Python Review 1

Jay Summet
CS 1803

Outline

- Introduction to Python
- Operators & Expressions
- Data Types & Type Conversion
- Variables: Names for data
- Functions
- Program Flow (Branching)
- Input from the user
- Iteration (Looping)

Introduction to Python

- Python is an interpreted programming language
- A program is a set of instructions telling the computer what to do.
- It has a strict syntax, and will only recognize very specific statements. If the interpreter does not recognize what you have typed, it will complain until you fix it.

Operators

- Python has many operators. Some examples are:

`+, -, *, /, %, >, <, ==`

`print`

- Operators perform an action on one or more operands. Some operators accept operands before and after themselves:

`operand1 + operand2`, or `3 + 5`

- Others are followed by one or more operands until the end of the line, such as: `print "Hi!", 32, 48`
- When operators are evaluated, they perform action on their operands, and produce a new value.

Example Expression Evaluations

- An expression is any set of values and operators that will produce a new value when evaluated. Here are some examples, along with the new value they produce when evaluated:

```
 5 + 10                    produces   15
"Hi" + " " + "Jay!"        produces   "Hi Jay!"
10 / (2+3)                 produces   2
10 > 5                     produces   True
10 < 5                     produces   False
10 / 3.5                   produces   2.8571428571
10 // 3                    produces   3
10 % 3                     produces   1
```

List of Operators: +, -, *, /, <, >, <=, >=, ==, %, //

- Some operators should be familiar from the world of mathematics such as Addition (+), Subtraction (-), Multiplication (*), and Division (/).
- Python also has comparison operators, such as Less-Than (<), Greater-Than (>), Less-Than-or-Equal(<=), Greater-Than-or-Equal (>=), and Equality-Test (==). These operators produce a True or False value.
- A less common operator is the Modulo operator (%), which gives the remainder of an integer division. 10 divided by 3 is 9 with a remainder of 1:

`10 // 3` produces 3, while `10 % 3` produces 1

DANGER! Operator Overloading!

- NOTE! Some operators will work in a different way depending upon what their operands are. For example, when you add two numbers you get the expected result: `3 + 3` produces 6.
- But if you "add" two or more strings, the + operator produces a concatenated version of the strings: `"Hi" + "Jay"` produces "HiJay"
- Multiplying strings by a number repeats the string!

`"Hi Jay" * 3` produces "Hi JayHi JayHiJay"

- The % sign also works differently with strings:

`"test %f" % 34` produces "test 34"

Data Types

- In Python, all data has an associated data “Type”.
- You can find the “Type” of any piece of data by using the type() function:

`type( “Hi!”)` produces <type 'str'>

`type( True )` produces <type 'bool'>

`type( 5)` produces <type 'int'>

`type(5.0)` produces <type 'float'>

- Note that python supports two different types of numbers, Integers (int) and Floating point numbers (float). Floating Point numbers have a fractional part (digits after the decimal place), while Integers do not!

Effect of Data Types on Operator Results

- Math operators work differently on Floats and Ints:
  - int + int produces an int
  - int + float or float + int produces a float
- This is especially important for division, as integer division produces a different result from floating point division:

`10 // 3` produces 3

`10 / 3` produces 3.3333

`10.0 / 3.0` produces 3.3333333

- Other operators work differently on different data types: + (addition) will add two numbers, but concatenate strings.

Simple Data types in Python

The simple data types in Python are:

- Numbers
  - int – Integer: -5, 10, 77
  - float – Floating Point numbers: 3.1457, 0.34
- bool – Booleans (True or False)
- Strings are a more complicated data type (called Sequences) that we will discuss more later. They are made up of individual letters (strings of length 1)

Type Conversion

- Data can sometimes be converted from one type to another. For example, the string “3.0” is equivalent to the floating point number 3.0, which is equivalent to the integer number 3
- Functions exist which will take data in one type and return data in another type.
  - `int()` - Converts compatible data into an integer. This function will truncate floating point numbers
  - `float()` - Converts compatible data into a float.
  - `str()` - Converts compatible data into a string.
- Examples:

`int(3.3)` produces 3 `str(3.3)` produces “3.3”
`float(3)` produces 3.0 `float(“3.5”)` produces 3.5
`int(“7”)` produces 7
`int(“7.1”)` throws an ERROR!
`float(“Test”)` Throws an ERROR!

Variables

- Variables are names that can point to data.
- They are useful for saving intermediate results and keeping data organized.
- The assignment operator (=) assigns data to variables.
  - Don't confuse the assignment operator (single equal sign, =) with the Equality-Test operator (double equal sign, ==)
- Variable names can be made up of letters, numbers and underscores (_), and must start with a letter.

Variables

- When a variable is evaluated, it produces the value of the data that it points to.
- For example:

`myVariable = 5`

`myVariable` produces 5

`myVariable + 10` produces 15

- You MUST assign something to a variable (to create the variable name) before you try to use (evaluate) it.

Program Example

Find the area of a circle given the radius:

```
Radius = 10
pi = 3.14159
area = pi * Radius * Radius
print( area )
```

will print 314.15 to the screen.

Code Abstraction & Reuse Functions

- If you want to do something (like calculate the area of a circle) multiple times, you can encapsulate the code inside of a *Function*.
- A Function is a named sequence of statements that perform some useful operation. Functions may or may not take parameters, and may or may not return results.

Syntax:

```
def NAME( LIST OF PARAMETERS):
    STATEMENTS
    STATEMENTS
```

How to use a function

- You can cause a function to execute by “calling” it as follows:

```
functionName( Parameters)
```

- You can optionally assign any result that the function returns to a variable using the assignment operator:

```
returnResult = functionName(Parameters)
```

Indentation is IMPORTANT!

- A function is made up of two main parts, the Header, and the Body.
- The function header consists of:
  `def funcName(param1,param2):`
  - def keyword
  - function name
  - zero or more parameters, comma separated, inside of parenthesis ()
  - A colon :
- The function body consists of all statements in the block that directly follows the header.
- A block is made up of statements that are at the same indentation level.

findArea function naive example

```
def findArea( ):
    Radius = 10
    pi = 3.1459
    area = pi * Radius * Radius
    print(area)
```

- This function will ONLY calculate the area of a circle with a radius of 10!
- This function will PRINT the area to the screen, but will NOT return the value pointed to by the area variable.

findArea function, with syntax error!

```
def findArea( ):
    Radius = 10
   pi = 3.1459
  area = pi * Radius * Radius
    print(area)
```

- You can NOT mix indentation levels within the same block! The above code will result in a syntax error!

What's wrong with findArea – Limited Applicability

```
def findArea( ):
    Radius = 10
    pi = 3.1459
    area = pi * Radius * Radius
    print(area)
```

- It will only work for circles of size 10!
- We need to make this function more general!
- Step 1: Use parameters to accept the radius of any sized circle!

findArea function better example

```
def findArea( Radius ):
    pi = 3.1459
    area = pi * Radius * Radius
    print( area)
```

- This function will work with any sized circle!
- This function will PRINT the area to the screen, but will NOT return the value pointed to by the area variable.

What's wrong with findArea

- `findArea(10)` prints 314.59 to the screen
- `findArea(15)` prints 707.8275 to the screen
- `myArea = findArea(10)` will assign “None” to the myArea variable. (Due to the lack of an explicit return statement, the function only prints the value, and does not return it.)
- We need to make this function return the value it calculates!
- Step 2: Use a return statement to return the calculated area!

findArea function best example

```
def findArea( Radius ):
    pi = 3.1459
    area = pi * Radius * Radius
    return area
```

- This function will work with any sized circle!
- This function will return the area found, but will NOT print it to the screen. If we want to print the value, we must print it ourselves:

```
circleArea = findArea(15)
print circleArea
```

- Note the use of the circleArea variable to hold the result of our findArea function call.

Keywords, Name-spaces & Scope

- In Python, not all names are equal.
- Some names are reserved by the system and are already defined. Examples are things like: def, print, if, else, while, for, in, and, or, not, return. These names are built in keywords.
- Names that are defined in a function are “local” to that function.
- Names that are defined outside of a function are “global” to the module.
- Local names overshadow global names when inside the function that defined them.
- If you want to access a global variable from inside of a function, you should declare it “global”.

Global vs Local example

```
myVariable = 7
myParam = 20

def func1(myParam):
 myVariable = 20
 print(myParam)

func1(5)
print(myVariable)
```

- What gets printed? 5 and 7
- The “local” myVariable inside func1 is separate from (and overshadows) the “global” myVariable outside of func1
- The “local” myParam inside func1 is different from the “global” myParam defined at the top.

Global vs Local example – part 2

```
myVariable = 7
myParam = 20

def func1(myParam):
 global myVariable
 myVariable = 20
 print(myParam)

func1(5)
print(myVariable)
```

- What gets printed? 5 and 20
- The “local” myVariable inside func1 is separate from the “global” myVariable outside of func1
- The function assigns 20 to the “global” myVariable, overwriting the 7 before it gets printed.

Making Decisions – Controlling Program Flow

- To make interesting programs, you must be able to make decisions about data and take different actions based upon those decisions.
- The IF statement allows you to conditionally execute a block of code.
- The syntax of the IF statement is as follows:

```
if  boolean_expression :
    STATEMENT
    STATEMENT
```

- The indented block of code following an if statement is executed if the boolean expression is true, otherwise it is skipped.

IF statement - example

```
numberOfWheels = 3
if ( numberOfWheels < 4):
    print(“You don't have enough wheels!”)
    print(“I'm giving you 4 wheels!”)
    numberOfWheels = 4

print(“You now have”, numberOfWheels,
  “wheels”)
```

- The last print statement is executed no matter what. The first two print statements and the assignment of 4 to the numberOfWheels is only executed if numberOfWheels is less than 4.

IF/ELSE

- If you have two mutually exclusive choices, and want to guarantee that only one of them is executed, you can use an IF/ELSE statement. The ELSE statement adds a second block of code that is executed if the boolean expression is false.

```
if  boolean_expression :
    STATEMENT
    STATEMENT
else:
    STATEMENT
    STATEMENT
```

IF/ELSE statement - example

```
numberOfWheels = 3
if ( numberOfWheels < 3):
    print("You are a motorcycle!")
else:
    print("You are a Car!")

print("You have", numberOfWheels, "wheels")
```

- The last print statement is executed no matter what. If numberOfWheels is less than 3, it's called a motorcycle, otherwise it's called a car!

IF/ELIF/ELSE

- If you have several mutually exclusive choices, and want to guarantee that only one of them is executed, you can use an IF/ELIF/ELSE statements. The ELIF statement adds another boolean expression test and another block of code that is executed if the boolean expression is true.

```
if  boolean_expression :
    STATEMENT
    STATEMENT
elif 2nd-boolean_expression ):
    STATEMENT
    STATEMENT
else:
    STATEMENT
    STATEMENT
```

IF/ELSE statement - example

```
numberOfWheels = 3
if ( numberOfWheels == 1):
    print(“You are a Unicycle!”)
elif (numberOfWheels == 2):
    print(“You are a Motorcycle!”)
elif (numberOfWheels == 3):
    print(“You are a Tricycle!”)
elif (numberOfWheels == 4):
    print(“You are a Car!”)
else:
    print(“That's a LOT of wheels!”)
```

- Only the print statement from the first true boolean expression is executed.

IF/ELSE statement – example – Semantic error!

```
numberOfWheels = 3
if ( numberOfWheels == 1):
    print(“You are a Unicycle!”)
elif (numberOfWheels > 1 ):
    print(“You are a Motorcycle!”)
elif (numberOfWheels >  2):
    print(“You are a tricycle!”)
elif (numberOfWheels > 3):
    print(“You are a Car!”)
else:
    print(“That's a LOT of wheels!”)
```

- What's wrong with testing using the greater-than operator?

Getting input from the User

- Your program will be more interesting if we obtain some input from the user.
- But be careful! The user may not always give you the input that you wanted, or expected!
- A function that is useful for getting input from the user is:

`input(<prompt string>)` - always returns a string

- You must convert the string to a float/int if you want to do math with it!

Input Example – possible errors from the input() function

```
userName = input("What is your name?")
userAge = int( input("How old are you?") )
birthYear = 2007 - userAge

print("Nice to meet you, " + userName)
print("You were born in: ", birthYear)
```

- `input()` is guaranteed to give us a string, no matter WHAT the user enters.
- But what happens if the user enters "ten" for their age instead of 10?

Input Example – possible errors from the input() function

```
userName = raw_input("What is your name?")
userAge = input("How old are you?")
try:
   userAgeInt = int(userAge)
except:
   userAgeInt = 0
birthYear = 2010 - userAgeInt

print("Nice to meet you, " + userName)
if userAgeInt != 0:
    print("You were born in: ", birthYear )
```

- The try/except statements protects us if the user enters something other than a number. If the int() function is unable to convert whatever string the user entered, the except clause will set the userIntAge variable to zero.

Repetition can be useful!

- Sometimes you want to do the same thing several times.
- Or do something very similar many times.
- One way to do this is with repetition:

```
print 1
print 2
print 3
print 4
print 5
print 6
print 7
print 8
print 9
print 10
```

Looping, a better form of repetition.

- Repetition is OK for small numbers, but when you have to do something many, many times, it takes a very long time to type all those commands.
- We can use a loop to make the computer do the work for us.
- One type of loop is the “while” loop. The while loop repeats a block of code until a boolean expression is no longer true.
- Syntax:

```
while boolean expression :
    STATEMENT
    STATEMENT
    STATEMENT
```

How to STOP looping!

- It is very easy to loop forever:

```
while  True :
    print( “again, and again, and again”)
```

- The hard part is to stop the loop!
- Two ways to do that is by using a loop counter, or a termination test.
  - A loop counter is a variable that keeps track of how many times you have gone through the loop, and the boolean expression is designed to stop the loop when a specific number of times have gone bye.
  - A termination test checks for a specific condition, and when it happens, ends the loop. (But does not guarantee that the loop will end.)

Loop Counter

```
timesThroughLoop = 0

while (timesThroughLoop < 10):
    print("This is time", timesThroughLoop,
          "in the loop.")
    timesThroughLoop = timesThroughLoop + 1
```

- Notice that we:
  - Initialize the loop counter (to zero)
  - Test the loop counter in the boolean expression (is it smaller than 10, if yes, keep looping)
  - Increment the loop counter (add one to it) every time we go through the loop
- If we miss any of the three, the loop will NEVER stop!

While loop example, with a termination test

- Keeps asking the user for their name, until the user types “quit”.

```
keepGoing = True
while ( keepGoing):
    userName = input(“Enter your name! (or
  quit to exit)” )
    if userName == “quit”:
        keepGoing = False
   else:
       print(“Nice to meet you, “ + userName)

print(“Goodbye!”)
```

The End!

- Next up – Python Review 2 – Compound Data Types and programming tricks..