Demo - if __name__ == '__main__'
Exercises – testing and debugging code
Exercises - Python string formatting

(1) Demo – if __name__ == '__main__'

- We can import user-defined modules (.py files), the same as modules from the standard library.
- In such cases, we want Python to execute the function definitions, but not any function calls (“scripting” code).
- Fortunately, we can use Python’s __name__ variable to check whether a function is being executed in the __main__ namespace, or has been imported.

(2) Exercises – testing and debugging code
def findlast(s, ch):
    '''(str, str) -> int

    return position of last occurrence of ch s, or -1 if ch does not occur in s
    >>> findlast('mississippi', 'm')
    0
    '''
    lastpos = -1
    for idx in range(len(s)):
        if s[idx] == ch:
            lastpos = idx
    return lastpos

Generate test cases to check that function findlast works per its docstring specification, including normal test cases:

edge (boundary) test cases:

test cases for different input formats:

test cases for different expected outputs:

Execute the test cases (by hand and then using IDLE); fix any bugs; re-test until code is stable.
(3) Exercises – Python string formatting – review tutorial below before trying these):

(1) Write a Python function to generate letterhead for a given name. The function should take in a single string and print it out surrounded by your letterhead. A simple example is here, but be as creative as you like.

***************
*       CIS 210     *
***************

(2) Write a Python function that returns a number that has been “stringified” into money format – meaning it has a dollar sign, uses thousands comma separator, and is rounded to 2 digits. moneyformat should take in a number and return a string.

For example:

```python
>>> moneyformat(5)
$5.00
>>> moneyformat(1420.8823423)
$1,420.88
```

(3) Given two lists of strings: one containing class titles and the second containing the corresponding class location, write a Python function that prints a 2-column table with neatly aligned columns. How wide should the columns be?

For example,

```
Name       Location
CIS 210    WIL 100
CIS 211    LLCS 101
```
Python string formatting

The Python string format method is really a mini-language (similar to turtle functionality).
https://docs.python.org/3.4/library/string.html

Python 3.6 and higher also has f-strings (no format method required):
https://docs.python.org/3/whatsnew/3.6.html#whatsnew36-pep498

Also (another take and includes deprecated-but-C-language-like % string formatting syntax):
https://realpython.com/python-f-strings/

This is a short introduction to the string format method, with some examples of handy string formatting. Notice that the string format method returns a NEW string in which the specified formatting has been applied to the original, or “format”, string.

String format uses {} inside a string to indicate where values should be inserted:

>>> print('Hello, {}.'.format('world'))
Hello, world.

>>> print(f'Hello, world')

Notice that we can do this using variables instead of string literals:

>>> name = 'world'
>>> print('Hello, {}.'.format(name))
Hello, world.

>>> print(f'Hello, {name}.
Hello, world.

Arguments to the format method supply the values that should be inserted in the {} placeholders. Note that these placeholders are PART of the string into which we are inserting. Note that the arguments are simply Python expressions, and can be string literals, integers, variables, etc.

>>> print('Hello, {}.'.format('w' + 'o' + 'r' + 'l' + 'd'))
Hello, world.

>>> print(f"Hello, {'w' + 'o' + 'r' + 'l' + 'd'}.\nHello, world.

Python docs for string formatting:
https://docs.python.org/3/library/string.html#formatstrings
https://docs.python.org/3/library/string.html#format-spec

Old style formatting (see text ch. 5.2.3)

>>> print('Hello, %s' % ('world'))
Hello, world.
Use {{commands}} to provide additional information to the string format method:
- :{[fillchar][align][width][,][decimal places][type]}
- Fillchar – what char to put in open places – used with align
- Align (right >, left < or center ^)
- Width – how many total spots to use
- Comma – use comma for thousands separator
- Decimal places – how many digits after the decimal (better than the round function)
- Presentation types – d for integers, f for floats, b for binary, c for characters, % for percentages
- More fields are available if you look
- All fields are optional if you don’t need them

For example:

```python
>>> my_string = "a = {:,.3f}, b = {:0>6}, c = {:.1%}"
>>> my_string = my_string.format(1.2, 2, 0.3)
>>> print(my_string)
a = 1.200, b = 000002, c = 30.0%

>>> print(f'a = {1.2:.3f}, b = {2:0>6}, c = {0.3:.1%}')
a = 1.200, b = 000002, c = 30.0%
```

Another example:

```python
>>> my_string = 'a = {:,.1f}, b = {:^10}, c = {:.2%}'
>>> my_string = my_string.format(1.2, 2, 0.3)
>>> print(my_string)
a = 1.2, b =     2     , c = 30.00%

>>> print(f'a = {1.2:.1f}, b = {2:^10}, c= {0.3:.2%}')
a = 1.2, b =     2     , c= 30.00%
```

Print with commas example:

```python
>>> my_string = "{:,.}".format(123456789)
>>> print(my_string)
123,456,789

>>> print(f'{123456789:,}')
123,456,789
```

Format/print in binary:
```python
>>> f'{44:b}'
'101100'
>>> print(f'{44:b}')
101100
>>> print('{:b}'.format(44))
101100
```

Now a character:
```python
>>> print(f'{101:c}')
e
>>> print('{:c}'.format(101))
e
```