Python Workshop

5. String operations

Quick jump: String construction | String formatting | Advanced topics

String and List

- **String** and List are very similar. In fact, string can be seen as a list/tuple of characters.

```
myList = [1, 2, 3]
myStr = "abc"
print('Length of', myList, "is", len(myList))
print('Length of', myStr, "is", len(myStr))
```

String in for-loop

- We can iterated a string in a **for-loop**:

```
mySTr = "abc"
for c in mySTr:
    print(c)
```

- We can also use a string in a **generator**:

```
mySTr = "abc"
g = [c + c for c in mySTr]
print(g)
```

Slicing

- **Slicing** is also possible for strings.

```
myStr = "abcdefg"
print(myStr[1:4])
print(myStr[1:6:2])
```

Output:

```
bcd
bdf
```

String is Immutable

- However since string is **immutable**, we cannot assign values to an index or a slice.

```
myStr = "abcdefg"
myStr[1] = 'X'
myStr[1:4] = 'XXX'
```

- The above code will throw an **error**.
- We need to construct the string using concatenation or string formatting.

Self-learning topics (~40min)

- String construction
- String formatting
- Advanced topics:
  - Better string construction
  - Character code
- You are encouraged to copy-and-paste the code to Spyder and test them.

String construction

String construction

- Suppose we try to produce a string that shows the result of adding two values:

```
a = 1
b = 2
c = a + b
s = a + "+" + b + "=" + c
print(s)
```

- The above code will throw an error when executed.

String concatenation

- Operator `+` can be used to concatenate two strings, but it cannot concatenate a string and a number.
- We need to **convert** the non-string value to a string using the `str()` function.

```
a = 1
b = 2
c = a + b
s = str(a) + "+" + str(b) + "=" + str(c)
print(s)
```

String repeatition

- Operator `*` can be used to generate a repeated sequence of strings.

```
print('=-=' * 10)
```

Output:

```
=-==-==-==-==-==-==-==-==-==-=
```

Exercises (1)

Exercise 1 (1)

- Write a program that reads an input string, and print a list of strings following a specific pattern as shown below.
- Suppose the input string is `hello`, the output should be:

```
hellohello
_ellohell
__llohel
___lohe
____oh
```

Exercise 1 (2)

- Another example output for the input `foobar`:

```
foobarfoobar
_oobarfooba
__obarfoob
___barfoo
____arfo
_____rf
```

Exercise 2 (1)

- Write a program that reads an input string, and print a strings following a specific pattern as shown below.
- Suppose the input string is `hello`, the output should be:

```
hhehelhellhelloellolloloo
```

- The output is the concatenation of `'h'`, `'he'`, `'hel'`, `'hell'`, `'hello'`, `'ello'`, `'llo'`, `'lo'`, and `'o'`.

Exercise 2 (2)

- Sample input/output as follow:

Input	Output
blah	bblblablahlahahh
foobar	ffofoofoobfoobafoobaroobarobarbararr

String formatting

Formating strings

- Another effective way of string construction is the use of **f-string**. f-string is available in Python 3.6 .
- f-string is defined by adding character `f` before a string literal, which includes placeholders for values. For example:

```
a = 1
b = 2
c = a + b
print(f'{a} + {b} = {c}')
```

- In the example above, `{a}` is a placeholder for `a`, the value of `a` will be placed there.

Using an expression in the placeholder

- The placeholder is defined by `{}`, which evaluate an expression, so we can also do this:

```
a = 1
b = 2
print(f'{a} + {b} = {a + b}')
```

Output:

```
1 + 2 = 3
```

format() and %

- You may encounter older Python codes that use other string formatting methods.
- `format()` in a relatively newer method:

```
print('{} + {} = {}'.format(a, b, a + b))
```

- `%` is widely used in Python 2:

```
print('%d + %d = %d'%(a, b, a + b))
```

- We recommend the use of f-string if possible, it is up to you to learn the format of the older methods by yourselves.

Adding format specifiers

- We can add a colon `:` in the placeholder, and specify a set of format specifiers.
- For example, to set the **minimum** length:

```python
x = "abc"
y = "abcdefg"
print(f'|{x:5}|{y:5}|')
```

Output:

```
|abc  |abcdefg|
```

Maximum length specifiers

- **Maximum** length is specified after a dot(`.`).

```
x = "abc"
y = "abcdefg"
print(f'|{x:.5}|{y:.5}|')
```

Output:

```
|abc|abcde|
```

- Note that `y` is truncated because of the length limit.

Min and Max

- We can specify both minimum and maximum length:

```
x = "abc"
y = "abcdefg"
print(f'|{x:5.5}|{y:5.5}|')
```

Output:

```
|abc  |abcde|
```

Alignment

- With minimum length, we can also specify an **alignment** by adding symbol `<`, `^`, or `>` before the specifier:

```python
x = "abc"
print(f'|{x:<5}|{x:^5}|{x:>5}|')
```

Output:

```
|abc  | abc |  abc|
```

- You can see that `<`, `^`, and `>` corresponds to left, center, and right alignment respectively.

Alignment

- Additionally, a **padding scheme** can also be specified by adding the padding character before the alignment:

```
x = "abc"
print(f'|{x:_<5}|{x:*^5}|{x:@>5}|')
```

which prints:

```
|abc__|*abc*|@@abc|
```

String formatting with numbers

Format specifiers for numbers

- By default all values will be formatted as string.
- To format a value as number, we add a `d` (for integer) or `f` (for floating point) at the end of the specifier in the place holder:

```
a = 123
b = 456.789
print(f'|{a:d}|{b:f}|')
```

Output:

```
|123|456.789000|
```

Number base

- Apart from d for decimal values, we can also output integer in binary (b), octal (o) or hexadecimal (x or X) forms.

```
a = 123
print(f'10: {a:d}, 2: {a:b}, 8: {a:o}')
print(f'16: {a:x}, {a:X}')
```

Output:

```
10: 123, 2: 1111011, 8: 173
16: 7b, 7B
```

- For hexadecimal values, x or X control the letter case for a to f.

Min for Numbers

- For integer, we can set the **minimum** length:

```
a = 123
a2 = 1234567
print(f'|{a:5d}|{a2:5d}|')
```

Output:

```
|  123|1234567|
```

- Note that numbers are **right-aligned** by default.

Min for floating

- For floating point values, specifying only the minimum length specifier may has no effect as there is a default precision setting of 6 decimal places:

```
b = 456.789
print(f'|{b:7f}|{b:9f}|{b:11f}|')
```

Output:

```
|456.789000|456.789000| 456.789000|
```

Precision specifier

- For floating point values, the specifier after the dot `.` specifies the **precision**:

```
b = 456.789
print(f'|{b:.2f}|{b:.3f}|{b:.4f}|')
```

Output:

```
|456.79|456.789|456.7890|
```

- There is no precision specifier for integers.

Min and precision

- Minimum length specifier becomes more meaningful when precision is specified:

```
b = 456.789
print(f'|{b:7.2f}|{b:8.4f}|{b:9.4f}|')
```

Output:

```
| 456.79|456.7890| 456.7890|
```

- Notice how the first value is rounded.

0-padding

- For any numbers, adding `0` before the specifier pad the value with zeros.

```
print(f'|{a:05d}|{-a:05d}|{b:07.2f}|{b:09.4f}|')
```

Output:

```
|00123|-0123|0456.79|0456.7890|
```

Sign alignment

- Finally, we can add `+` to force a positive sign, or add a space to force a space for positive value. This help aligning positive values with negative values.

```
a = 123
print(f'|{a:d}|{a:+d}|{a: d}|')
print(f'|{-a:d}|{-a:+d}|{-a: d}|')
```

Output:

```
|123|+123| 123|
|-123|-123|-123|
```

Variable specifier

- We can also use another placeholder in any part of the specifier. This allows **variables** to be used in the specifiers:

```
b = 456.789
len = 5
prec = 2
print(f'|{b:{len}.{prec}f}|{b:{len+2}.{prec}f}|')
print(f'|{b:{len}.{prec+2}f}|{b:{len+4}.{prec+2}f}|')
```

Output:

```
|456.79| 456.79|
|456.7890| 456.7890|
```

Exercises (2)

Exercise 3 (1)

- Consider the following code finding π using **Leibniz formula:**

```
n = 10
pi = 0
for i in range(n):
    pi += (-1) ** i * 4 / (2 * i + 1)
print(pi)
```

Exercise 3 (2)

- The result of PI with n equals 1, 10, 100, ..., 100000 can be summarized in a table:

```
+------+--------+
|  n   |   pi   |
+------+--------+
|     1|4.000000|
|    10|3.041840|
|   100|3.131593|
|  1000|3.140593|
| 10000|3.141493|
|100000|3.141583|
+------+--------+
```

Exercise 3 (3)

- Write a program that reads one input integer, which controls the number of rows to be printed in the previous table.
- For the example in the previous slide, the input value should be `6`.
- You should format the table so that the values are properly aligned. You can decide on the exact format of the table yourselves.

Exercise 3 (4)

- Here is another sample output when input value equals 3.

```
+---+--------+
| n |   pi   |
+---+--------+
|  1|4.000000|
| 10|3.041840|
|100|3.131593|
+---+--------+
```

Advanced topics

- Remember: you may skip this section and come back later.

Better string construction

Concatenation performance

- Consider the following example:

```
a = "...100 characters..."
s = ""
for i in range(10):
    s += a
```

- Every time the statement `s += a` is executed, a new string is created and all characters are copied to a new string.
- What happens if the for-loop is running on `range(1000)`?
- Number of characters copied will be huge!

Better string construction

- For better performance, it is very common that string is constructed by the following steps:
  - Constructng a **list** of strings of different parts
  - **Join** the list of strings into a single string.
- In this way, we avoided generating intermediate concatenation results.

join() (1)

- To join a list of string, we can use the `join()` function.
- The statement `x.join(y)` will join the list of string `y` using string `x`. For example:

```
a = "aaa"
b = "bbb"
c = "ccc"
print(','.join([a, b, c]))
```

Output:

```
aaa,bbb,ccc
```

join() (2)

- To concatenate a list of strings, we join them with an empty string:

```
a = "aaa"
b = "bbb"
c = "ccc"
print(''.join([a, b, c]))
```

Output:

```
aaabbbccc
```

Character code

Character code

- Each character is represented by a code internelly in Python.
- This is also true for most programming languages.
- We can convert a character to its code by using the `ord()` function, or the `chr()`function for the reverse.

ord() and chr()

- Here is an example printing the code for character a:

```
c = "a"
a = ord(c)
c2 = chr(a)
print(c, "has a code of", a)
print(a, "is the code of the character", c2)
```

Output

```
a has a code of 97
97 is the code of the character a
```

Use of character code

- Since the character code of `a` to `z` are consecutive, we can easily convert letter `a` to `z` to a range of `0` to `25`. For example:

```
c = "k"
ord_a = ord('a')
print(c, "is", ord(c) - ord('a'), "letters after 'a'")
```

Output:

```
k is 10 letters after 'a'
```

Case checking

- We can check the letter case of a character using a similar method:

```
def testCase(c):
    if ord(c) >= ord('a') and ord(c) <= ord('z'):
        print(c, "is a lower case letter")
    elif ord(c) >= ord('A') and ord(c) <= ord('Z'):
        print(c, "is a capital letter")
    else:
        print(c, "is not a letter")

testCase("k")
testCase("K")
testCase("*")
```

- We will discuss the use of function in the next section.

ROT13

- As an example, we can implement ROT13, a substitution cipher in this way:

```python
def rot13(text):
    return "".join([ rot13c(c) for c in text ])

def rot13c(c):
    lowerc = ord(c) - ord('a')
    upperc = ord(c) - ord('A')
    if lowerc >=0 and lowerc < 26:
        return chr(ord('a') + (lowerc + 13) % 26)
    elif upperc >=0 and upperc < 26:
        return chr(ord('A') + (upperc + 13) % 26)
    else:
        return c

print(rot13("Why did the chicken cross the road?"))
print(rot13("Gb trg gb gur bgure fvqr!"))
```