Python as a First Programming Language

STEMBOPS

Davidson Academy of Nevada
University of Nevada, Reno

Supported by NSF grant #1301726

Author:
Justin Stevens
Giselle Serate

Supervisor:
Richard Kelley

March 6, 2016

1 Introduction

Python is a powerful, high-level programming language. It is simple to read and easy to understand, since it hides many details from users. It also eliminates potentially confusing characters like brackets and semicolons, as used in programming languages such as C, C++, Java, and Javascript. Programming errors tend to be easy to find because of Python's readability. This makes Python advantageous for beginning programmers to learn.

Interestingly enough, some of the most meaningful characters in Python are whitespace. Python uses them to denote what is contained in loops, functions, or conditional statements. Programs in Python must be written in a specifically indented format that makes it easy to understand the logic that's being used. Many other languages largely ignore whitespace, making indenting a convention for humans to rely on that doesn't necessarily translate to what the computer understands.

Python is an interpreted language, which means that the computer understands the program by reading it line by line. Because of that, it's slow compared to compiled languages, where programs are reduced to a set of basic computer instructions and then run. This sacrifice of speed is the main drawback to Python. However, in many applications, the speed difference doesn't significantly affect the execution and basic function of the program.

Example 1.1. Only one line is needed for this simple print program, which is a typical first program for a Python user:

Code 1.1

```
print "Hello, World!"
```

Output 1.1

```
Hello, World!
```

More complex programs can define functions, which are essentially small parts of a program that can be run repeatedly in different situations. For example, a program might need to check if numbers in a set are positive and less than five. Defining a function named checkNumber and calling it each time would be more efficient than manually checking if the number is positive and also less than five each time.

Example 1.2. In this program, we define a function, add, which takes two numbers, adds them, and returns the sum. Then we call the add function with the numbers 2 and 3 and get 5. We can call add as many times as we'd like. As this is a very simple function, it doesn't save much time or space, but functions become useful as the tasks they must do become more complex.

Code 1.2

```
def add(x,y):
    answer = x + y
    return answer
print add(2,3)
```

Output 1.2

```
5
```

2 Variables

Variables in Python do not need to be declared, and the interpreter will determine which data type to store the variable as. In other languages, this is something that the user would have to assign manually. Some common data types are integers (counting numbers), floats (numbers with decimal places), strings (sequences of characters), and booleans (True or False). Using the same data type when comparing is very important, because the computer will tell you that the integer **5** is different from the one-character string **"5"**. A user has the ability to easily make an integer a string and vice versa:

Example 2.1. In this program, we print the string representation of 10.

Code 2.1

```
def main():
    print str(10)
main()
```

Output 2.1

```
10
```

Another useful data type in Python is a **list**. A list can be defined using square brackets and separating elements by commas, for instance:

Code 2.2

```
Colors=["Red", "Yellow", "Blue", "Orange", "Green", "Purple"]
print Colors[0]
```

If we want to access a specific element of the list, we can do this using a method called hashing. To do this, we type the list name followed by square brackets and the index that we want to hash into. An important thing to remember is that Python lists start at 0. Therefore, in the preceding code, hashing into the first list item would yield the following:

Output 2.2

```
"Red"
```

3 If/Else Statements

A boolean expression evaluates to either True or False. The comparison symbol == is used to compare the values of two objects. For instance, 1 == 1 will return True, while the string comparison “Apple” == “Banana” will return False. A conditional statement is used if you want an action to happen only if specific criteria are met. For instance, if you only wanted to print a number if it is even, you might write a program like the following:

Example 3.1. In this program, we define a function, IsEven, which takes in an integer, and prints whether the integer is even or odd. Notice that we use a new operator, a%b, which returns $a \pmod{b}$. In our main function, we then print the parity of both 16 and 101.

Code 3.1

```
def IsEven(x):
        if x%2==0:
                print x, " is even"
        if x%2==1:
                print x, " is odd"
def main():
        IsEven(16)
        IsEven(101)
main()
```

Output 3.1

```
16 is even
101 is odd
```

We can use a **if-else** statement if we want a second action to happen when the if condition evaluates to False.

Example 3.2. In this program, we define a function, GreaterThanFive, which prints “Success” if the input is an integer greater than 5. If this condition is not met, then the “else” statement executes, meaning that the function prints “Failure”.

Code 3.2

```
def GreaterThanFive(x):
        if x>5:
                print x, " is greater than 5"
        else:
                print x, " is less than five"
def main():
        GreaterThanFive(10)
        GreaterThanFive(2)
        GreaterThanFive(5.1)
main()
```

Output 3.2

```
10 is greater than 5
2 is less than five
5.1 is greater than five
```

4 Loops

In all of the examples above, we only do an action one specific time before ending the function. For instance, in the GreaterThanFive function, we evaluate the expression $x > 5$, and then print a statement based on this expression. However, in many programs, we want to execute an action more than once. One way to do this is through a **for** statement. A for statement will execute an action across a list or a range. For example, we can write a simple program to print all of the elements in the list.

Example 4.1. In the below program, we define a list, Groceries, with three elements. In our for statement, we say for each *grocery* in the list groceries, print that item. Notice that the word "grocery" could have been replaced by any other placeholder, such as "for item in Groceries:", and the program would have executed the same so long as we changed our print statement to be consistent. It is generally a good idea to use placeholder names that are descriptive, in the case that we have a long for loop.

Code 4.1

```
def main():
        Groceries=["Milk", "Eggs", "Meat"]
        for grocery in Groceries:
                print grocery
main()
```

Output 4.1

```
Milk
Eggs
Meat
```

The **range** function can be used to get a list of numbers. When a single argument, x, is passed into the range function, the list of all integers from 0 to $x - 1$, inclusive, is returned:

$$\text{range}(4) = [0, 1, 2, 3]$$

The reason that the range list begins at 0 is because when hashing into a list, it is important to remember that the first element of a Python list has an index of 0. Similarly, if two arguments, x and y, are passed into the range function, the list of all integers from x to $y - 1$, inclusive, is returned:

$$\text{range}(1, 5) = [1, 2, 3, 4]$$

One of the best uses of the range function is when used with a for statement, such as in the example below:

Example 4.2. In this function, we print the integers from 0 to 9, inclusive, with a for loop.

Code 4.2

```
for i in range(10):
    print i
```

Output 4.2

```
0123456789
```

The **length** function, len(), can be used to determine the length of a string or list.

Example 4.3. For instance, len("abc")=3, because there are 3 characters in the string "abc". In the example below, the output of the length of a string with 4 elements is 4.

Code 4.3

```
print len(["Baseball", "Football", "Basketball", "Hockey"])
```

Output 4.3

```
4
```

because there are 4 elements in the list. Using the range and len function, we can write a function similar to our print groceries one above:

Example 4.4. In this program, we define a list of classes, and then print them based on their index in the list. Notice that since the first index is 0, and our first class will be Class 1, rather than Class 0, we add 1 to our print statement for index.

Code 4.4

```
def main():
        classes=["English", "Spanish", "Math", "History"]
        for index in range(len(classes)):
                print "Class", index+1, ":", classes[index]
main()
```

Output 4.4

```
Class 1 : English
Class 2 : Spanish
Class 3 : Math
Class 4 : History
```

Another very powerful loop in Python is the **while** statement. The while statement performs an action while a condition is true, which gives the programmer more control than a program which simply iterates over a list using a **for** statement. For instance, if we wanted to write a program for the height of the ball that is dropped, but only want to consider it when the ball is above ground level, we could use a while statement. Another reason that Python is a great beginning first language is due to how similar it is to the English language. In the two examples below, we show the pseudo-code, and then the actual code, which read very similarly.

Example 4.5 (Pseudo-Code)**.** The while loop tests the statement "ball_height>0", and if it evaluates to true, it then performs an action. The height of the ball is then decreased, and the statement is tested again, until it returns false, at which point it exits out of the program.

Code 4.5

```
while ball_height>0:
    perform action
    decrease ball_height
```

Example 4.6. If we want to decrease a variable by a value, we can use either of the following expressions:

Code 4.6

```
ball_height=ball_height-4.9
ball_height -= 4.9
```

Example 4.7. We can combine these two ideas into one program, which will have the ball start at an initial height, then print the height of the ball and decrement it while it is still above the ground:

Code 4.7

```
def main():
    ball_height=16
    while ball_height>0:
        print "Ball is at height", ball_height
        ball_height-=4.9
main()
```

Output 4.7

```
Ball is at height 16
Ball is at height 11.1
Ball is at height 6.2
Ball is at height 1.3
```

5 Complex Applications of Python

Putting these basic concepts together can yield fairly complex programs. One of the first applications of the while statement is to write a simple program for base conversion. In order to convert a number n into base b, we can divide n by b repeatedly, until we have a quotient of 0:

$$\begin{aligned} n &= bq_1 + r_1 \\ q_1 &= bq_2 + r_2 \\ q_2 &= bq_3 + r_3 \\ \cdots & \\ q_{n-2} &= bq_{n-1} + r_{n-1} \\ q_{n-1} &= b \times \boxed{0} + r_n \end{aligned}$$

The **base b representation of n** is then $n = (r_n r_{n-1} r_{n-2} \cdots r_1)_b$. For instance, if we want to convert 83 into base 7, we can do the following:

$$\begin{aligned} 83 &= 7 \times 11 + 6 \\ 11 &= 7 \times 1 + 4 \\ 1 &= 7 \times \boxed{0} + 1 \end{aligned}$$

Once the quotient becomes 0, we immediately stop, and find the base representation by appending all the remainders together backwards, therefore, $83 = 146_7$.

Example 5.1. To write this in a program, the starting point will be to use a while loop, because we repeatedly want to divide n by b until this quotient becomes 0. Therefore, we start the program by making a function that takes two inputs (n,b) and uses a while loop to check if n is greater than 0.

Code 5.1

```
while n>0:
        n=n/b
```

Example 5.2. The second step is finding the remainder. In the example above, we notice that $6 \equiv 83 \pmod 7$, and $4 \equiv 11 \pmod 7$, therefore, we can add the following to our existing code:

Code 5.2

```
n=83
b=7
while(n>0):
        rem=n%b
        print rem
        n=n/b
```

Output 5.2

```
6
4
1
```

Example 5.3. This is approaching quickly to the desired answer that we want (146_7). Outputting the base representation in the form of a string is easiest, therefore, we create a new string, base_rep. We also make each remainder a string, and add it to the beginning of base_rep, rather than at the end:

Code 5.3

```
def base(n,b):
        base_rep=""
        while(n!=0):
                rem=str(n%b)
                base_rep=rem+base_rep
                n=n/b
        return base_rep
print base(83,7)
```

Output 5.3

```
146
```

One immediate applications of using a for statement is to be able to print all the prime numbers from 1 to some range. In order to do this, we first off have to understand how to know whether a number is prime or not. The simplest primality test is through trial division, meaning that if we want to determine if a number, n, is prime, we test to see if any integer from 2 to $\sqrt{n}$ divide into n. If so, the number is composite. Otherwise, the number is prime.

Example 5.4. In the IsPrime function, if $x \leq 1$, then by definition of prime numbers, x can not be prime, therefore we return False. Once we return a value, the program ends immediately. Next, if x is 2 or 3, the function returns True, since they are both prime numbers. Otherwise, the function uses the method described above to determine the primality of x. Notice that we add 1 to the range because if $\sqrt{x}$ is an integer, then the range function will only include values from 2 to $\sqrt{x} - 1$.

Code 5.4

```
def IsPrime(x):
        if x<=1:
                return False
        elif x<=3:
                return True
        else:
                for i in range(2, int(x**0.5)+1):
                        if x%i==0:
                                return False
        return True
def PrintPrimes(limit):
        for i in range(limit):
                if IsPrime(i)==True:
                        print i,
def main():
        PrintPrimes(100)
main()
```

Output 5.4

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

6 Conclusion

Though it's simple enough to be a first programming language, Python doesn't need to be classified as just for beginners. Part of its convenience lies in the fact that it lacks the often confusing syntax found in other languages, but in many cases, trading functionality for simplicity is barely noticeable. Python is powerful and versatile enough for uses in mathematics, robotics, cryptography, computer vision, and much more. And its accessibility makes the list of applications ever-expanding.

There are many resources available to get started with Python. The official Python site explains how to get started, with links to many useful lessons:
https://www.python.org/about/gettingstarted

Google has a class on Python with video lectures:
https://developers.google.com/edu/python

LearnPython features a place to run code in the browser while reading tutorials:
http://www.learnpython.org

CodeAcademy's Python track reads and interprets code, giving feedback on mistakes and errors: https://www.codecademy.com/learn/python

For the more advanced coder, Python Challenge tests knowledge of Python concepts in the context of solving a puzzle: http://www.pythonchallenge.com