lecture_2

February 15, 2017

1 Beginning Python Class -- Lecture 2

1.1 Expected Understanding

At this point in the class it is very important that you have a good understanding of the topics in Lecture_1. The topics today will build directly on the Basic Data Types. - Int/Float - Bool - String - List/Tuple - Dictionary Today's topics will allow our code to execute in a way that is not directly linear. We will cover Conditionals and Looping. Conditionals will allow us to make decisions in our code and only execute specific parts. Loops will give us a quick way to execute multiple instructions over and over without having to directly write out every instruction.

More information can be found on these topics here: `http://www.openbookproject.net/books/bpp4awd/ch04.html`

Indentation and Syntax will be increasingly important this week and therefore I will leave the sections about `Style` and `Syntax` in this lecture for easy reference.

1.1.1 Running Python in the Lab

Not everyone has a dedicated Windows machine that they can remote into, and not everyone has their own Linux account. Therefore we are going to use an online Python Interpreter to run our Python code for this class. This way we can practice in the lab and students can practice at home without having to install python on their PC. I will run python from my machine during the lecture so that the process of running python from the command line is understood.

Here is the link to the online Python Interpreter -- `https://repl.it/languages/python3`

1.1.2 PEP8 Style Guide

PEP8 style guide should be used as a reference for styling decisions. This document (`http://legacy.python.org/dev/peps/pep-0008/`) covers topics like: - Indentation - Comments - Line Length These are a set of standards that allow python developers to easily work on each others code and avoid nasty syntax bugs caused by white space inconsistencies.

1.1.3 Syntax

Syntax in the context of computer code is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language.

Python was designed to have a very clean look to the code. The compromise is that to correctly interpret the code python elected to use white space as a syntactical character. The last language to use syntactical white space was Fortran which was created by IBM in 1957.

Languages such as Perl and C ignore white space and rely on other special characters that often mean different things in different contexts. This allowed the programmer to develop their own style independent of the format of the code.

A quick example is shown below:

```
foreach my $a (@list_of_numbers){
    print "value of a: $a\n";
}
```

The exact same Perl code could be written like:

```
foreach my $a (@list_of_numbers){ print "value of a: $a\n"; }
```

This is because one of the characters that Perl uses to group instructions are "{}" (brackets).

Python avoids this type of problem by requiring white space syntax to group sequences of instruction:

```
for a in list_of_numbers:
    print( "Value of a: " + str(a))
```

Since the 'print' statement is executed during the loop it is nested/grouped inside the loop with 4 spaces. (More on loops later in Lecture 2)

1.2 Conditional Statements

At the heart of every `if` statement is an expression that can be evaluated as `True` or `False` and is called a *conditional test*. Python uses the values `True` and `False` to decide whether the code in an `if` statement should be executed. If a conditional test evaluates to `True`, Python executes the code following the `if` statement. If the test evaluates to `False`, Python ignores the code following the `if` statement.

1.2.1 Conditional Statements/Boolean Expressions

Conditional Statements check a condition and then either return `True` or `False`. These statements can be thought of as a way to ask a question that is either answered 'yes' or 'no'. Conditional Statements are the same as Boolean Expressions and are interchangeable vocabulary. Sometimes the evaluation of Conditional Statements is called Boolean Algebra.

Equality This Conditional statement checks for equality and returns `True` or `False`. To check for equality we use the == operator.

```
In [1]: print( 5 == 5 )
        print( 5 == 6 )
        print( 5 == 'a' )
        print( 5 == [5] )
```

```
True
False
False
False
```

Inequality Inequality returns `True` if the values compared are not equal. Inequality can be denoted in two ways. Either by using the inequality operator `!=` pronounced (Bang Equals) or by negating an equality statement.

```
In [2]: print( 5 != 5 )
        print( 5 != 6 )
        print( 5 != 'a' )
        print( 5 != [5] )
        # negation turns True to False, and False to True.
        # the 'not' key word is used to denote negation.
        print( "Now we are using negation")
        print( not( 5 == 5 ) )
        print( not( 5 == 6 ) )
```

```
False
True
True
True
Now we are using negation
False
True
```

Numerical Comparisons Numerical Comparisons are used to know how a number compares to another number. There are a number of operators that we will be using: - > Greater-than - < Less-than - >= Greater-than-or-Equal-to - <= Less-than-or-Equal-to It should also be noted that some of these operators can be used on strings and characters. But just as + works differently on numbers than strings, these operators also work differently.

```
In [3]: print( 5 > 10 )
        print( 5 < 10 )
        print( 5 > 5 )
        print( 5 >= 5 )
        print( 5 < 5 )
        print( 5 <= 5 )
```

```
False
True
False
True
False
True
```

Containment or the `in` Operator The `in` key word checks if an item is contained in a sequential data type (string, list, tuple, dictionary).

```
In [4]: print( 'a' in "Dallin" )
        print( 'b' in "Dallin" )
        print( 1 in [0,1,2,3,4,5] )
        print( -1 in [0,1,2,3,4,5] )
```

```
True
False
True
False
```

Multiple Conditions Multiple Conditions can be checked at the same time using the keywords `'and'` and `'or'`. - `and` gives us `True` if statements on both sides of the `and` are `True` otherwise it gives us `False` - `or` gives us `True` if the statements on either side of the `or` is `True`

```
In [1]: x = 7
        print( 5 <= x and x < 10 )

        x = -100
        print( x < 0 or x > 10)
```

```
True
True
```

1.2.2 If Statements

If statements are paired with a conditional statement and do something if the statement is `True` and something different if the statement evaluates to `False`.

Pieces of code that are nested together are sometimes called 'blocks'. In order for statements to be included in the `if` statement they must *indented* by 4 spaces. `if` statements are written in the following form.

```
if BOOLEAN EXPRESSION:
    STATEMENTS # 4 spaces of white space
    STATEMENTS # 4 spaces of white space
```

Please note the white space that nests the statements inside the `if` statement. This is piece of code could be considered a block.

```
In [6]: x = 3
        if x in [0,1,2,3,4,5]:
            print( "x is a good number" )

        y = 7
        if y in [0,1,2,3,4,5]:
            print( "y is a good number") # This will not print
```

```
x is a good number
```

A more general form of the `if` statement is shown below. This will be a good way to chain statements together.

```
if BOOLEAN EXPRESSION:
    TRUE STATEMENTS # 4 spaces of white space
    TRUE STATEMENTS # 4 spaces of white space
elif DIFFERENT BOOLEAN EXPRESSION:
    DIFFERENT TRUE STATEMENTS # 4 spaces of white space
    DIFFERENT TRUE STATEMENTS # 4 spaces of white space
elif DIFFERENT BOOLEAN EXPRESSION:
    DIFFERENT TRUE STATEMENTS # 4 spaces of white space
    DIFFERENT TRUE STATEMENTS # 4 spaces of white space
else:
    FALSE STATEMENTS # 4 spaces of white space
    FALSE STATEMENTS # 4 spaces of white space
```

There is no limit to the number of `elif` statements that can be added to an `if` statement.

```
In [7]: score = 88
        if score >= 90:
            print( "You got an A" )
        elif score >= 80:
            print( "You got a B" )
        elif score >= 70:
            print( "You got a C" )
        elif score >= 60:
            print( "You got a D" )
        else:
            print( "You Failed" )
```

```
You got a B
```

1.3 Looping

Looping is a control flow statement for specifying iteration/repetition, this allows code to be executed repeatedly. In other words loops are a way that we can move back up and execute earlier lines of Python. It is possible to write loops that will loop forever. These are called infinite loops. Running the loop below would result in the value 10 being printed forever. Please note the difference between this loop and the first loop in the `while` section. Infinite loops should be avoided except under special conditions.

```
x = 10
while x != 0:
    print( "x is equal to: " + str(x) )
```

1.3.1 While Loops

A `while` loop is the most basic loop structure. It will continue to execute the statements in the `while` loop while the Boolean Expression is `True`. The Boolean Expression is checked each time through the loop.

```
while BOOLEAN EXPRESSION:
    TRUE STATEMENTS # 4 spaces of white space
    TRUE STATEMENTS # 4 spaces of white space
```

The key with `while` loops is that eventually the Boolean Expression should eventually evaluate to `False` to end the loop. This may require the user to create a situation that will terminate the loop. When writing loops keep in mind the situation that will cause the loop to terminate.

```
In [8]: x = 10
        while x != 0:
            print( "x is equal to: " + str(x) )
            x = x - 1 # Notice that I modify the value of x in the loop
```

```
x is equal to: 10
x is equal to: 9
x is equal to: 8
x is equal to: 7
x is equal to: 6
x is equal to: 5
x is equal to: 4
x is equal to: 3
x is equal to: 2
x is equal to: 1
```

1.3.2 For Loops

A `for` loop is the most common loop in Python. It is most commonly used to do something on each item of a Sequential Data Type. Once the last item is reached in the Sequential Data Type the loop is exited and Python executes the next line after the loop. The basic form of the `for` loop is shown below.

```
for x in SEQUENTIAL_DATA_TYPE:
    STATEMENTS # 4 spaces of white space
```

In the example below the variable `x` gets assigned to each subsequent value in `[0,1,2,3,4,5]`. Therefore the first time through the loop `x = 0` and the next time through `x = 1`. This will continue until the last time through the loop where `x = 5`.

```
In [9]: for x in [0,1,2,3,4,5]:
            print( "The value of x is: " + str(x) )
```

```
The value of x is: 0
The value of x is: 1
The value of x is: 2
The value of x is: 3
The value of x is: 4
The value of x is: 5
```

1.4 Nested Code Blocks

Many of you may have already guessed but blocks of code can be nested as many times as desired in Python. For example you could add an `if` statement inside a `for` loop. This is a very important application of white space syntax. Pay attention to the comments explaining the white space.

```
In [10]: for x in 'abcdefghijklmnopqrstuvwxyz':
             if x in 'aeiou': # 4 spaces
                 print( x + " is a vowel") # 8 spaces
             else: # 4 spaces
                 print( x + " is a consonant") # 8 spaces
```

```
a is a vowel
b is a consonant
c is a consonant
d is a consonant
e is a vowel
f is a consonant
g is a consonant
h is a consonant
i is a vowel
j is a consonant
k is a consonant
l is a consonant
m is a consonant
n is a consonant
o is a vowel
p is a consonant
q is a consonant
r is a consonant
s is a consonant
t is a consonant
u is a vowel
v is a consonant
w is a consonant
x is a consonant
y is a consonant
z is a consonant
```

1.5 Practice Problem 1

For every value from 0 - 19 check whether the number is even or odd and print it out. Use a for-loop to iterate through the values.

Example: - 0 is even - 1 is odd - 2 is even - 3 is odd

Extra Credit: There is an operator that will help you complete this problem. It is the modulus operator `%`. This operator is similar to division except that it only retains the remainder. See the example below.

```
In [11]: print( 10 % 3 ) # 10/3 divides 3 times with 1 remainder
         print( 1275 % 100 ) # 1275/100 divides 12 times with a remainder of 75
```

```
1
75
```

1.6 Practice Problem 2

Create a loop that will ask the user to guess a number from 1-100, and tell the user if the guess is too low, too high, or correct. When the correct guess is found please end the script.

Example: - Please guess a number (1-100): 65 - 65 is too high - Please guess a number (1-100): 20 - 20 is too low - Please guess a number (1-100): 41 - 41 is Correct

Extra Credit: You can write the above problem with a hard coded value to guess or you can make use of the following snippet of code to get a different number each time the script is ran.

```
In [5]: import random
        x = random.randint(1,100) # Give me a random integer number from 1 to 100
        y = random.randint(1,100) # Give me a random integer number from 1 to 100, Likely not th

        print( x )
        print( y )
```

```
74
37
```

1.7 Practice Problem 3

Create a program that will allow you to quiz small children on their multiplication tables where the numbers can be from 0-12. This program should ask 5 multiplication questions from the user and keep track of the number right and the number wrong. At the end of the five questions tell the user the overall score. You may use the same `random.randint` function that was introduced in 'Practice Problem 2' to make your script more interesting.

Example: - Problem Number 1. What is the answer to 4*8: 32 - Correct! - Problem Number 2. What is the answer to 7*1: 8 - Incorrect! - Problem Number 3. What is the answer to 10*11: 110 - Correct! - Problem Number 4. What is the answer to 0*2: 0 - Correct! - Problem Number 5. What is the answer to 3*6: 18 - Correct! - You got 4 out of 5 correct. Thanks for playing.