Python #12
Characters, Strings, Text

Location: https://people.sc.fsu.edu/~jburkardt/classes/python_2022/python12/python12.pdf

Freely adapted from the Python lessons at https://software-carpentry.org/

Text

- *String data is a standard Python data type;*
- *Strings are somewhat similar to an array of characters;*
- *The* `print()` *function has many useful options;*
- *A program can read and respond to interactive user input;*

1 Strings

In Python, any text delimited by single or double quotes is a string. By text, we usually mean a sequence of characters. And by character, we think of alphabetic letters, numbers, punctuation marks, blanks, symbols and some more obscure items. Here are some examples of characters:

```
'A'
'7'
'&'
'"'
"'"   a single quote has to be delimited by double quotes!
'\t'  a TAB
'\n'  a new line
'\r'  a carriage return
'\\'  a backslash
```

A string is a sequence of characters, and has the type `<class 'str'>`. A string is similar to (but **not the same as** an array or list of characters. Thus, if we define a string `s = 'A flock of seagulls'` then we can examine any substring we like by the usual indexing:

```
'A flock of seagulls'
 0123456789012345678
```

```
s[0:4] is 'A fl'
s[6:16:2] is 'ko_eg'
s[-1] is 's'
```

2 Character ↔ Integer Conversion

The ASCII character set contains 256 characters, indexed from 0 to 255. The most common characters are in the first half of the table, which is given below:

dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char	dec	hex	oct	char
0	0	000	NULL	32	20	040	space	64	40	100	@	96	60	140	`
1	1	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB	41	29	051	)	73	49	111	I	105	69	151	i
10	a	012	LF	42	2a	052	*	74	4a	112	J	106	6a	152	j
11	b	013	VT	43	2b	053	+	75	4b	113	K	107	6b	153	k
12	c	014	FF	44	2c	054	,	76	4c	114	L	108	6c	154	l
13	d	015	CR	45	2d	055	-	77	4d	115	M	109	6d	155	m
14	e	016	SO	46	2e	056	.	78	4e	116	N	110	6e	156	n
15	f	017	SI	47	2f	057	/	79	4f	117	O	111	6f	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1a	032	SUB	58	3a	072	:	90	5a	132	Z	122	7a	172	z
27	1b	033	ESC	59	3b	073	;	91	5b	133	[	123	7b	173	{
28	1c	034	FS	60	3c	074	<	92	5c	134	\	124	7c	174	|
29	1d	035	GS	61	3d	075	=	93	5d	135	]	125	7d	175	}
30	1e	036	RS	62	3e	076	>	94	5e	136	^	126	7e	176	~
31	1f	037	US	63	3f	077	?	95	5f	137	_	127	7f	177	DEL

www.alpharithms.com

Python offers two functions, `ord()` and `chr()`, which can convert between the ASCII index of a single character and the character itself. For example:

```
ord(' ') = 32      chr(32) = ' '
ord('0') = 48      chr(48) = '0'
ord('1') = 49      chr(49) = '1'
ord('A') = 65      chr(65) = 'A'
ord('Z') = 90      chr(90) = 'Z'
ord('a') = 97      chr(97) = 'a'
```

This is only a mapping between the ASCII indices and the ASCII characters. It will not convert the integer 1234 into the string '1234'. For that, we will need to learn about the `str()` function, coming shortly.

As an example of using the `ord()` function, we can print the index value for each character in a string:

```
for c in 'Monty Python':
  print ( '  ord(' + c + ') = ', ord(c) )
end{lstlisting}

\section{Concatenation}

We can make a new string using the {\bf{+}} operator to paste several strings together.
\begin{lstlisting}[numbers=none]
  rng = np.random.default_rng ( )
  for i in range ( 0, 6 ):
    day = rng.choice ( [ 'Monday', 'Wednesday', 'Friday' ] )
    time = rng.choice ( [ '10:00am', 'noon', '2:00pm', '4:00pm' ] )
    appointment = time + ' ' + day
    print ( appointment )
```

3 Number to string

We often need to create a string that combines character and numeric data. This might be for a plot title, a filename, a date or time. The `str()` function can do this conversion, and, in the case of real numbers, tries to make an intelligent decision about how many digits to use.

Suppose we wished to stored data in a sequence of files, with titles like 'journal_1900.txt', 'journal_1901.txt' and so on. We can generate the years with a `for` loop, and then combine them with the character data, as follows:

```
  for date in range ( 1900, 1906 ):
    s = 'journal_' + str ( date ) + '.txt'
    print ( '  "' + s + '" was published in ', date )
```

If, instead, we are generating plots of some iteration involving the real parameter `theta`, we may want to generate a corresponding name for each plot. Again, we can use `str()`:

```
  for theta in [ 0.0, 0.25, 0.50, 0.75, 1.00 ]:
    s = 'theta_' + str ( theta ) + '.jpg'
    print ( '  "' + s + '" is an image of data for theta = ', theta )
```

Recall that in one of our earlier exercises, we had to deal with a set of files ranging from *inflammation-01.csv* to *inflammation-12.csv*. Clearly, simply using `str()` will give us some wrong names. In particular, we will generate the name *inflammation-1.csv*. And if, instead, we specify a leading '0' in the string, we will instead generate some wrong file names like *inflammation-012.csv*.

What we want, in this case, is to tell `str()` to convert an integer to a string, but to make sure that that string is 2 characters in width. If it would be less than that, then an initial '0' should be inserted to fill it out. The somewhat cumbersome function `str().zfill()` will do that, if we specify the number to convert as the first argument, and the width of the output string as the second argument:

```
  for i in range ( 5, 15 ):
    filename = 'inflammation-' + str ( i ).zfill ( 2 ) + '.csv'
    print ( '  file #', i, 'is "' + filename + '"' )
```

If we had used a second argument of 3, then our first filename would be *inflammation-001.csv* and our last would be *inflammation-012.csv*. The important reason for doing this is that then an alphabetical list of the files will correspond to their numerical order.

4 Upper case, lower case, title case

The alphabetic characters come in two flavors, uppercase and lowercase. Sometimes it is desirable to force an alphabetic character or string to have a specific case. This can be done with the functions `lower()` and `upper`. For example,

```
s1 = '   My name is Elmer!   '
s2 = s1.lower()
s3 = s1.upper ( )
s4 = s1.title ( )
```

Here, the `title()` method capitalizes each word.

5 Removing or replacing blank characters

We might want to remove all initial and trailing blanks from a string. In that case, we use the `.strip()` method. If we only want to remove blanks on the left (initial blanks) or right (final blanks) we use `.lstrip()` or `.rstrip()` respectively.

```
s1 = '   My name is Elmer!   '
s5 = s1.strip ( )
s6 = s1.rstrip ( )
s7 = s1.lstrip ( )
```

We might want to remove **all** blanks from a string. In that case, the `.replace()` method can be called, listing the string to be removed, and its replacement. In this case, we want to replace space, ' ', with an empty character ''. The `.replace()` method is actually much more general, and can replace any given character by another.

```
s1 = '   My name is Elmer!   '
s8 = s1.replace ( ' ', '' )
s8 = s1.replace ( ' ', '_' )
s10 = s1.replace ( 'e', 'Z' )
```

6 How strings are not like arrays

In most ways, a string can be regarded as an array or list. In particular, we can use indexing to identify a particular character or a consecutive sequence of characters in the string.
One of the common operations in a string is to directly modify an entry. For instance, we might have an array like

```
prime = np.array ( [ 2, 3, 5, 7, 11, 15, 17 ] )
```

and realize that we made a mistake. To correct it, we write

```
prime[5] = 13
```

Similarly, we might have written

```
city = 'Pittzburgh'
```

in which case we'd be tempted to try to correct this by

```
city[4] = 's'  # Warning!  This won't work!
```

Such a statement will result in a warning from Python that strings are *immutable*. For our purposes, that just means that to change a string, we have to create a new one

```
city2 = city[0:4] + 's' + city[5:]
```

or simply redefine it correctly

```
city = 'Pittsburgh'
```

To add the state, however, we can simply concatenate:

```
city = city + ', Pennsylvania'
```

which is legal because it does not use indexing to try to change the string.

7 Comparison

Alphabetical order means that we can say that one alphabetic character is less than, equal to, or greater than another. (In fact, this is true for any characters, not just alphabetic ones.) This ordering depends on the ASCII table, so if the ASCII index of character c_1 is lower than that of character c_2, then it is true that $c_1 < c_2$. This means, in particular, that

```
'A' < 'B'
'A' < 'a'
'B' < 'a'
'0' < 'A'
'&' < '5'
```

Our primary interest in this fact is that, if we are given a set of characters or strings, we can use this alphabetic ordering to sort them. Given a bunch of file names, we can place them in alphabetic order, by comparing individual characters one by one.

From what we have said, we only know how to compare characters c_1 and c_2. Suppose we are given, instead, two strings, possibly of different lengths. How do we determine if $s_1 < s_2$, or $s_1 == s_2$, or $s_1 > s_2$? You should be able to write a short function that takes as input two strings and returns the result of this comparison!

8 Printing a value with a label

The Python command `print()` can display information about almost any Python object. We are used to printing variables, vectors, arrays, and strings, and seeing their values:

```
x = np.array ( [ [ 1, 2 ]; [ 3, 4 ] ] )
print ( x )
```

We can also print the value of an expression. Here we create a vector of values:

```
print ( [ x*5 for x in range ( 0, 4 )] )
```

or information about a built-in function

```
print ( print )
```

However, we need to know a little more about the `print()` command in order to create neat, informative reports. A common task is to print the value of something, along with a descriptive label. For instance, we may wish to print out the square root of 10 along with a label. There are several ways to do this:

```
print ( 'The square root of 10 =', np.sqrt(10) )
print ( 'The square root of 10 = %f' % ( np.sqrt(10) ) )
print ( 'The square root of 10 = %4.2f' % ( np.sqrt(10) ) )
s = "The square root of {value:d} is {root:.2f} to two decimal places."
```

```
print ( s.format ( value = 10, root = np.sqrt ( 10 ) ) )
s = "The square root of 10 = " + str ( np.sqrt(10) )
print ( s )
```

9 Printing a table neatly

Suppose we want to print a table of the powers of the first 10 integers, something like this:

```
 n n^2  n^3   n^4
 1   1    1     1
 2   4    8    16
 3   9   27    81
 ...
10 100 1000 10000
```

Generating the data is no problem, but printing it in a neatly lined up format is harder. Let's start with this first try:

```
print ( 'n  n^2  n^3  n^4' )
for n in range ( 1, 11 ):
  print ( n, n**2, n**3, n**4 )
```

The table does indeed get printed out, but instead of seeing neat columns we see ragged lists of values that don't line up. We know that column 1 needs at most 2 spaces, while columns 2, 3 and 4 need 3, 4, and 5 spaces at most. We can use the symbolic print statement in which the place of each numeric value is indicated by a format string beginning with a percent sign, then an indicator of the number of places to be used, followed by a **d** to indicate that these are integers. Remember that, in such a print out, the symbolic string is followed by a percent sign, and then a list of values to be printed, enclosed in a separate set of parentheses. Setting this up is a little more troublesome, but we get a table that is readable, rather than ugly:

```
print ( ' n  n^2   n^3    n^4' )
for n in range ( 1, 11 ):
  print ( '%2d  %3d  %4d  %5d' % ( n, n**2, n**3, n**4 ) )
```

10 More about format strings

As we saw above, one way to control how a variable is printed is to use a format.

The simplest format for an integer is `%d` which simply indicates that an integer is to be printed. If the corresponding value is real, then it will be rounded when printed.

Especially when you want data to appear in columns, you want each value to occupy a specify number of spaces. The format `%4d` indicates that the corresponding integer value is to be printed using four spaces. If the integer is less than 1000, then one or more blanks will be printed initially.

In some contexts, instead of leading blanks, you want leading 0's. The format `%03d`, by starting with a '0', indicates that integer values should occupy 3 spaces, and be padded with initial zeros, if necessary.

```
1  print ( '  I have %d brothers and %d cats' % ( 3, 12 ) )
2  print ( 'A circle of radius 10 has area roughly %d' % ( 10 * np.pi ) )
3  for i in range ( 1, 5 ):
4    print ( '%d  %2d  %4d' % ( i, i**2, i**4 ) )
5  for i in range ( 1, 5 ):
6    print ( '%01d  %02d  %04d' % ( i, i**2, i**4 ) )
```

Real or decimal numbers can be printed in *exponential format*, *floating format* or *general format*. Exponential format is allowed to print the value in scientific notation; floating format prints all the whole digits of the value, and may include some decimal values; the general format makes the better choice between exponential format (for very large and small values) and floating format (for values not too small or large).

A real number format will involve the symbol 'e', 'f', or 'g', possibly preceded by a value of the form `s.d` where `s` is the total number of spaces to be used to print this number, and `d` is the number of spaces reserved for the decimal part. If there is no `s.d` value, then the number is printed with default values. If `d` is omitted, then a default number of decimals will be used.

Typical formats might be `%12.4e`, `%16.2f`, or `%15g`. Consider the results of the following:

```
print ( '  The value of pi is', np.pi )
print ( '  The value of pi is %g' % ( np.pi ) )
print ( '  The value of pi is %5.3g' % ( np.pi ) )
print ( '  The value of pi is %5.3f' % ( np.pi ) )
print ( '  The value of 5^10 is %e' % ( 5**4 ) )
print ( '  The value of 5^10 is %f' % ( 5**4 ) )
print ( '  The value of 5^10 is %g' % ( 5**4 ) )
```

Strings have a very simple format symbol of `s`. If a specific number of spaces are to be allocated to the string, then this value precedes the `s`. By default, the string will be printed so that it terminates at the rightmost. If the string is shorter than the number of spaces, the remainder will be blank. If it is longer, then the whole string will be printed, but will extend further than expected.

```
print ( '  First        Last name   Age' )
print ( '  --------     ----------  ---' )
print ( '  %8s     %10s  %d' % ( 'Viktor', 'Frankenstein', 35 ) )
print ( '  %8s     %10s  %d' % ( 'Andy', 'Warhola', 45 ) )
print ( '  %8s     %10s  %d' % ( 'Engelbert', 'Humperdinck', 55 ) )
```

If you prefer strings to be "left justified", that is, to start at the first space that is allocated, simply precede the number of spaces by a minus sign:

```
print ( '  First        Last name   Age' )
print ( '  --------     ----------  ---' )
print ( '  %-8s     %-10s  %d' % ( 'Viktor', 'Frankenstein', 35 ) )
print ( '  %-8s     %-10s  %d' % ( 'Andy', 'Warhola', 45 ) )
print ( '  %-8s     %-10s  %d' % ( 'Engelbert', 'Humperdinck', 55 ) )
```

It is often advisable to print a string with quotes, to delimit it from other text. This is especially useful when a filename or plot title is displayed:

```
print ( '  Graphics saved as "' + filename + '"' )
```

11 Combining several print statements on one line

Normally, the output of each print statement shows up on a new line. This is because each print statement automatically includes a *new line* character at the end. Sometimes, you actually want several print statements to appear on a single line. To do this, you can suppress the new line character with the argument `end = ''`. If three print statements should appear on one line, then the first two will need to suppress the new line, but the last print statement should have the usual form:

```
print ( 'The square root of 10 =', end = ''  )
print ( '%4.2f' % ( np.sqrt(10) ), end = '' )
print ( 'to two decimal places.' )
```

12 Using tab and newline characters

There are several characters which don't appear on your keyboard, but which can be useful in dealing with text. Two of the most useful are

```
'\n'  newline  shifts output to the next line
'\t'  tab      shifts output to the next column
```

Using the newline character, it is possible for a single print statement to generate several lines of output:

```
print ( 'Polonius: What do you read, my lord?\nHamlet: Words, words, words!' )
```

If we include some tab characters, we can get the texts to line up:

```
print ( 'Polonius: \tWhat do you read, my lord?\nHamlet: \tWords, words, words!' )
```

Note that if you actually want to print strings like \n and \t, you have to use two backslashes (\\) to describe them.

```
print ( '' )
print ( '  To actually print strings like \\n and \\t, use TWO backslashes (\\).' )
```

Similarly, to print a single quote in a string that is delimited by single quotes, precede it by a backslash:

```
print ( '' )
print ( '  That quote is from Hamlet\'s meeting with Polonius.' )
```

You can make simple tables using tab characters, but unless you know how to change the tab settings, you may find that some long text stretches over multiple columns, and so makes the columns no longer line up:

```
print ( '' )
print ( '  Characters in Hamlet:' )
print ( '  Gertrude: \tThe queen \t(poisoned)' )
print ( '  Horatio:  \tHamlet\'s friend \t(alive at end)')
print ( '  Ophelia: \tHamlet\'s fiancee \t(drowned)' )
print ( '  Polonius: \tA courtier \t(stabbed)' )
```

13 Interactive user input

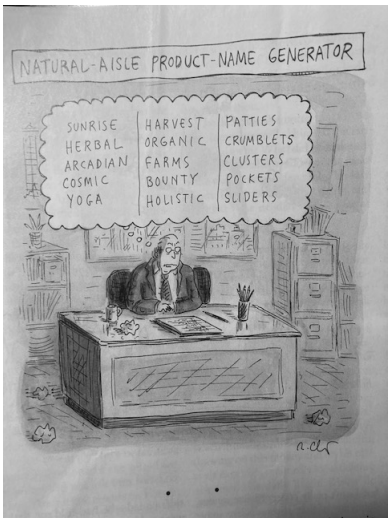
An interactive Python function allows the user to specify certain quantities while it is running. These quanties are usually simple numbers or 'Yes/No' choices or string to be used for file names or titles. In order for the user to know when to respond, the function usually first prints out a request for the data, called a *prompt*, and the user then types a response.
For instance, when the user runs an interactive program to compute the body mass index (BMI), the process might appear like this:

```
python3
>>> bmi
This program computes your body mass index.
Enter your height in inches: 70
Enter your weight in pounds: 145
Your BMI is 20.8
```

We have already seen a program to compute the BMI, but in that case, the height and weight came as inputs to the function. Now, instead, we need to ask the user to supply this information. This is done with the command `input(prompt)`. The BMI program might look like this:

```
print ( 'This program computes your body mass index.' )
height = input ( 'Enter your height in inches: ' )
weight = input ( 'Enter your weight in pounds: ' )
bmi = 703 * weight / height**2
print ( 'Your BMI is', bmi )
```

14 A program challenge



The cartoon suggests a way to generate names for health foods by selecting one word from each column. Thus, one new name would be *COSMIC FARMS PATTIES*.

Write a Python program that produces all the possible names suggested by this process. Your output should be a table of 125 names. The three words in the name should be separated by blanks. It might be nice if only the first letter of the first word was capitalized *Cosmic farms patties*.

You should be able to write a 4 line program that does this!