CptS 111, Spring 2023
Lect. #20, Apr. 3, 2023
Class Notes

Today's Agenda:

1. `eval()` function redux
2. String formatting
3. String formattting using modulo-formatting
4. String formatting using the `.format()` method
5. String methods

Ch. 8

Strings

We were introduced to the use of strings many chapters ago, and we've used them now as both literals and iterables. In Ch. 8, we learn more about the use of strings in Python.

1. `eval()` Function

In PA #5, you used the `eval()` function to convert a text form of a dictionary into a Python dictionary construct. The `eval()` function can be used for so much more. It essentially evaluates a string argument and does as the argument instructs or converts the argument based on its form. Examples make this much more clear.

Recall that we can't convert a string float to an integer using the `int()` function. We can, however, use the `eval()` function to do this.

In [1]:
```
# Use eval() to convert a string float to a float.

num = eval('1.618')
print(f'num equals {num} and the type of num is {type(num)}.')
```

```
num equals 1.618 and the type of num is <class 'float'>.
```

In [2]:
```
# Use eval() to make a calculation

print(eval('25 - (2 ** 8)'))
```

```
-231
```

In [3]:
```python
# Use eval to allow multiple input values

ht, wt = eval(input('Enter height and weight separated by a comma: '))
print(f'height is {ht} inches and weight is {wt} pounds')
print(f'type of height is {type(ht)}, and type of weight is {type(wt)}')
```

```
Enter height and weight separated by a comma: 72.5,160
height is 72.5 inches and weight is 160 pounds
type of height is <class 'float'>, and type of weight is <class 'int'>
```

2. String Formatting

In Python, there are a number of ways to format a string. In Ch. 3, we learned how to format strings using f-string formatting together with replacement fields, i.e., the sets of curly braces, and format specifiers, e.g., `{:<24}` . Today, we're going to learn two other ways of formatting strings. The first is one that was used in Python 2.x referred to as modulo-formatting because of its use of %. Even after introduction of the `.format()` method in Python 2.6, the second way of formatting we'll cover today, coders continued to use modulo-formatting. Modulo-formatting was popular because coders were used to it and because it was easier to use than the `.format()` method. Because you're likely to run across all three approaches, it's a good idea to be familiar with them all.

3. String Formatting Using Modulo-Formatting and Conversion Specifiers

One way to print variables to stdout is to use conversion specifiers. There are others, but we will typically use the following **conversion specifiers**:

```
%s: use to write variable as a string
%d: use to write variable as an integer
%f: use to write variable as a float
```

`%s` and `%d` are straightforward; however, we may want to impose some constraints on `%f` .

Template for using conversion factors for string formatting:

```
'<text> %s, <text> %d, and <text> %f.' % (var1, var2, var3)
```

If only one variable is used, the parentheses aren't needed, i.e., you don't need to use a tuple.

In [4]:
```python
# Modulo-formatting of a string variable

cat = 'dog'
print('I love my cat named %s!' % cat)
```

```
I love my cat named dog!
```

In [5]:
```python
# Modulo-formatting converting a float to an integer

answer = 42.0
print('The answer to life, the universe, and everything is %d.' % answer)
```

```
The answer to life, the universe, and everything is 42.
```

```
In [6]: # Modulo-formatting converting a float to a string

        print('The answer to life, the universe, and everything is %s.' % answer)
```

```
The answer to life, the universe, and everything is 42.0.
```

In the example above, we've used the conversion specifier `%s` which converts a float to a string, but as it's a string float, it prints as a float.

```
In [7]: # Attempt to convert string float to an integer using %d

        answer = '42.0'
        print('The answer to life, the universe, and everything is %d.' % answer)
```

```
------------------------------------------------------------------------
----
TypeError                                 Traceback (most recent call l
ast)
Input In [7], in <cell line: 4>()
      1 # Attempt to convert string float to an integer using %d
      3 answer = '42.0'
----> 4 print('The answer to life, the universe, and everything is %d.'
% answer)

TypeError: %d format: a real number is required, not str
```

In the example above, we see that we still can't convert a string float to an integer. The only way to do so is by using `eval()` .

```
In [8]: # Modulo-formatting of a float

        savings = 2300
        annual_interest = 0.0225 * savings
        print('Interest earned in one year: $%f' % annual_interest)
```

```
Interest earned in one year: $51.750000
```

This doesn't look very good because we don't want all the trailing zeros. As with f-string formatting, we can use `%.2f` to restrict the value printed to two decimal places.

```
In [9]: # Modulo-formatting with constraint on %f

        savings = 2300
        annual_interest = 0.0225 * savings
        print('Interest earned in one year: $%.2f' % annual_interest)
```

```
Interest earned in one year: $51.75
```

Note that as with f-string formatting, we can use modulo-formatting in the `input()` function.

In [10]:
```
# Use of modulo-formatting in the input() function

i = 1
cost = float(input('Enter the cost of item #%d: ' % i))
```

```
Enter the cost of item #1: 42
```

In [11]:
```
# Modulo-formatting with more than one argument

print('Hello, %s and %s!' % ('Yin', 'Yang'))
```

```
Hello, Yin and Yang!
```

4. String Formatting Using the `.format()` Method

The `.format()` method of string formatting is similar to that of f-string formatting. As with f-strings, the `.format()` method uses replacement fields and format specifiers.

String Formatting Template

```
    '  '.format() = method to format a string
```

where

```
    '  ' = format string: combination of string literals and replacem
    ent fields
     {}  = replacement field
```

When a replacement field uses no numbers, then we simply use a 1-to-1 mapping based on position. For example:

In [12]:
```
# inferred positional

print('Hello, {} and {}!'.format('Yin', 'Yang'))
```

```
Hello, Yin and Yang!
```

We can also use numeric indexes. For example:

In [13]:
```
# numeric positional and use of lvalues as arguments

me ='Me'
you = 'You'
print('Hello, {0}, {1}, and {0}!'.format(me, you))
```

```
Hello, Me, You, and Me!
```

Note the correspondence between index numbers and positions. We can also use named replacement fields. For example:

In [14]:
```python
# named (keyword)

print('Hello, {n1}, {n2}, and {n1}!'.format(n1 = 'Me', n2 = 'You'))
```

```
Hello, Me, You, and Me!
```

As mentioned above, format specifiers can be used with the `.format()` method. They're the same ones we learned for f-strings. For example,

In [15]:
```python
# len('gray') > specified width so ignored.

print('{:10} and {color:3}'.format('crimson', color='gray'))
```

```
crimson    and gray
```

5. String Methods

Use `dir('')` or `dir(str)` to see what string methods are available.

In [16]:
```python
# See what string methods are available in Python

print(dir(''))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__d
oc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem
__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subcla
ss__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod
__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshoo
k__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswit
h', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islowe
r', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'joi
n', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefi
x', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartitio
n', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

And use `help()` for information on a particular method.

In [17]:
```python
# Use help() to get information about particular string method

help(str.upper)
```

```
Help on method_descriptor:

upper(self, /)
    Return a copy of the string converted to uppercase.
```

Recall that in general, methods can be nested just like functions. This is known as chaining.

- functions: `outer(inner())` -- inner performed first, then outer

- methods: `identifier.left().right()` -- left performed first, then right

A. Case methods: `.capitalize()`, `.title()`, `.swapcase()`, `.lower()`, `.upper()`

In [18]:
```
# Capitalize first letter of string using .capitalize()

s = 'the cat in the hat'
s.capitalize()
```
Out[18]:
```
'The cat in the hat'
```

In [19]:
```
# Capitalize first letter of each word in string using .title()
# We used this in PA #5

s.title()
```
Out[19]:
```
'The Cat In The Hat'
```

In [20]:
```
# Use .swapcase() to swap cases in string

s.swapcase()
```
Out[20]:
```
'THE CAT IN THE HAT'
```

In [21]:
```
# Use .lower() to change all string characters to lowercase
# We also used this in PA #5

s.lower()
```
Out[21]:
```
'the cat in the hat'
```

In [22]:
```
# Use .upper() to change all string characters to uppercase

s.upper()
```
Out[22]:
```
'THE CAT IN THE HAT'
```

In [23]:
```
# Note, however, that none of these methods changed s; why?

s
```
Out[23]:
```
'the cat in the hat'
```

In [24]:
```
# But string methods can be non-void

t = s.upper()
print(t)
print(s)
```
```
THE CAT IN THE HAT
the cat in the hat
```

```
In [25]: # We can always reassign an lvalue; use chained methods

         s = 'JOSEPH GORDON-LEVITT'
         s.lower().capitalize()

Out[25]: 'Joseph gordon-levitt'
```

```
In [26]: # Order matters: left to right

         s.capitalize().lower()

Out[26]: 'joseph gordon-levitt'
```

B. `.count()`, `.find()`, and `.replace()` Methods

```
In [27]: # Use of .count() to count occurrences of character (case matters!)

         phrase = 'She sells seashells down by the seashore.'
         phrase.count('s')

Out[27]: 7
```

```
In [28]: # Count all occurrences of s, regardless of case

         phrase.lower().count('s')

Out[28]: 8
```

```
In [31]: # Use .find() to find index of first occurrence of character(s)
         # If no occurrence, -1 returned

         phrase.find('s')

Out[31]: 4
```

```
In [32]: # Use .replace() to replace chr or substring; use .lower() first
         # so uppercase S also replaced

         phrase.lower().replace('s', '$')

Out[32]: '$he $ell$ $ea$hell$ down by the $ea$hore.'
```

```
In [33]: # We can restrict number of times chr is replaced

         phrase.replace('s', '$', 3)

Out[33]: 'She $ell$ $eashells down by the seashore.'
```

```
In [34]: # Methods didn't change phrase; why?

         phrase

Out[34]: 'She sells seashells down by the seashore.'
```

In [35]:
```python
# However, we can change by reassigning lvalue

phrase = phrase.replace('She', 'He')
phrase
```

Out[35]:
```
'He sells seashells down by the seashore.'
```

C. `.lstrip()`, `.rstrip()`, and `.strip()` Methods: Important for removing whitespace!

In [36]:
```python
# Two tabs on the left; two newlines on the right

cheer = '\t\tGo, Cougs!\n\n'
print(cheer)
print('Here I am.')
```

```
		Go, Cougs!


Here I am.
```

In [37]:
```python
# Remove whitespace on the left, i.e., tabs

print(cheer.lstrip())
print('Here I am.')
```

```
Go, Cougs!


Here I am.
```

In [38]:
```python
# Remove whitespace on the right, i.e., newlines

print(cheer.rstrip())
print('Here I am.')
```

```
		Go, Cougs!
Here I am.
```

In [39]:
```python
# Remove whitespace from both sides

print(cheer.strip())
print('Here I am.')
```

```
Go, Cougs!
Here I am.
```

Note that `.strip()` methods don't remove whitespace between text but only leading and trailing whitespace.