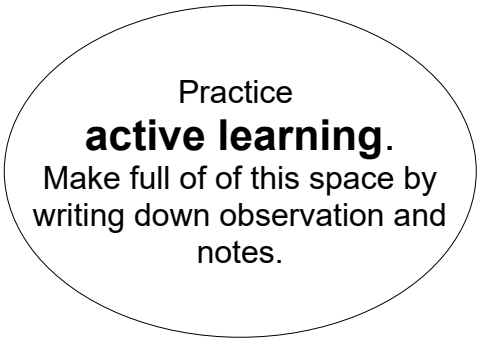
01. Print statements and strings



Objectives

- Print strings and characters using `print`
- Use the `\n`, `\t`, `\"`, `\'`, `\\` characters
- Use `std::endl` to force newline
- Write multiple print statements

In this set of notes, we learn to print strings and characters.

A quick advice to ease the pain of learning your first programming language: Type the program **exactly** as given. Even your spaces and blank lines must match the spaces and blank lines in my programs.

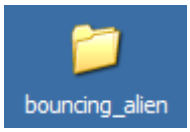
Let's begin ...

Introducing the bug

Before going ahead with the notes, go to our web site and look for the notes. You will see “Bouncing Alien”. Click on the link and save the file to your Desktop. The file is a zipped folder. You will need to unzip it.

We've caught an alien bug and kept it in a box. It's desperately bouncing around trying to escape. To see it follow these instructions ...

You're given a folder called bouncing_alien:



Go into the folder by double-clicking on it. Inside the folder you will see the following four files:

This is a Python program:

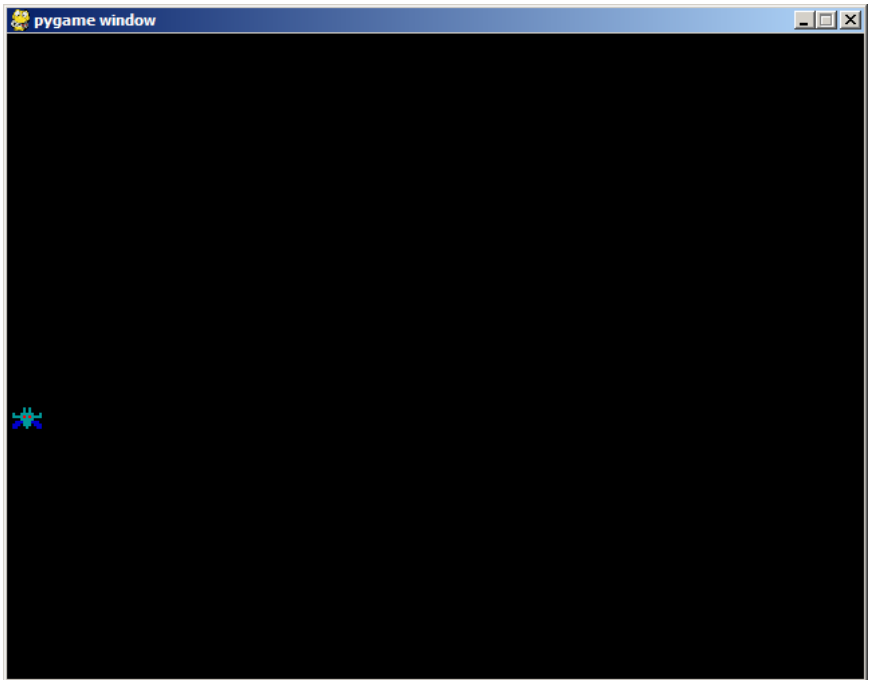
A program is a bunch of instructions to the computer kept in a file. The actual name of the file of this program is bouncing_alien.py To run this program, double-click on the icon. You will see an alien bug in the box:

ASIDE: If you installed an older version of Python, you will see this icon for a Python program:

The file ChatTag.wav

ChatTag

is a sound file. The remaining two files GalaxianAquaAlien.gif and GalaxianFlagship.gif are images files. Try double-clicking on all of them.



Phew ... an alien that size can't possible do much harm to humans ...

OK. Forget about the silly story line.

The above program bouncing_alien.py is a (relatively simple) Python program with graphics, animation and sound. By the end of this set of notes, you will learn how to read the program that produced the graphics, animation and sound. You will learn some “parameters” that control the program and how to modify them. These are called **integer variables**. Later you will learn how to draw the bug, how to make it move, how to make it bounce off a wall, how to play a sound, etc. So we have a lot of stuff to cover. Let's get started ...

Hello world

... commercial break ... go to notes on software tool(s) for writing and running a program …

Go ahead and save the following to `main.py`:

```
print("Hello, world!\n")
```

and then in your bash shell, you execute `python main.py`:

```
[student@localhost ~]$ python main.py
Hello, world
```

Or you can execute the python statement in the python shell.

Next, try this program:

```
print("Hello ... world! ... mom\n")
```

Exercise. Write a program that prints the following:

```
hello columbia
```

Exercise. Write a program that gets the computer to greet you:

```
hello dr. liow, my name is python.
```

(replace my name with yours ... you're probably not “dr. liow”).

Exercise. Debug (i.e., correct) this program by hand and then verify by running it with your C++ compiler.

```
print("Hello, world!\n"
```

Practice doing a hello world program until you can do it in < 2 minutes and without your notes.

Statement

Here's the first jargon. Look at our program again:

```
print("Hello, world!\n")
```

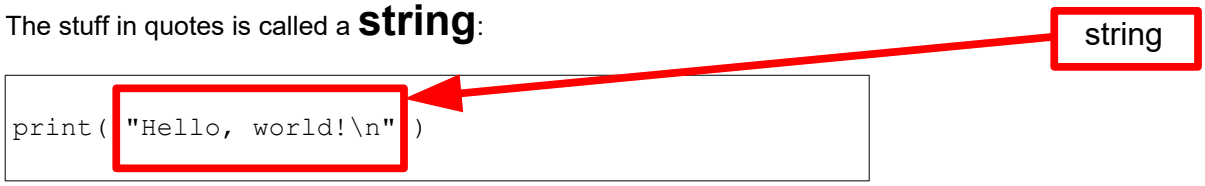
The line in bold is a **statement**.

At this point you should think of a statement as something that will cause your computer to perform some operation(s) when you run the program.

If you like, you can think of a statement as a sentence.

Strings

The stuff in quotes is called a **string**:



You can think of a string as textual data. (Soon I'll talk about numeric data for numeric computations. Got to have that for computer games, right?)

Double quotes are used to mark the beginning and ending of the string. So technically they are ***not*** part of the string. That's why when you run the above program, you do not see double-quotes. I will say that the double quotes are **delimiters** for the string since they are used to mark the beginning and the end.

In Python you can also use single quotes, triple double quotes and triple single quotes as delimiters of string. Run this:

```
print('Hello, world!\n')
```

And then this:

```
print("""Hello, world!\n""")
```

and this:

```
print('''Hello, world!\n''')
```

In the string `"Hello, world!\n"`, `H` is a **character**. The next character is `e`. Etc. Note that the space is also a character. You can think of the character as the smallest unit of data in a string.

The string `"abc"` contains 3 characters.

Note that a string can contain as many characters as you like: 10, 20, 50, 100, etc. There's actually a limit, but we won't be playing around with a string with 1,000,000 characters anyway for now – that would be a pain to type!!! Note that a string can contain no characters at all: `""`. The string `""` is an **empty string** (also called a **null string**): it's a string with no characters.

In Python a character is just a string of length 1. Here's an example: `"H"`.

The number of characters is a string is called the **length** of the string. The string `"abc"` contains 3 characters and therefore has a length of 3. And an empty string has length 0.

(C/C++ note: Python does not have a character type. In Python a character is just a string of length 1.)

Printing more than one string

Try this

```
print("abc", "def", "ghi")
```

Here I'm printing three strings. Note that Python will print a space between two consecutive strings.

Also, try this:

```
print()
```

In other words, it's OK to print nothing. Of course from the above, even if you print nothing, you will still get a newline.

Exercise. Does this program

```
print("Hello", ",", "world!\n")
```

print

```
Hello, world!
```

Exercise. Write a program that prints

```
green eggs and ham
```

by printing characters.

Case sensitivity

Is Python case sensitive?

Exercise. Does this work? (i.e. `Print` instead of `print`):

```
Print("Hello, world!\n")
```

Fix it.

Now answer this question:

Is Python case sensitive? (Circle one) YES NO

(duh ... I'm not taking answers in class.)

Whitespaces

A whitespace is ... well … a white space.

Spaces, tabs, and newlines are whitespaces.

Exercise. Modify your program by inserting some whitespaces:

```
print  (    "abc",      "def",          "ghi)
```

Does it work? What about this version:

```
print("abc",
"def",
"ghi")
```

Or this:

```
print("abc"
,
"def"
,
"ghi")
```

Or this:

```
print(
"abc"
,
"def"
,
"ghi"




)
```

In general you can insert whitespaces between “basic words” understood by C++. These “basic words” are called **tokens**. (Oooooo another big word.)

If you think of statements as sentences then you can think of tokens as words.

For instance the following are some tokens from the above program:

```
print
"abc"
"def"
"ghi"
(
,
```

We say that Python **ignores some whitespaces.**

Exercise. Try this:

```
pr    int("abc", "def", "ghi")
```

Does it work?

This shows you that `print` is a token – you cannot break it down.

Exercise. How about this one:

```
print
("abc", "def", "ghi")
```

Try this:

```
print("Hello,          world!\n")
```

The whitespaces between tokens are removed when your computer runs your Python program. So, to the computer, it doesn't matter how many whitespaces you insert between tokens – it still works. However spaces in a ***string*** are ***not*** removed before the program runs. The space characters `' '` (there are 10 in the above string) are actually characters within the string.

Try this:

```
        print(1, 2, 3, 4)
```

The whitespace in front of your Python statement is significant.

Note that although you can insert whitespaces, in general, good spacing of a program makes it easier to read. Therefore you must follow the style of spacing shown in my notes. In particular, I **don't** want to see monstrous programs like this (although the program does work):

```
print    (   1,       2,    3, 4)
```

And don't try to be cute this like ...

```
print(
 1,
  2,
   3,
    4)
```

The above examples are excellent candidates for the F grade.

The left trailing spaces of a statement is called the **indentation** of the statement:

```
print("Hello, world!\n")
```

For our hello world program, indentation is not allowed. But later, you'll see that indentations are used a lot and they are very important. It's a common programming style to use 4 spaces for indentation.

Special characters

Hmmmm ... you don't see the `\n` printed out. In fact … what's it for???

Try this:

```
print("Hello, world!\n\n\n")
```

And this

```
print("Hel\nlo, \nworld!\n")
```

And finally this:

```
print("Hello, world!")
```

So you can think of the print cursor as jumping to the next line whenever a `\n` is encountered.

As a matter of fact you cannot separate the `\` from the `n`. `\n` is consider ***one*** single character. `'\n'` is called the **newline character**.

For instance the string

```
"Hello, world!\n"
```

has a length of 14 and not 15.

Exercise. Write a program that prints this:

```
^ ^
0 0
 |
---
```

Try this:

```
print("1\t2\tbuckle my shoe\n3\t4\t...\n")
```

The `'\t'` move the print cursor to the next tab position. `'\t'` is also a character. It's called the **tab character**. You should not use the tab character. There are better ways to use the tab character to create a tabulation of data.

Special characters like the above (the newline and the tab characters) are not printable and so we have to use some special notation to say “the newline character” or the “the tab character”. Computer scientists decided (long time ago) to use \ to indicate a special character. These are called **escape characters**.

Now recall that `"` is used to mark the beginning and end of a string. What

if you want to print something like this:

```
.....".....
```

You might try this:

```
print(".....".....")
```

But it doesn't work right? Do you see why?

Now try this:

```
print("He shouted, \"42!\"\n")
```

So in a string, `\"` will let you print `"`.

`'\"'` is actually a character. There are two `'\"'` characters in the above string `"He shouted, \"42!\"\n"`.

Exercise. Write a program that prints this:

```
She said, "I would rather marry a pig."
```

Note that

```
print("He shouted, \"42!\"\n")
```

can be rewritten this way:

```
print('He shouted, "42!"\n')
```

i.e., use ' as string delimiters so that you can put `"` inside the string.

So in general if a string contains double quotes, you might want to use single quotes as delimiters. And if a string contains single quotes, you might want to use double quotes as delimited.

Exercise. Write a program that prints this:

```
Here's a standard formula:

       2      2          2
(x + y)   =  x  + 2xy + y
```

There are two `' '` characters on each side of the `'='` character.

Exercise. Write a program that prints this:

```
 d  3     2
-- x  = 3x
dx
```

Exercise. Write a program that print the backslash character, i.e. \.

If a string contains both single and double quotes, you might want to use

triple double quotes or triple single quotes as delimiters. Run this:

```
print("""..."....'..."""")
```

Exercise. Write a program that prints this:

```
He said, "I am! I'm a pig! Really!"
```

Exercise. Write a program that prints this:

```
Get the Gold!
+--------------+
|              |
| --------+ | |
|         | | |
| +---+ | | | |
| |G  | | |   |
| +-+ +-+ | | |
| | |       | |
| | +-----+-+ |
|               <---
+--------------+
```

Exercise. Write a program that prints this:

```
"""'''
```

Exercise. Write a program that prints this:

```
print("Hello, world!\n")
```

Exercise.
What is the number of characters (i.e., the length) of the following strings?
(a) `"columbia"`
(b) `""`
(c) `"columbia,mo"`
(d) `"columbia, mo"`
(e) `"ma ma mia!"`
(f) `"ma ma mia!\n"`
(g) `"ma\tma\tmia!\n\n"`

Controlling the automatic newline

Note that the print does certain things for you automatically. For instance if you print a three strings:

```
print("abc", "def", "ghi")
```

Python will automatically print a space between two consecutive strings. Also, after printing the three strings, Python will automatically print a newline.

You can tell Python not to auto print a newline by doing this:

```
print("abc", "def", "ghi", end="")
```

Of course the default auto newline at the end is really the same as

```
print("abc", "def", "ghi", end="\n")
```

This is really important for future programs. As for controlling the auto space between two values, see sections below.

Multiple statements

Exercise. Run this program

```
print("1")
print("2")
```

This tells you that you can have **more than one statement** in your program. Furthermore Python executes from **top to bottom** (at least for now!)

Exercise. The following program has three print statements:

```
print("Hello, World! ", end="")
print("I'm the queen of England. ", end="")
print("I have 3 arms.", end="")
print()
```

Run it. Rewrite it so that it has only one print statement.

Exercise. True or False? The output of this program (when you run it of course)

```
print("this is the last line")
print("this is the second line")
print("this is the first line")
```

is

```
this is the first line
this is the second line
this is the last line
```

Exercise. Continuing with the previous program, rewrite it so that each sentence is printed on a separate line; use only one statement.

Multi-line strings

Suppose I want to write a program that prints this:

```
Hello, World!
I’m the queen of England.
I have 3 arms.
```

One way of doing that is

```
print("Hello, World!\nI’m the queen of England.\nI have 3 arms.")
```

Here’s another possible solution:

```
print("Hello, World!")
print("I’m the queen of England.")
print("I have 3 arms.")
```

This is more readable.

Triple quoted strings (whether with triple double quotes or triple single quotes) allow you to run this:

```
print("""Hello, World!
I’m the queen of England.
I have 3 arms.""")
```

You cannot create the same effect as the above using non-triple quoted strings:

```
print("Hello, World!
I’m the queen of England.
I have 3 arms.")
```

This will give you an error.

Exercise. Rewrite the program that prints the following using a multiline string:

```
Get the Gold!
+-------------+
|             |
| --------+ | |
|         | | |
| +---+ | | | |
| |G  | | |   |
| +-+ +-+ | | |
| | |       | |
| | +-----+-+ |
|               <---
+-------------+
```

Very long statements

When your statement is very long, you can break it up into two lines. In the case of

```
print("abc", "def", "ghi")
```

You can write it as

```
print("abc",
"def",
"ghi")
```

or (to make it more readable), write it like this:

```
print("abc",
      "def",
      "ghi")
```

Python will know how to join up the three lines to form one statement.

There are cases where if you break the statement at the wrong place, Python will get confused. For instance run this

```
x = 1 + 2 + 3
print(x)
```

(See next chapter on integers.) Perfectly fine. But if I break it up this way:

```
x = 1 +
    2 + 3
print(x)
```

you'll get an error. To fix this you add a line continuation symbol (a backslash) like this:

```
x = 1 + \
    2 + 3
print(x)
```

Test it and you'll see that it works. It's important to not put any character after the backslash (watch out for an accidentally space after the backslash!)

It's a common practice to not write programs with very long statements because long statements can make programs difficult to read. You'll see what I mean when you start writing huge programs.

Exercise. Put a space after the backslash and run the following:

```
x = 1 + \
    2 + 3
print(x)
```

String variables

In math, you have the concept of variables. For instance in math you might write

x = 42

You can do the above in Python. Run this:

```
print(42)
x = 42
print(x)
```

In this case x is an integer variable. (See next chapter on details about integers.)

In Python (and many other programming languages), I can create a **string variable**. Run this:

```
print("abc")
x = "abc"
print(x)
```

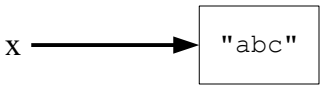
Get it?

In the Python statements

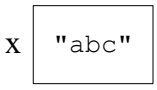
```
x = "abc"
```

the = is called an **assignment** operator. After this assignment statement, `x` is a string variable.

When you run the above program, Python remembers this in its brain:



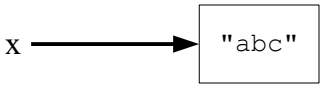
Sometimes to save on ink I might draw this instead (i.e. without the arrow):



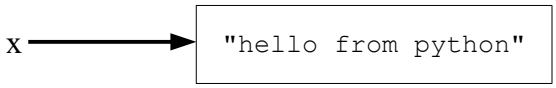
The important thing to note is that the value of `x` now is `"abc"`, You can put a different string value into the box. Run this:

```
x = "abc"
print(x)
x = "hello from python"
print(x)
```

At the third statement, your program changes the value in the box of x from



to this:



Note that `x`'s box can only hold one value. Python does not remember the old value of `x`. There's no such thing as "undo"!!!

String binary operation: concatenation

For integers, you know from your math classes that there are operations that involve pairs of integer (such as addition). These are called binary integer operations. There are also binary operations on strings. Here's one. Run this program:

```
print(1 + 2)
print("abc" + "def")
```

In Python you have + for integer values. Of course 1 + 2 gives you 3. In Python you also have + for strings. This + is not addition. It's called **concatenation**, i.e., "join up".. Here's a longer example. Run it:

```
print("Hello" + ", " + "Wor" + "ld!" + "\n")
```

Experiment with a few different strings on your own to get the hang of this idea

Of course it's obvious that you can do string concatenation involving string variables. Run this:

```
x = "Hello"
y = "Wor"
z = "\n"
print(x + ", " + y + "ld!" + z)
```

Also, try this:

```
x = "Hello"
y = "Wor"
z = "\n"
s =  x + ", " + y + "ld!" + z
print(s)
```

Exercise. True or false. The output of this program

```
x = "4"
y = "2"
print("the answer to life,",
      "universe and everything is",
      x + y)
```

is

```
the answer to life, universe and everything is 6
```

String input

Instead of giving a string variable a string value in your code like

```
x = "Hello"
```

you can ask the user to enter a string for x like this:

```
x = input("gimme a string: ")
print("you said ... ", x)
```

Here's an execution of the above program (named `main.py`) in linux:

```
[student@localhost ~]$ python main.py
gimme a string: wassup
you said ... wassup
```

In the above I entered the string `wassup` (and press the enter key). As mentioned earlier, I will use underlined text to indicate user input to a program.

The statement

```
x = input("gimme a string: ")
```

basically does the following:

- Python prints the string `"gimme a string: "`
- Python waits for a **string** value from the keyboard
- Once you entered a string and press the enter key, the string value you typed is given to variable `x`.

`input` is an example of a **function**. Don't worry about the concept of a function for the time being. I'll come back to that soon.

Exercise. What if you execute this program instead. Does it work?

```
x = input("")
print("you said ... ", x)
```

Exercise. What if you execute this program instead. Does it work?

```
x = input()
print("you said ... ", x)
```

Exercise. Write a program (named `chatbot.py`) that works as follows:

```
[student@localhost ~]$ python chatbot.py
What is your firstname: John
What is your lastname: Doe
Hi John Doe. My name is Python. How are you?
```

Controlling output: string formatting

Frequently you want to produce output with better control over the area where some data is printed, including the “window size” (or field size) and whether you want to left-justify or right-justify the data. Run the following program:

```
print("%s, World!" % "Hello")
print("hi %s" % "John")
x = "John"
feprint("hi %s" % x)
```

Get it? Study the above code again very carefully.

Basically, Python will compute

```
"%s, World!\n" % "Hello"
```

as

```
"Hello, World!\n"
```

The above string computation is called **string formatting**.

The `"Hello"` goes into the `"%s"` of `"%s, World!"`. Get it? You should also run this:

```
s = "%s, World!" % "Hello"
print(s)
```

You can have as many %s as you like. Run this example:

```
print("%s, %s!" % ("Hello", "World"))
print("%s, %s%s" % ("Hello", "World", "!"))
print("%s, %s%s" % ("Hello", "World", "?"))
print("%s, %s%s" % ("Hello", "Columbia", "?"))
```

Exercise. Write a program (named `greet.py`) that works as follows. Use string formatting Here’s an execution:

```
[student@localhost ~]$ python greet.py
What is the greeting (example: Hi/Hello/Hey): Hi
What is your firstname: John
What is your lastname: Doe
Hi Doe, John. My name is Python. How are you?
```

Here’s another execution of the program:

```
[student@localhost ~]$ python greet.py
What is the greeting (example: Hi/Hello/Hey): Hello
What is your firstname: Tom
What is your lastname: Smith
Hello Smith, Tom. My name is Python. How are you?
```

How would you write this program if you are not to use string formatting?

String formatting allows you to specify the “window size” for the output. Try this:

```
print("%s, World!" % "Hello")
print("%1s, World!" % "Hello")
print("%5s, World!" % "Hello")
print("%10s, World!" % "Hello")
print("%15s, World!" % "Hello")
```

To see the output window for `"Hello"`, it’s helpful to write your program this way:

```
print("[%s], World!" % "Hello")
print("[%1s], World!" % "Hello")
print("[%5s], World!" % "Hello")
print("[%10s], World!" % "Hello")
print("[%15s], World!" % "Hello")
```

The output looks like this:

```
[Hello], World!
[Hello], World!
[Hello], World!
[     Hello], World!
[          Hello], World!
```

Note that the `"Hello"` is right-justified in the output window. You can also force the output to be left-justified. Run this:

```
print("[%-s], World!" % "Hello")
print("[%-1s], World!" % "Hello")
print("[%-5s], World!" % "Hello")
print("[%-10s], World!" % "Hello")
print("[%-15s], World!" % "Hello")
```

Creating output windows for data and controlling left- or right-justification is very helpful for producing columns in a table.

Exercise. Write a program that produces the following output. Do not use the space or the tab character.

```
    n  n^3
    -  ---
    0    0
    1    1
    2    8
    3   27
```

Summary

A **statement** is something that will cause your computer to perform some operation(s).

Python is **case-sensitive**.

Python **ignores whitespace** (between tokens) except that the indentation is significant.

Something that looks like `"Hello, World!\n"` (or `'Hello, World!\n'` or `"""Hello, World!\n"""` or `'''Hello, World!\n'''`) is called a **string**. A **character** is a string of length 1.

A string is either empty (the null string), i.e. `""`, or it is made up of characters. For instance the first character of the string `"Hello, World!\n"` is `'H'`.

There are special characters: `'\n'` causes the cursor to jump to a new line while `'\t'` moves to the next tab position, `'\"'` is the double-quote character, and `'\''` is the single-quote character. The backslash character is `'\\'`. These are called **escape characters**.

Here are some string operations:

- String assignment. Example: `x = "aaa"`
- String concatenation. Example: `"aaa"` + `"bbb"` is `"aaabbb"`
- String formatting. Examples:
  - `"a %s c" % "bbb"` is `"a bbb c"`
  - `"a %s c %s" % ("bbb", "ddd")` is `"a bbb c ddd"`
  - `"<%5s>" % "bbb"` is `"<  bbb>"`
  - `"<%-5s>" % "bbb"` is `"<bbb  >"`

Exercises

1. Write a Python program that produces the following output:

```
"Prediction is very difficult,
especially about the future."
    Niels Bohr
    Danish physicist (1885 - 1962)
```

Use one print statement.

2. True or false: The output of

```
print("To be ...",
      "or not to be ...",
      "That", "is the question.")
```

is

```
To be ...
or not to be ...
That is the question.
```

Verify with Python

3. Find all the error in this program (without using the C++ compiler):

```
print("Hello!/n",
      "I'm the king of Spain./n",
      "And I have 3 arms.")
```

The expected output is

```
Hello!
I'm the king of Spain.
And I have 3 arms.
```

When you're done, verify your correction with Python. If your program does not run, continue correcting the program until the program is error-free and runs correctly.

4. Will this program get an A from the instructor?

```
print("Hello,","world!","Hello,","Columbia!")
```

Why? What about this:

```
print("Hello,","world!",



      "Hello,","Columbia!",



)
```

5. What is the length of the following string:

```
"Ablee, \tAblee, \t... That's all folks!!!\n"
```

6. A string with length 0 is called an _____ string or a _____ string.

7. Write the shortest program that produces the following output:

```
          8     5     3
    +     3     8     2
----------------------
    1     2     3     5
----------------------
```