Download this page as a jupyter notebook at Lesson 4

ENGR 1330 Computational Thinking with Data Science

Last GitHub Commit Date: 29 August 2021

Lesson 4 User Interaction:

- The `input()` function
- triple quotes
- escape characters
- The `print()` function

In [36]:
```
# Script block to identify host, user, and kernel
import sys
! hostname; ! whoami; ! pwd;
print(sys.executable)
```

```
atomickitty
sensei
/home/sensei/engr-1330-webroot/1-Lessons/Lesson04
/opt/jupyterhub/bin/python3
```

In [37]:
```
%%html
<!-- Script Block to set tables to left alignment -->
<style>
  table {margin-left: 0 !important;}
</style>
```

Objectives

1. Develop awareness of interactive inputs
2. Implement interactive inputs to generalize solution tools
3. Develop awareness of output formatting to improve readability
4. Implement output formats

Input and Output

- *Useful* programs take input and generate output
  - Command-line interface
  - Graphical user interface (GUI)
  - Files
  - Network sources
  - Databases

User Interaction

Until this point we have explicitly specified input values for variables (and constants) in a script; now lets leverage intrinsic functions that lets us makes use of variables. We'll revisit earlier examples, but this time we'll make them interactive. Instead of just computing and sending output, we want read into variables values that may change from time to time. In order to do that, our script needs to be able to prompt us for information and display them on the screen.

This whole process is the essence of user interaction, and from the simple examples herein, one builds more complex scripts.

Command-line input

- Function `input(prompt)`

  - prints the *optional* prompt on the command line, the prompt can be a string variable, or just a string literal
  - waits for the user to type something
  - the text is sent to the program only after the user types enter/return
  - the entered data is always interpreted as text, *even if it's numeric*

About the `input()` function

Consider the script below

In [38]:
```
MyName=input('What is your name ?')
print(MyName)
```

```
Jimmy Johns
```

The `input` method sent the string 'What is your name ?' to the screen, and then waited for the user to reply. Upon reply, the input supplied was captured and then placed into the variable named `MyName`.

Then the next statement, used the `print()` method to print the contents of `MyName` back to the screen. From this simple structure we can create quite useful input and output.

As a matter of good practice, we should explicitly type the input variable, as shown below which takes the input stream and converts it into a string.

In [39]:
```
MyName=str(input('What is your name ?'))
print(MyName)
```

```
Taco Bell
```

Below we prompt for a second input, in this case the user's age, which will be put into an integer. As a sdie note, we are not error checking, so if an input stream that cannot be made into an integer is supplied we will get an exception warning or an error message.

In [40]:
```
MyAge=int(input('How old are you ? '))
print(MyAge)
```

```
66
```

More examples

In [41]:
```
# Try getting some input
value = input( "Please enter a value: ") # input with a prompt
# print(type(value)) # get data type -- note always a string
# input() # without the prompt
# print(hex(id(value))) # get memory location of value in hex - sometimes useful for squashing bugs
value #return contents
```

Out[41]:
```
'grow 3'
```

In [42]:
```
# building the prompt
string_variable = 'Enter a value'
value = input(string_variable)
value
```

Out[42]:
```
'grow 3'
```

Command-line output

- Function `print()`
  - prints the value(s) passed to it
  - automatically converts data values to strings
  - goes to the next line by default
  - separates values with space by default

- optional arguments can set different separator and end of line values
- Format output using string formatting functions

About the `print()` function

The `print()` function is used to display information to users. It accepts zero or more expressions as parameters, separated by commas.

Consider the statement below, how many parameters are in the parameter list?

In [43]:
```
print ("Hello World, my name is", MyName, "and I am", MyAge, "years old.")
```

```
Hello World, my name is Taco Bell and I am 66 years old.
```

There are five parameters;

1. "Hello World, my name is"
2. MyName
3. "and I am"
4. MyAge
5. "years old"

Three of the parameters are string literals and are enclosed in quote marks, two are variables that are rendered as strings.

Some more examples

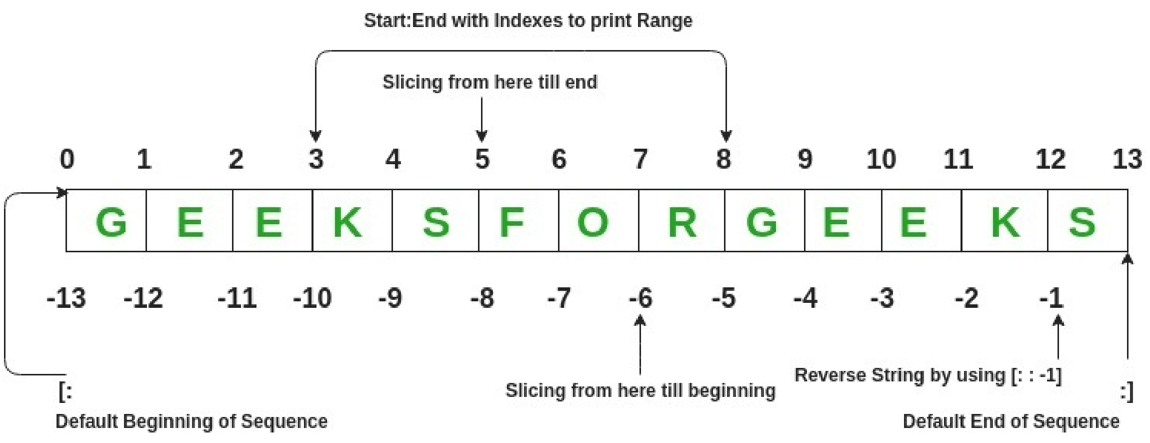
In [44]:
```
# Examples of output

print("Here is a message." )
print("Here is a value:", value )  # where did val come from?
print("Don't go to next line.", end = ' ' )
print("more text same line")
print("more text new line ")
```

```
Here is a message.
Here is a value: grow 3
Don't go to next line. more text same line
more text new line
```

About Strings

- A string is a complex data type
  - a *sequence* of characters that is *immutable*
  - individual characters are identified using indexing syntax: `s[position]`



Start:End with Indexes to print Range
Slicing from here till end
0 1 2 3 4 5 6 7 8 9 10 11 12 13
G E E K S F O R G E E K S
-13 -12 -11 -10 -9 -8 -7 -6 -5 -4 -3 -2 -1
[:
Default Beginning of Sequence
Slicing from here till beginning
Reverse String by using [: : -1]
:]
Default End of Sequence

- the general `len()` function returns the length of a string

In [45]:
```
# Experiment with indexing syntax
s = "birds"
print(s[3])
print(s[-1])
print(len(s))
type(s[3])
```

```
u
s
5
```

Out[45]:
```
str
```

String Operators

- `+`
  - concatenate strings
- `==` `!=`
  - test the equality of strings
- `<` `<=` `>` `>=`
  - compare strings alphabetically

In [46]:
```
# String operators
s1 = "cat"
s2 = "dog"
s3 = "cat"
print(s1 + s2)
print(s1 == s3)
print(s2 < s1)
print("Dog" < "cat")
```

```
catdog
True
False
True
```

Special and unprintable characters

- Represented with *escape* sequences
  - preceded by backslash `\`
- Common special characters

Character	Escape Sequence
newline	`\n`
tab	`\t`
backslash	`\\`
quotes	`\'` `\"`

In [47]:
```
# Escape sequences
print("hello\n")
print("1\t2")
print("\xEA")
```

```
hello

1       2
ê
```

About Escape Sequences

Sometimes we may need to print some special “unprintable” characters such as a tab or a newline. In this case, you need to use the `\` (backslash) character to escape characters that otherwise have a different meaning. For instance to print a tab, we type the backslash character before the letter t, like this `\t` using our earlier example we have:

In [48]:
```
print ("Hello\t World, my name is {} and I am {} years old.".format(MyName,MyAge))
```

```
Hello    World, my name is Taco Bell and I am 66 years old.
```

Here are a few more examples:

In [49]:
```
#newline after World
print ("Hello World\n, my name is {} and I am {} years old.".format(MyName,MyAge))
```

```
Hello World
, my name is Taco Bell and I am 66 years old.
```

In [50]:
```
# backslash after World
print ("Hello World\\, my name is {} and I am {} years old.".format(MyName,MyAge))
```

```
Hello World\, my name is Taco Bell and I am 66 years old.
```

In [51]:
```
# embedded quotes in the string literal
print ("I am 5\'9\" tall")
```

```
I am 5'9" tall
```

> If you do not want characters preceded by the `\` character to be interpreted as special characters, you can use raw strings by adding an `r` before the first quote. For instance, if you do not want `\t` to be interpreted as a tab in the string literal "Hello\tWorld", you would type

In [52]:
```
print(r"Hello\tWorld")
```

```
Hello\tWorld
```

String slicing

- expanded indexing
- a slice of a string is a new string, composed of characters from the initial string

Syntax	Result
`s[start]`	a single character
`s[start:end]`	a substring
`s[start:end:step]`	a selection of characters

- the end position is not inclusive (up to but not including *end*)
- the step can be positive or negative
  - a negative step proceeds backwards through the string

Empty and default values in slicing

- the default step is `1`
- the default end is `start+1`
- an empty value for end means the end of the string (in the *step* direction)
- an empty value for start means the start of the string (in the *step* direction)

In [53]:
```
# What will this do?
# s[-1::-1]
```

String functions

Name	Behavior
s.split()	Split a string into pieces based on a delimiter
s.strip()	Remove leading/trailing whitespace or other characters
s.upper()	Convert a string to uppercase
s.lower()	Convert a string to lowercase
s.isnumeric()	Return True if a string is numeric
s.find()	Return the index of a substring
s.replace()	Replace one substring with another
Many, many, more ...	Look them up as needed

In [54]:
```
# some string functions
s = "hello"
print(s.upper())
print(s)
```

```
str = s.upper()
print(str)
str = "I am a string."
print(str.replace("am", "am not"))
"222".isnumeric()
```

```
HELLO
hello
HELLO
I am not a string.
```

Out[54]:
```
True
```

In [55]:
```
s = "Fox. Socks. Box. Knox. Knox in box. Fox in socks."
print(s[1])
print(s[-1])
print(s[:3])
print(s.replace("ocks", "ox"))
print(s[0] + s[5] + s[12])
```

```
o
.
Fox
Fox. Sox. Box. Knox. Knox in box. Fox in sox.
FSB
```

The `%` operator

Strings can be formatted using the `%` operator. This gives you greater control over how you want your string to be displayed and stored. The syntax for using the `%` operator is “string to be formatted” %(values or variables to be inserted into string, separated by commas)

An example using the string constructor (`%`) form using a placeholder in the print function call is:

In [56]:
```
print ("Hello World, my name is %s and I am %s years old." %(MyName,MyAge))
```

```
Hello World, my name is Taco Bell and I am 66 years old.
```

Notice the syntax above. The contents of the two variables are placed in the locations within the string indicated by the `%s` symbol, the tuple (MyName,MyAge) is parsed using this placeholder and converted into a string by the trailing `s` in the `%s` indicator.

See what happens if we change the second `%s` into `%f` and run the script:

In [57]:
```
print ("Hello World, my name is %s and I am %f years old." %(MyName,MyAge))
```

```
Hello World, my name is Taco Bell and I am 66.000000 years old.
```

The change to `%f` turns the rendered tuple value into a float. Using these structures gives us a lot of output flexibility.

> The `%f` formatter can also be used to place the decimal by preceeding the f with the decimal point and the number of digits after the decimal you want to render as in:

In [58]:
```
print ("Hello World, my name is %s and I am %.1f years old." %(MyName,MyAge))
```

```
Hello World, my name is Taco Bell and I am 66.0 years old.
```

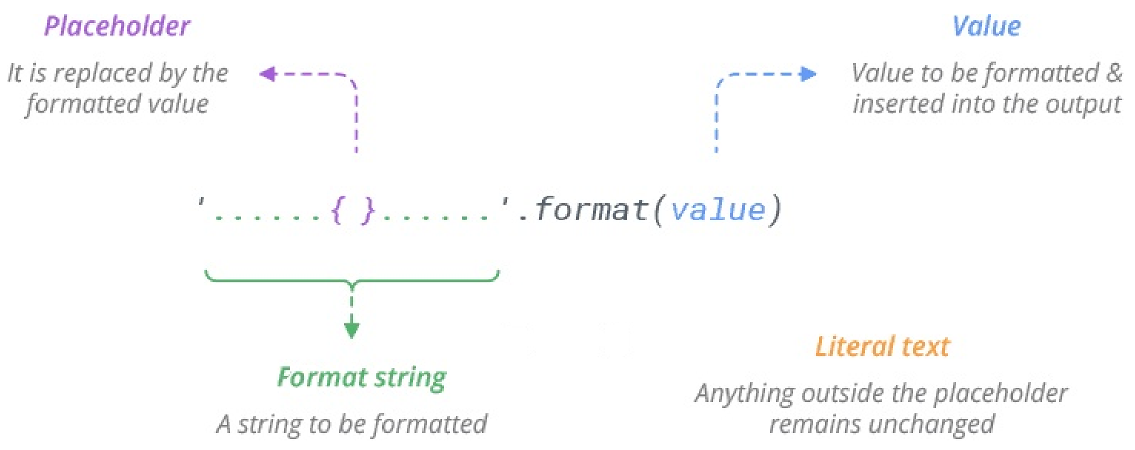
About the `format()` method

Similar to the `%` operator structure there is a `format()` method. Using the same example, the `%s` symbol is replaced by a pair of curly brackets `{}` playing the same placeholder role, and the `format` keyword precedes the tuple as

In [59]:
```
print ("Hello World, my name is {} and I am {} years old.".format(MyName,MyAge))
```

```
Hello World, my name is Taco Bell and I am 66 years old.
```

Observe the keyword `format` is joined to the string with a dot notation, because `format` is a formal method associated with all strings, and it is attached when the string literal is created.


Placeholder
It is replaced by the formatted value
Value
Value to be formatted & inserted into the output
'......{ }......'.format(value)
Format string
A string to be formatted
Literal text
Anything outside the placeholder remains unchanged

In this example the arguments to the method are the two variables, but other arguments and decorators are possible allowing for elaborate outputs.

Triple quotes

If you need to display a long message, you can use the triple-quote symbol (‘’’ or “””) to span the message over multiple lines. For instance:

In [60]:
```
print ('''Hello World, my name is {} and I am a resturant
that is over {} years old. We serve sodium chloride infused
lipids in a variety of shapes'''.format(MyName,MyAge))
```

```
Hello World, my name is Taco Bell and I am a resturant
that is over 66 years old. We serve sodium chloride infused
lipids in a variety of shapes
```

Future Versions

- repr and map methods
- color codes

Readings

1. Computational and Inferential Thinking Ani Adhikari and John DeNero, Computational and Inferential Thinking, The Foundations of Data Science, Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International (CC BY-NC-ND) Chapters 3-6 https://www.inferentialthinking.com/chapters/03/programming-in-python.html

2. LearnPython.org (Interactive Tutorial) (https://www.learnpython.org/) Short, interactive tutorial for those who just need a quick way to pick up Python syntax.

3. Brian Christian and Tom Griffiths (2016) ALGORITHMS TO LIVE BY: The Computer Science of Human Decisions Henry Holt and Co. (https://www.amazon.com/Algorithms-Live-Computer-Science-Decisions/dp/1627790365)

4. Learn Python in One Day and Learn It Well. Python for Beginners with Hands-on Project. (Learn Coding Fast with Hands-On Project Book -- Kindle Edition by LCF Publishing (Author), Jamie Chan https://www.amazon.com/Python-2nd-Beginners-Hands-Project-ebook/dp/B071Z2Q6TQ/ref=sr_1_3?dchild=1&keywords=learn+python+in+a+day&qid=1611108340&sr=8-3

5. Learn Python the Hard Way (Online Book) (https://learnpythonthehardway.org/book/) Recommended for beginners who want a complete course in programming with Python.

6. How to Learn Python for Data Science, The Self-Starter Way (https://elitedatascience.com/learn-python-for-data-science)

7. String Literals https://bic-berkeley.github.io/psych-214-fall-2016/string_literals.html

8. Tutorial on `input()` and `print()` functions https://www.programiz.com/python-programming/input-output-import

In [ ]:

Loading [MathJax]/jax/output/CommonHTML/fonts/TeX/fontdata.js