

LECTURE 5

Strings

STRINGS

- We've already introduced the string data type a few lectures ago. Strings are subtypes of the sequence data type.
- Strings are written with either single or double quotes encasing a sequence of characters.

```
s1 = 'This is a string!'
s2 = "Python is so awesome."
```

- Note that there is no character data type in Python. A character is simply represented as a string with one character.

ACCESSING STRINGS

- As a subtype of the sequence data type, strings can be accessed element-wise as they are technically just sequences of character elements.
- We can index with typical bracket notation, as well as perform slicing.

```
>>> s1 = "This is a string!"
>>> s2 = "Python is so awesome."
>>> print (s1[3])
s
>>> print (s2[5:15])
n is so aw
```

MODIFYING STRINGS

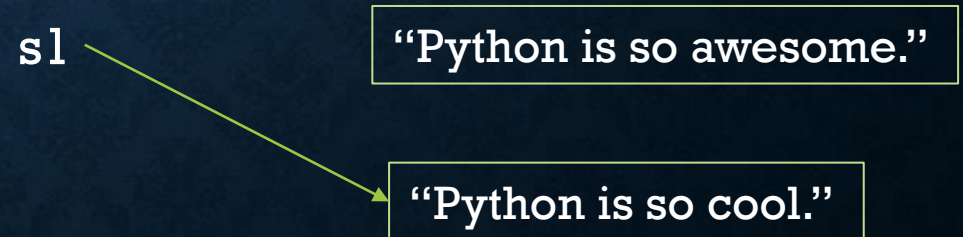
- Strings are *immutable* – you cannot update the value of an existing string object. However, you can reassign your variable name to a new string object to perform an “update”.

```
>>> s1 = "Python is so awesome."  
>>> s1 = "Python is so cool."
```



s1 → "Python is so awesome."

A diagram illustrating the initial state of the variable `s1`. The variable `s1` is on the left, and a horizontal arrow points from it to a rectangular box on the right containing the text "Python is so awesome."



s1 → "Python is so cool."

A diagram illustrating the state of the variable `s1` after reassignment. The variable `s1` is on the left, and an arrow points from it to a rectangular box on the right containing the text "Python is so cool." The previous box from the first diagram is no longer shown, indicating that the reference to the old object has been lost.

MODIFYING STRINGS

Alternatively, we could have done the following:

```
>>> s1 = "Python is so awesome."  
>>> s1 = s1[:13] + "cool."
```

This will create a substring “Python is so ”, which is concatenated with “cool.”, stored in memory and associated with the name `s1`.

The “+” operator can be used with two string objects to concatenate them together. The “*” operator can be used to concatenate multiple copies of a single string object.

We also have `in` and `not in` available for testing character membership within a string.

ESCAPE CHARACTERS

- As a side note, there are a number of escape characters supported by Python strings. The most common ones are:
 - `'\n'` – newline
 - `'\s'` – space
 - `'\t'` – tab

BUILT-IN STRING METHODS

- Python includes a number of built-in string methods that are incredibly useful for string manipulation. Note that these *return* the modified string value; we cannot change the string's value in place because they're immutable!
- `s.upper()` and `s.lower()`

```
>>> s1 = "Python is so awesome."  
>>> print (s1.upper())  
PYTHON IS SO AWESOME.  
>>> print (s1.lower())  
python is so awesome.
```

BUILT-IN STRING METHODS

- `s.isalpha()`, `s.isdigit()`, `s.isalnum()`, `s.isspace()` – return `True` if string `s` is composed of alphabetic characters, digits, either alphabetic and/or digits, and entirely whitespace characters, respectively.
- `s.islower()`, `s.isupper()` – return `True` if string `s` is all lowercase and all uppercase, respectively.

```
>>> "WHOA".isupper()
```

```
True
```

```
>>> "12345".isdigit()
```

```
True
```

```
>>> "\n".isspace()
```

```
True
```

```
>>> "hello!".isalpha()
```

```
False
```


BUILT-IN STRING METHODS

- `str.split([sep[, maxsplit]])` – Split *str* into a list of substrings. The *sep* argument indicates the delimiting string (defaults to consecutive whitespace). The *maxsplit* argument indicates the maximum number of splits to be done (default is -1).
- `str.rsplit([sep[, maxsplit]])` – Split *str* into a list of substrings, starting from the right.
- `str.strip([chars])` – Return a copy of the string *str* with leading and trailing characters removed. The *chars* string specifies the set of characters to remove (default is whitespace).
- `str.rstrip([chars])` – Return a copy of the string *str* with only trailing characters removed.

BUILT-IN STRING METHODS

```
>>> "Python programming is fun!".split()  
['Python', 'programming', 'is', 'fun!']  
>>> "555-867-5309".split('-')  
['555', '867', '5309']  
>>> "***Python programming is fun***".strip('*')  
'Python programming is fun'
```


BUILT-IN STRING METHODS

- `str.capitalize()` – returns a copy of the string with the first character capitalized and the rest lowercase.
- `str.center(width[, fillchar])` – centers the contents of the string *str* in field-size *width*, padded by *fillchar* (defaults to a blank space). See also `str.ljust()` and `str.rjust()`.
- `str.count(sub[, start[, end]])` – return the number of non-overlapping occurrences of substring *sub* in the range *[start, end]*. Can use slice notation here.
- `str.endswith(suffix[, start[, end]])` – return True if the string *str* ends with *suffix*, otherwise return False. Optionally, specify a substring to test. See also `str.startswith()`.

BUILT-IN STRING METHODS

```
>>> "i LoVe pYtHoN".capitalize()
'I love python'
>>> "centered".center(20, '*')
'*****centered*****'
>>> "mississippi".count("iss")
2
>>> "mississippi".count("iss", 4, -1)
1
>>> "mississippi".endswith("ssi")
False
>>> "mississippi".endswith("ssi", 0, 8)
True
```


BUILT-IN STRING METHODS

- `str.find(sub[, start[, end]])` – return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the slice *str[start:end]*. Return -1 if *sub* is not found. See also `str.rfind()`.
- `str.index(sub[, start[, end]])` – identical to `find()`, but raises a *ValueError* exception when substring *sub* is not found. See also `str.rindex()`.
- `str.join(iterable)` – return a string that is the result of concatenating all of the elements of *iterable*. The *str* object here is the delimiter between the concatenated elements.
- `str.replace(old, new[, count])` – return a copy of the string *str* where all instances of the substring *old* are replaced by the string *new* (up to *count* number of times).

BUILT-IN STRING METHODS

```
>>> "whenever".find("never")
3
>>> "whenever".find("what")
-1
>>> "whenever".index("what")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> "-".join(['555', '867', '5309'])
'555-867-5309'
>>> " ".join(['Python', 'is', 'awesome'])
'Python is awesome'
>>> "whenever".replace("ever", "ce")
'whence'
```


THE STRING MODULE

- Additional built-in string methods may be found [here](#).
- All of these built-in string methods are methods of any string object. They do not require importing any module or anything – they are part of the core of the language.
- There is a string module, however, which provides some additional useful string tools. It defines useful string constants, the string formatting class, and some deprecated string functions which have mostly been converted to methods of string objects.

STRING CONSTANTS

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
```


STRING CONSTANTS

```
>>> import string
>>> string.lowercase #locale-dependent
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase #locale-dependent
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.letters # lowercase+uppercase
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.octdigits
'01234567'
>>> print string.punctuation
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

STRING CONSTANTS

- `string.whitespace` – a string containing all characters that are considered whitespace. On most systems this includes the characters space, tab, linefeed, return, formfeed, and vertical tab.
- `string.printable` – string of characters which are considered printable. This is a combination of digits, letters, punctuation, and whitespace.

STRING FORMATTING

- String formatting is accomplished via a built-in method of string objects. The signature is:
`str.format(*args, **kwargs)`
- Note that the `*args` argument indicates that `format` accepts a variable number of positional arguments, and `**kwargs` indicates that `format` accepts a variable number of keyword arguments.

The string on which this method is called can contain literal text or replacement fields delimited by braces `{}`. Each replacement field contains either the numeric index of a positional argument, or the name of a keyword argument. A copy of the string is returned where each replacement field is replaced with the string value of the corresponding argument.

STRING FORMATTING

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
```

```
'a, b, c'
```

```
>>> '{}, {}, {}'.format('a', 'b', 'c')
```

```
'a, b, c'
```

```
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
```

```
'c, b, a'
```

```
>>> '{2}, {1}, {0}'.format(*'abc')
```

```
'c, b, a'
```

```
>>> '{0}{1}{0}'.format('abra', 'cad')
```

```
'abracadabra'
```


STRING FORMATTING

You can also use keyword arguments to the format function to specify the value for replacement fields

```
>>> 'Coords: {lat}, {long}'.format(lat='37.24N', long='-115.81W')
'Coords: 37.24N, -115.81W'
>>> coord = {'lat': '37.24N', 'long': '-115.81W'}
>>> 'Coords: {lat}, {long}'.format(**coord)
'Coords: 37.24N, -115.81W'
```

STRING FORMATTING

Within the replacement field, you are able to access attributes and methods of the object passed as an argument to format. Here, we pass a complex number as an argument, but we access its member attributes in the replacement field.

```
>>> c = 2+3j
>>> '{0} has real part {0.real} and imaginary part {0.imag}.'.format(c)
'(2+3j) has real part 2.0 and imaginary part 3.0.'

>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```


STRING FORMATTING

- There are reserved sequences for specifying justification and alignment within a replacement field.

```
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
>>> '{:*^30}'.format('centered') # use '*' as a fill char
'*****centered*****'
```

STRING FORMATTING

- There are a number of options for formatting floating-point numbers.

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14) # show sign always  
'+3.140000; -3.140000'
```

```
>>> '{: f}; {: f}'.format(3.14, -3.14) # show space for positive  
' 3.140000; -3.140000'
```

```
>>> '{:-f}; {:-f}'.format(3.14, -3.14) # show only minus  
'3.140000; -3.140000'
```

```
>>> '{:.3f}'.format(3.14159) # limit to three dec places  
'3.142'
```


STRING FORMATTING

- There are still quite a few more formatting specifiers that we haven't covered. A list of them is available [here](#).
- We'll now turn our attention back to functions and begin OOP in Python.