Strings
Writing strings

- Strings can be written in single quotes, '...', or double quotes, "...",
- These strings cannot contain an “actual” newline
- Certain special characters can be written in these strings by *escaping* them with a backslash
  - \n=newline, \t=tab. \'=single quote, \"=double quote, \\
  =backslash, \u{hhhh}=unicode character hhhh
- Strings can be written in triple quotes, '''...''' or """...""
  - Triply-quoted strings can contain unescaped newline and unescaped single and double quote marks
- A *raw string* is one in which all characters just mean themselves
  - To create a raw string, just prefix it with r or R
  - **Example:** r"abc\txyz\n" contains ten characters, not eight
- These rules are for strings that appear in code; input is always “raw”
Basic operations

- A string is a sequence of characters, and can be treated as such
  
  ```python
  >>> s = "abc123"
  >>> s[2]
  'c'
  >>> s[2:5]
  'c12'
  ```

- Strings are **immutable** (cannot be changed)
  
  ```python
  >>> s[2] = "*

  Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
      s[2] = "*
  TypeError: 'str' object does not support item assignment
  ```

- However, you can create a new string, and even assign it back to the original variable
  
  ```python
  >>> s = s + "xyz"
  >>> s
  'abc123xyz'
  ```
Basic operations II

• Strings can be concatenated (joined) with +
  
  • >>> s = "apple"
  • >>> s + 's'
       'apples'

• Strings can be “multiplied” with *
  
  • >>> 'bum ' * 3 + 'BUM!'
       'bum bum bum BUM!'

• You can use the in and not in tests with strings
  
  • >>> 'cap' in 'escape sequence'
       True
Methods on strings

• These are methods, not functions, so use the method syntax, `string.method()`, not `function(string)`

• Here are some of the more useful methods:
  • `s.isupper()` and `s.islower()` test whether all letters in `s` are uppercase or lowercase, respectively
  • `s.upper()` and `s.lower()` converts all letters in `s` to uppercase or lowercase, respectively
    • Because strings are immutable, these methods return a new string
  • `s.isalpha()` tests if all characters in `s` are alphabetic
  • `s.isdigit()` tests if all characters in `s` are digits
  • `s.isspace()` tests if all characters in `s` are `whitespace`
    • `whitespace` includes spaces, tabs, newlines, and a few other nonprinting characters
Methods on strings II

- `s.lstrip()`, `s.rstrip()`, `s.strip()` removes whitespace from the left end, the right end, or both ends of `s`.
  - These methods don’t change `s`, they return a new string.

- `s.startswith(substring)`, `s.endswith(substring)` test whether `s` starts with, or ends with, `substring`.

- `s.find(substring)` finds `substring` in `s` and returns its index, or `-1` if not found.

- `sep.join(sequence)` inserts the string `sep` between the elements of `sequence`, and returns a new string.
  - Example:
    ```python
    >>> ', '.join(['one', 'two', 'three'])
    'one, two, three'
    ```
Fonts and tabs

- **In a *monospace font***, all characters have the same width: `| | | |wwwww`
  - Programmers generally prefer monospace fonts
- **In a *proportional font***, characters have different widths: `|||wwwww`
  - Proportional fonts are better for almost everything else
- **Tabs** are inherited from typewriters, where you could set mechanical *tab stops*
  - But on a computer, a *tab* is an *actual character* (ASCII value 9) representing an *arbitrary* amount of space
- Try this:
  - In a text editor, make a little table, using tabs to get neat columns
  - Save the file, and open it up in a different text editor
  - Columns *may or may not* still be lined up properly
- If you want neat columns, or other careful control over spacing:
  - With a proportional font, you *must* use tabs to get precise control over spacing
  - With a monospace font, you *should* always use spaces, never use tabs
More string methods

- The methods `ljust`, `rjust`, and `center` will left-justify, right-justify, or center a string in a field of a given width
  - ```python
define s = 'abc'
define s.ljust(6)
   'abc   '
define s.rjust(6)
   '   abc'
define s.center(6)
   ' abc   '
define s.ljust(6, '*')
   'abc***'
```
- As always, these methods return a `new` string
The **format** method

- The **format** method looks for braces, **{}**, in a string, and substitutes values for those braces
  - **format** is mostly used to prepare a string for printing, and often occurs within a call to `print`
  - We will cover only the simplest cases of this very complex method
- Simple substitution, in order:
  - ```
>>> print('one={}, half={}'.format(1, 1/2))
one=1, half=0.5
```  
  - ```
>>> print('Strings {} and tuples {}'.format('abc', (3, 5)))
Strings abc and tuples (3, 5)
```  
- Simple substitution, specifying the order:
  - ```
>>> print('{2} - {0} - {2} - {1}'.format('zero', 'one', 'two'))
two - zero - two - one
```  
- Substitution, specifying the number of characters
  - ```
>>> print('--{:10}--'.format("Hello"))
--Hello     --
>>> print('--{:10}--'.format(12345))
--     12345--
```
Formatting numbers

• Integers can be printed in decimal (default), binary, octal, or hexadecimal
  
  ```python
  >>> print('{:d} | {:b} | {:o} | {:x}'.format(43, 43, 43, 43))
  43 | 101011 | 53 | 2b
  ```

• This can be combined with specifying the number of characters
  
  ```python
  >>> print('{:10d} | {:10X}'.format(43, 43))
  43 |         2B
  ```

• Floating point numbers can be printed in standard or in scientific notation
  • You can specify the precision (number of characters after the decimal point), or you can specify both the width and the precision
    
    Syntax: `{:width.precisionf}` or `{:width.precisione}`
  
  ```python
  >>> print('{:.3f}, {:.3e}'.format(math.pi, math.pi))
  3.142, 3.142e+00
  >>> print('{:10.3f}, {:10.3e}'.format(math.pi, math.pi))
  3.142, 3.142e+00
  ```

• All of the above can be combined with specifying the order
  
  ```python
  >>> print('{1:5.3f}, {0:1.3f}'.format(1/3, 1/2))
  0.500, 0.333
  ```
The End