Reto de lectura 1

Durante el reto, puedes recurrir a los diccionarios y glosarios igual que a tu material de clase.

Section 1. _____

Paragraph 1. Besides numbers, Python can also manipulate strings. Python expresses strings in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result 2. Symbol \ can be used to escape quotes:

```
>>>
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> 'doesn\'t'  # use \' to escape the single quote...
"doesn't"
>>> "doesn't"  # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
```

Illustration 1

Paragraph 2. In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. This looks different from the input (the enclosing quotes could change), but the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes. If a string contains double quotes, it is enclosed in single quotes. The print() function produces a more readable output. It omits the enclosing quotes and prints escaped and special characters.

```
>>>
>>> '"Isn\'t," they said.'
'"Isn\'t," they said.'
>>> print('"Isn\'t," they said.')
"Isn't," they said.
>>> s = 'First line.\nSecond line.'  # \n means newline
>>> s  # without print(), \n is included in the output
'First line.\nSecond line.'
>>> print(s)  # with print(), \n produces a new line
First line.
Second line.
```

Illustration 2

Paragraph 3. If you don't want characters prefaced by \ to be interpreted as special characters, you can use *raw strings* by adding an r before the first quote:

```
>>>
>>> print('C:\some\name')  # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name')  # note the r before the quote
C:\some\name
```

Illustration 3

Paragraph 4. String literals can span multiple lines. One way is using triple-quotes: """..." or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. The following example:

```
print("""\
Usage: thingy [OPTIONS]
    -h                  Display this usage message
    -H hostname             Hostname to connect to
""")
```

Illustration 4

produces the following output (note that the initial newline is not included):

```
Usage: thingy [OPTIONS]
    -h                  Display this usage message
    -H hostname             Hostname to connect to
```

Illustration 5

Section 2. _____

Paragraph 5. Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain different typed of items, but this is not that usual.

```
>>>
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Illustration 1

Paragraph 6. Like strings (and all other built-in sequence types), lists can be indexed and sliced:

```
>>>
>>> squares[0]  # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing returns a new list
[9, 16, 25]
```

Illustration 2

Paragraph 7. All slice operations return a new list containing the requested elements. This means that the following slice returns a shallow copy of the list:

```
>>>
>>> squares[:]
[1, 4, 9, 16, 25]
```

Illustration 3

Paragraph 8. Lists also support operations like concatenation:

```
>>>
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Illustration 4

Paragraph 9. Unlike strings, which are immutable, lists are a mutable type, i.e. it is possible to change their content:

```
>>>
>>> cubes = [1, 8, 27, 65, 125]  # something's wrong here
>>> 4 ** 3  # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64  # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Illustration 5

Paragraph 10. You can also add new items at the end of the list, by using the append() *method* (we will see more about methods later):

```
>>>
>>> cubes.append(216)  # add the cube of 6
>>> cubes.append(7 ** 3)  # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Illustration 6

Paragraph 11. Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
>>>
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Illustration 7

Paragraph 12. It is possible to nest lists (create lists containing other lists), for example:

```
>>>
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Illustration 8

Section 3. _____

Paragraph 13. Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the Fibonacci series as follows:

```
>>>
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
...
0
1
1
2
3
5
8
```

Illustration 1

This example introduces several new features. Here are 2:

- **Paragraph 14.** The first line contains a *multiple assignment*: the variables a and b simultaneously get the new values 0 and 1. On the last line this is used again. This shows that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.

- **Paragraph 15.** The *body* of the loop is ***indented***: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.