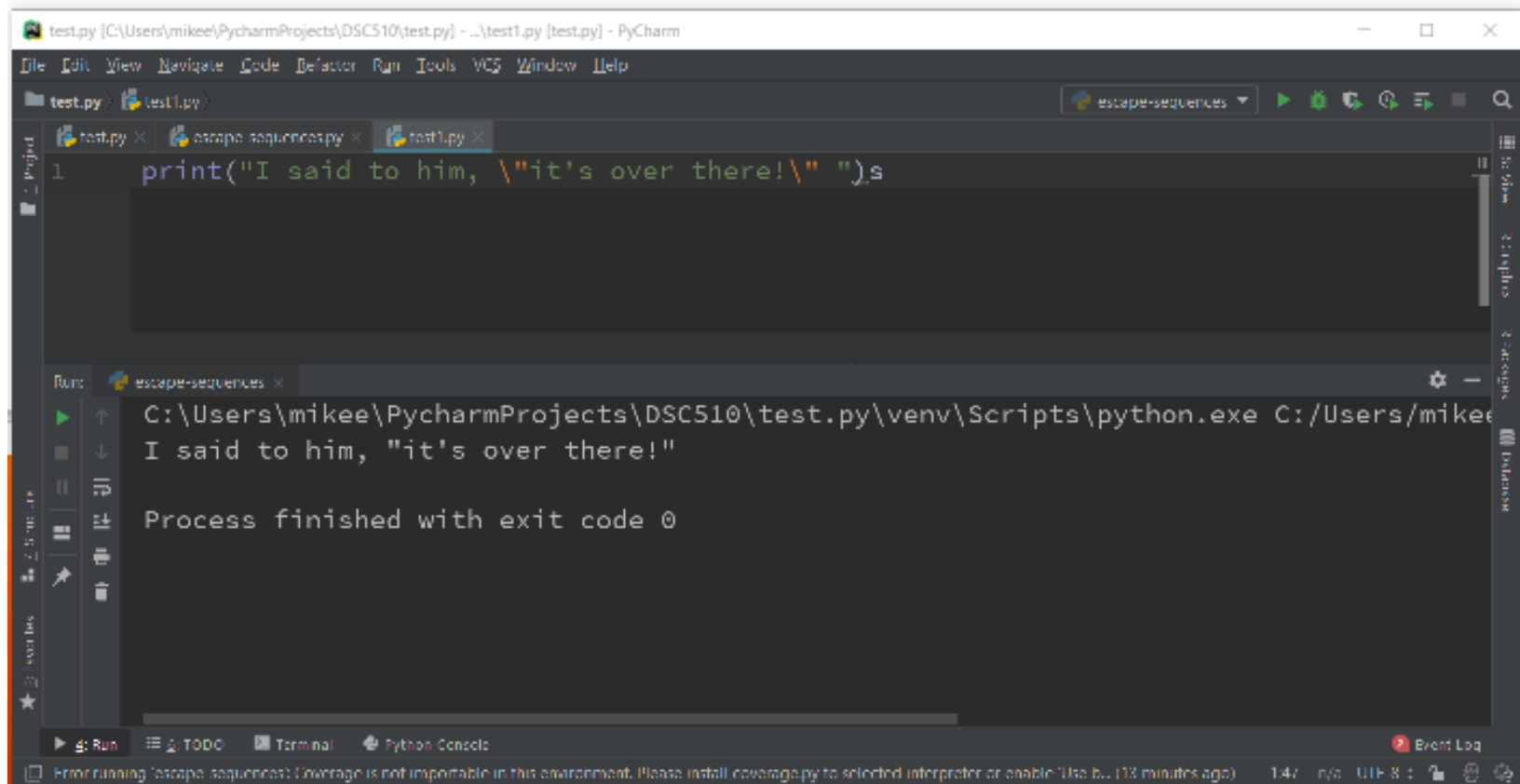
Here are more examples of assignment statements and print statements, using all four types of data:

```
>>> numberInADozen = 12
>>> print 'There are', numberInADozen, 'items in a dozen'
There are 12 items in a dozen
>>> learningPython = True
>>> print 'It is', learningPython, 'that I am learning Python'
It is True that I am learning Python
>>> priceOfCandy = 1.99
>>> print 'My candy costs', priceOfCandy
My candy costs 1.99
>>> myFullName = 'Irv Kalb'
>>> print 'My full name is', myFullName
My full name is Irv Kalb
>>>
```

Escape Sequences

- Escape sequences are useful in order to print specific pieces of data which would otherwise not be interpreted correctly when included within a printed string.
- `\\` Backslash (`\`)
- `'` Single- quote (`'`)
- `"` Double- quote (`"`)
- `\a` ASCII bell (BEL)
- `\b` ASCII backspace (BS)
- `\f` ASCII formfeed (FF)
- `\n` ASCII linefeed (LF)
- `\N{name}` Character named name in the Unicode database (Unicode only)
- `\r` ASCII carriage return (CR)
- `\t` ASCII horizontal tab (TAB)
- `\uxxxx` Character with 16- bit hex value xxxx (Unicode only)
- `\Uxxxxxxxx` Character with 32- bit hex value xxxxxxxx (Unicode only)
- `\v` ASCII vertical tab (VT)
- `\ooo` Character with octal value oo
- `\xhh` Character with hex value hh

Escape Sequence Example (Printing Double Quotes)



The image shows a PyCharm IDE window with a Python file named `test.py`. The code in the editor is:

```
1 print("I said to him, \"it's over there!\" ")
```

Below the editor, the Run tool window is open, showing the output of the script:

```
C:\Users\mikee\PycharmProjects\DSC510\test.py\venv\Scripts\python.exe C:/Users/mikee
I said to him, "it's over there!"
Process finished with exit code 0
```

The output correctly displays the double quotes around the phrase "it's over there!".

Print Statement in Python 2 vs Python 3

In Python 3, the print statement has a different form (syntax). In Python 3, the print statement must have parentheses around the item(s) that you want to print, as follows: `print (<item1>, <item2>, ...)` This is perhaps the most noticeable difference between Python 2 and Python 3. If you see code elsewhere written in Python 3 that uses parentheses in print statements, you can often modify these statements to work in Python 2 by removing the outermost set of parentheses.

Math Operators

Now let's move on to some simple math for use in assignment statements. Python and all computer languages include the following standard set of math operators:

- + Add
- – Subtract
- / Divide
- * Multiply
- ** Raise to the power of
- % Modulo (also known as remainder)
- () Grouping (we'll come back to this)

Simple Math

Let's try some very simple math. For demonstration purposes, I'll just use variables named x and y. In the Shell, try the following:

```
>>> x=9
>>> y=6
>>> print x + y
15
>>> print x - y
3
>>> print x * y
54
>>> print x / y
1
>>>
```

Division in Python 2 vs Python 3

In Python 3, the division operator works differently. In Python 3, if you use the slash to do a division, you will always get a floating-point answer. If you want to do an explicit integer division, you must use a new operator; two slashes, for example:

```
forcedIntegerAnswer = integer1 // integer2
```

Division with Floats

As humans, we represent integers using base 10 (digits from 0 to 9). Computers represent integers using base 2 (using only ones and zeroes). But there is an exact mapping between the two bases. For every base 10 number, there is an exactly equivalent base 2 number. However, because of the way that computers represent floating-point numbers, this is not the case for floating-point numbers. There is no such mapping between base 10 fractions and base 2 fractions. When representing floating-point fractional numbers, there is often some small amount of "rounding"; that is, floating-point fractional numbers are a close approximation of the intended number. For example, if we attempt to divide 5.0 by 9.0, we see this:

```
>>> print 5.0 / 9.00.555555555556
```

The decimal values goes on forever, but when represented as a float, the value gets rounded off.

Comments

When you are writing software, you wind up making a lot of decisions about how you approach different problems. Sometimes, your solutions are not exactly apparent and could use some documentation. You may want to explain to the reader (who could be a future version of you, or someone else) why you did something the way you wound up doing it, or how some intricate piece of code works.

Documentation like this, written directly in your code, is called a *comment*. Comments are completely ignored by Python; they are only there for humans. There are three ways to write comments in Python: provide a full-line comment, add a comment after a line of code, or use a multiline comment.

Full-Line Comments

Full-Line Comment

Start a line with the # character, followed by your comment:

```
# This whole line is a comment
# This is another comment line
# All comment lines are ignored by Python
# Even though the next line looks like code, it's just a comment
# x=1
# -----
```

Comments after a Line of Code

Add a Comment Alter a Line of Code

You can put a comment at the end of a line of code to explain what is going on inside that line. The following lines are very simple and don't really need comments, but they should serve as a good example of how to add this type of comment:

```
score = 0 # Initializing the score  
priceWithTax = price * 1.09 # add in 9% tax
```

Multiline Comments

Multiline Comment

You can create a comment that spans any number of lines. You do this by having one line with three quote marks (single or double quotes), any number of comment lines, and ending with the same three quote characters (single or double quotes), as follows:

```
'''
```

A multiline comment starts with a line of three quote characters(above) This is a long comment block It can be any length You do not need to use the # character here You end it by entering the same three quotes you used to start (below)

```
'''
```

Getting User Input

This is the typical flow of a simple computer program:

1. Input data.
2. Work with data.
3. Do some computation(s).

Output some answer(s). In Python, we can get input from the user using a built-in function called `raw_input`. Here's how it is used, most typically in an assignment statement:

```
<variable> = input(<prompt string>)
```

On the right side of the equals sign is the call to the `raw_input` built-in function. When you make the call, you must pass in a *prompt string*, which is any string that you want the user to see. The prompt is a question that you want the user to answer. The `raw_input` function returns all of the characters that the user types, as a string. Here is an example of how `raw_input` might be used in a program:

```
favoriteColor = input('What is your favorite color? ')  
print 'Your favorite color is', favoriteColor
```

Assignment Statement

When the assignment statement runs, the following steps happen in order:

1. The prompt string is printed to the Shell.
2. The program stops and waits for the user to type a response.
3. The user enters some sequence of characters into the Shell as an answer.
4. When the user presses the Enter key (Windows) or the Return key (Mac), `input()` returns the characters that the user typed.
5. Typically, `input()` is used on the right side of an assignment statement. The user's response is stored into the variable on the left-hand side of the equals sign.