A Little Linguistic Introduction, Some Coding Guidelines, and Fashionable Input and Output in Python

With some extras...

Author:
Jon Fast
Course Assistant
jonathan.fast@msu.montana.edu

Class:
CSCI107
Spring 2014

1. Introduction

1.1. Some Ideas to Think About. So you've probably been at this coding thing for a week now, and feel pretty confident about doing simple things in Python. You probably already know how to give messages to the user, the simplest example being the infamous *"Hello World"* program...

```
# a simple (correct) hello world program
print("Hello World!")
```

Sure, it's a simple program, but it can tell us a lot about the steps we must take to get a message onto the screen, and even more about the language we're using. All good languages have these important components:

First: Morphemes
These are simple 'units' or 'building blocks' of the language, *words or symbols.*
Second: Syntax
This is a structure, determining how those 'building blocks' are allowed to fit together. If you're in English class (or perhaps learning a new language), you might have seen numerous books on *grammar* that you've begun to love or hate. A grammar is a way to structure a sentence so that it makes sense. *Note: Grammar is only one piece of the syntax puzzle, but it's enough for us!*
Third: Semantics
How does the language create meaning? Does a sentence convey what we want it to? *Does what you say make sense?*

You'll soon find that programming in Python requires knowledge of all of these things. For example, *print()* is a morpheme and a string like "Hello World" is also a morpheme. A *print()* statement is ordered in such a way that it is clear (with a grammar!). It starts with the word 'print', then a left parenthesis, a string, strings, or other arguments inside, and finally a right parenthesis. Anything else would sound wrong to Python. For instance, we wouldn't want to write:

```
# a bad hello world program (which would not run if we tried)
("Hello World!")print
```

by putting 'print' on the right hand side, or something as silly as ...

```
# an awful hello world program (don't even think about running this either!)
""Hello print World)(!
""" If you're so inclined, garbly mess can come after this... """
```

and expect Python to to know what we're trying to say! If somebody told you that *'cat The roof jumped off'* instead of *'The cat jumped off the roof'*, would we be more inclined first to:

(1) Say we didn't understand?
(2) Ponder the meaning of a very strange sentence? *OR ...*
(3) Help the cat?

I wouldn't suggest choosing #3 is impossible for us. Humans have the amazing ability to fudge an understanding of a poorly prepared sentence by picking out words like *cat*, *jumped*, and *roof*...enough to know the 'big picture' about what is going on. If you chose #2, analytically for a human it would not be a very hard task to reorder the words, or sit on a rock for 12 hours with a quizzical look. If philosophy is your thing, or perhaps being a poet with a broad sense of creative license, go right ahead.

Where does option #1 fit into the scheme of things then? Unfortunately, computers are very precise and have no way of figuring out what you've said if you've said it incorrectly. They can only throw an error if they know something is wrong. Understandably, a computer will continue to pick #1 over and over again if you continue to throw it the malformed sentence: *'cat The roof jumped off'*. To say the least, you would never get anything done.

1.2. Why At All Are We... Going to *English class* again? I know some of us would like to put these repressed memories of grammar torture behind us (and some of us wouldn't, who am I to judge?).

The reality is that understanding language essentials prevent us from making mistakes! Otherwise, the computer will find it hard, if not *IMPOSSIBLE* to understand us. Here's why:

"Language is the means of getting an idea from my brain into yours without surgery." - *Mark Amidon*

The being said, if you're not interested in making your own Python zombie clone, you follow the simple, but strict grammar rules that were given to you and use the morphemes that were provided. They will make your life so much easier.

The final part of all this 'language stuff' is the semantics. *Does your program do what you want?* For instance, if your program is designed to find the sum of the numbers 0 through 10 inclusive, does your program give you the value 55? Or does it tell you that it would "rather go to the movies with its friends" as a string? Or does it give you 49? Or does it do something else completely unexpected? Semantic or 'logic errors' are often hard to detect, but through careful programming can be avoided! You will avoid these kinds of errors by following these simple rules:

(1) Do you really know what the function or language feature you are using does? Make sure you 100% do.
(2) Have you looked at the documentation first? This will eliminate confusion with [1].
(3) Check your program over for simple typos. For instance, did you mean '/' for floating point division or '//' for integer division?

1.3. How Can I Make You Understand? Also, you should make your code understandable by others, preferably others in your discipline. Classmates, and teachers alike fall into this category. If we don't know what you are doing, how will anybody know?

1.4. What Should I Do? (Guidelines). All this being said, take some time for reflection when writing your program. Identify its strengths and weaknesses. Really connect with YOUR work for a moment and ask yourself all of the questions below...

I should ask myself:

(1) What is the purpose of my program? What does it accomplish?
(2) Is my program well written?
  (a) Is my program readable?
  (b) Is my program understandable?
  (c) Can I give my program to a friend and have them understand what it does?
  (d) Did I write my program with TLC (that is, tender loving care) or did I 'throw it together'?
  (e) Could I thin parts of my program down to make it more understandable? Could I remove extraneous 'spaghetti' code (YUCK!) from my program?
  (f) Do parts of my program that I think are unclear require expansion (or expounding upon)?
  (g) Have I commented enough? Do my comments explicitly inform the reader of what I am doing?
(3) Does my program follow (to the LETTER) the rules of the Python 3 language?
  (a) Does it run on in the interactive textbook?
  (b) Does it run in IDLE (discussed later on in the course)?
  (c) Does the program produce any error messages while running?

Hopefully these questions will be able to help you make sense of how to write your programs. Now on to the fun stuff!

2. Simple Output

2.1. Hit F5 to Refresh. Just knowing about the *print()* function isn't enough to say we know everything, so let's delve into it a bit and see if we can't understand more about it.

2.2. Print Needs '(' and ')' ! Without left and right parentheses, we wouldn't be able to give a string to Python and expect it to do anything.

> ***Note:*** *The interactive textbook has the propensity to allow the Python 2 'print statement' to run.* We are using Python 3 in this class so don't be fooled for a second into believing that a bare print statement like:
>
> ```
> print "Hello World!"
> ```
>
> will work all of the time. The environment installed on the lab computers does not support this functionality, so chances are, if you tried to click on the **.py* file to run it, you'd get an error!

Moral of the story: always put parentheses around your print function, and put the string (i.e. "Hello World") inside of the parenthesis to print out what you want.

2.3. On Commas and Plusses (Concatenation). When using the *print()* function, commas are useful for separating things you want to print out with a space. For instance, this small program will print out "I like cookies!":

```
# print the phrase "I like cookies!"
print("I", "like", "cookies!")
```

Spaces will be placed at the position of each comma. There's an alternate way of doing this too, using the '+' or concatenation operator like so:

```
# print the phrase "I like cookies!" using concatenation
print("I " + "like " + "cookies!")
```

Notice that I had to add spaces inside of each string to make Python print the same thing as before. Concatenation allows you to more precisely tune what you're printing than the comma does, unless you're printing simple sentences (then use the comma). One downside of the comma is that the strings get separated, so if you'd like to perform one operation on multiple strings inside of the *print()* function, you're better off using concatenation. Otherwise, if it looks like you're painfully adding a space to each string to make it look right, use the comma operator.

2.4. On Justification. Remember that if you want output to print to the left, right, or center of the screen, you will need to justify it. Justification adds 'just' the right amount of spaces to a string you provided to center it inside of a field of a given length. This may not make sense at the moment, but consider an analogy to packing boxes when moving.

(1) If you have a delicate vase and you want to place it into a large box, you will need to surround it with packing peanuts. You should only use enough so that the vase doesn't move around during shipping, which could lead to damage.
(2) Also, what if the box is too small? Then you'll need to find a bigger box right?

Consider a string (or in this case, a word) with 7 characters in it (i.e. "cookies"). We have a box that we want to stick "cookies" in that has size 21. We want to push the word "cookies" to the center of the box. We might then use the snippet:

```
# print the 7 character word cookies in
# the center of a 21 character field (box)
print("cookies".center(21))
```

What is actually happening internally? Well, Python knows we have 21 spaces (or a box of size 21) to work with. It is going to take the size of the box (or our field width) and subtract the number of character in the string "cookies" from it:

$$21 - 7 = 14$$

...then Python will divided that number by two:

$$\frac{14}{2} = 7$$

and add 7 spaces (or packing peanuts) to the beginning of the string and 7 spaces to the end of it. Granted, if we have an odd number of spaces the word might not be exactly centered. Likewise, with left justification (or any other type of justification for that matter):

```
# print the 7 character word cookies to
# the left of a 20 character field (box)
print("cookies".ljust(21)) # or... print("cookies".rjust(21))
```

. . . 14 spaces will be appended to the **right** side of the string (pushing it to the far left in the field). If we replaced *ljust(21)* with *rjust(21)*, 14 spaces would be appended to the left side of the string (which will push "cookies" all the way to the right of the field). If we make the field bigger (say change the number 21 to 42, for instance), the string "cookies" would be pushed further right when we *rjust(42)* or *center(42)* it.

> **Note: The box is only as large as you make it.** If you try to center "cookies" in a field width of 6, **nothing** will happen! Also, note that if you have a string with spaces like "I like cookies" that you can center the entire string inside of a field width that is greater than its size (for the current example, it's 14). You can also center a string that is concatenated inside of a field:
>
> ```
> print(("something " + "out there").center(50))
> ```
>
> . . . will center an entire string "something out there" inside of a field width (or box, if you will) of 50, and will keep the spaces in between the words intact.

In all, justification is useful for making your output neat. For instance, if you want to display a table with the proper column widths across in Python using, using these functions would be a way to make it look neat!

3. Simple Input

3.1. Same as Before. *input()* is also a function. Inside of the parenthesis you place a string value that will be the prompt. For instance, one prompt might be *'What is your first name?: '* or *'What is your age?: '*. Beside the prompt you enter values corresponding to the answer to any one of those questions. 'Jon' might be the answer to the first question and '22' the answer to the second question. Other values could be used as well . . .

3.2. Regarding Types. The *input()* function will always return a string if you assign a variable name to it. For instance, if:

```
# assign a variable 'a' to some input
a = input('Something: ')
# get the type of a
print(type(a))
```

The printed result would be 'string'. The only two other types of values you may want to deal with are integers and floating point numbers. To retrieve these values from an input you will have to 'convert' them to the appropriate types by doing something like this...

```
# assign a variable 'a' to some input, and cast it to a float
a = float(input('Some float: ')) # or, a = int(input('Some int: '))
# get the type of a
print(type(a))
```

The type of 'a' in this case will be a float. You can also wrap the *int()* function around the *input()* function to get the same result, with the exception that the type of 'a' will now be an integer, and not a float or a string. After you 'cast' or 'convert' the input to the appropriate type, you can do whatever you want with it, including math (provided its allowed for that type). For instance, adding strings symbolizes concatenation where adding numbers symbolizes addition, so expecting '3' + '5' to be equivalent to '8' would not work for a string, whereas $3+5$ would equal 8 if both 3 and 5 were integers or floats. However, note that if either 3 or 5 were floats (being 3.0 or 5.0) the type of the variable assigned to the addition operation (i.e. $3.0+5$ or $5.0+3$ would be 8.0) due to conversion that occurs automatically (some use the term 'implicit' conversion to describe this phenomena). Just remember that types are super important and will become more important in your later forays into Python.

3.3. Waiting On Input. To wait on input, simply use the *input()* function with an empty string inside, and assign the value of input to a dummy variable (which I have called 'wait'):

```
        # wait for input
        wait = input("")
```

This will prevent your program from exiting out of the console window after you've run it, so you can see the output in the window.

4. Conclusion

Well, I think that's it for today! I sure hope you got through all of this, and next time we'll cover some more cool Python programming things. If you want things covered that you've become interested in about Python, let me know by email. Otherwise, until next time then!