Introduction to Variables: Strings

Introduction

As we already know from living in the digital age and the lessons we've already seen, programming is a powerful tool for answering questions about data. It allows us to collect, clean up and format our data and then perform calculations on that data. Much of our digital information is in the form of text, for example song lyrics and emails. To clean up and format that text with Python, we need to become familiar with our first type of data, the String.

Objectives

By the end of this lesson, you will be able to:

- Apply string methods to make changes to a string
- Inspect documentation with various methods

What are Strings?

A lot of information in the world is in the form of text. To capture this information and operate on it in Python we take this text and make it into the **String** (`str`) data type.

Below, we have the name of a cartoon character, Homer Simpson. By putting quotes (`""`) or (`''`) around the name, we create a string.

```
"Homer Simpson"
```

When programmers say *string*, what they mean is text. When programmers say *data type*, they just mean type of data. We can think of `'Homer Simpson'` as an *instance* of the string data type.

Here are a few other types of data in Python that we will talk more about in later lessons:

```
100 # Integer
10.0 # Float
True # Boolean
```

Since there are several types of data in Python we can discover the type of any piece of data by calling, or executing, the `type()` function. By calling or executing a function, we mean running the function so that it executes the code within it.

Let's look at an example below:

> **Note:** Press the shift + enter keys to run the code below. The cell that populates below is the return or output of the type function.

```
type("Homer Simpson")
```

```
str
```

We need to pay attention to what type of data we are working with because they operate differently and have different values as well as functions that we are able to use on them.

For example, to create a new string (or to *initialize* a string) we cannot simply type letters. Instead, we need to be very explicit and tell Python it is about to see some text. We do this by surrounding our text with quotes, `""` . If we don't do that or end our quotation marks too early, Python will throw an error.

```
"This is a properly formatted string!"
```

```
'This is a properly formatted string!'
```

```
"Th"is will throw an error!
```

```
  File "<ipython-input-10-08f2a98e3ef7>", line 1
    "Th"is will throw an error!
                     ^
SyntaxError: invalid syntax
```

> **Note:** double quotes and single quotes can be used interchangeably in Python; however, for readability it is important that we stay consistent. At first, it might seem strange how picky programmers are about details like this, but after a couple of years of coding, you too might end up in a fight like this one!

Strings can contain numbers as well. It may look like 42, but it if it wrapped with `'42'`, then it is a string. See the difference below.

```
type('42')
```

```
str
```

```
type(42)
```

```
int
```

```
type(42.0)
```

```
float
```

Changing Data With Built In Methods

Python is picky like this for a reason. For example, once it knows we are working with a string, it gives us specific functionality for operating on strings. We call this functionality a **function** or a **method**.

Below we have a method that works with a String, but does not work with an integer.

```
'Homer Simpson'.upper()
```

```
'HOMER SIMPSON'
```

```
42.upper()
```

```
  File "<ipython-input-15-8ab5bf46fc2d>", line 1
    42.upper()
           ^
SyntaxError: invalid syntax
```

Yep. Bad news bears.

As we can see in the examples above, we can operate on a datatype using the following format:

- [INSTANCE OF A DATATYPE] [DOT] [METHOD NAME] [PARENTHESES]

Here is an examples that follows this format and returns a `True` or `False` value:

```
"Homer Simpson".endswith('Simpson')
```

```
True
```

```
"Charles Montgomery Burns".endswith('Simpson')
```

```
False
```

As you can see in this notebook, most of our operations on data will follow the data-dot-method_name-parentheses format.

Discovering New Methods

You may be starting to worry about there being too many methods to keep track of. Let's ask Python for help with finding more information about what we can do with strings.

The `help()` function in Python comes built-in and is like an old school Alexa. We give our prompt or *data type* to the `help()` function and it tells us everything it knows about that data type.

Let's see what happens when we type `help(str)`.

```
help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by format_spec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |  
 |  __getitem__(self, key, /)
 |      Return self[key].
 |  
 |  __getnewargs__(...)
 |  
 |  __gt__(self, value, /)
 |      Return self>value.
 |  
 |  __hash__(self, /)
 |      Return hash(self).
 |  
 |  __iter__(self, /)
 |      Implement iter(self).
 |  
 |  __le__(self, value, /)
 |      Return self<=value.
 |  
 |  __len__(self, /)
 |      Return len(self).
 |  
 |  __lt__(self, value, /)
 |      Return self<value.
 |  
 |  __mod__(self, value, /)
 |      Return self%value.
 |  
 |  __mul__(self, value, /)
 |      Return self*value.
 |  
 |  __ne__(self, value, /)
 |      Return self!=value.
 |  
 |  __repr__(self, /)
 |      Return repr(self).
 |  
 |  __rmod__(self, value, /)
 |      Return value%self.
 |  
 |  __rmul__(self, value, /)
 |      Return value*self.
 |  
 |  __sizeof__(self, /)
 |      Return the size of the string in memory, in bytes.
 |  
 |  __str__(self, /)
 |      Return str(self).
 |  
 |  capitalize(self, /)
 |      Return a capitalized version of the string.
 |
 |      More specifically, make the first character have upper case and the rest
lower
 |      case.
 |
 |  casefold(self, /)
 |      Return a version of the string suitable for caseless comparisons.
 |
 |  center(self, width, fillchar=' ', /)
 |      Return a centered string of length width.
 |
 |      Padding is done using the specified fill character (default is a space).
 |
 |  count(...)
 |      S.count(sub[, start[, end]]) -> int
 |
 |      Return the number of non-overlapping occurrences of substring sub in
 |      string S[start:end].  Optional arguments start and end are
 |      interpreted as in slice notation.
 |
 |  encode(self, /, encoding='utf-8', errors='strict')
 |      Encode the string using the codec registered for encoding.
 |
 |      encoding
 |        The encoding in which to encode the string.
 |      errors
 |        The error handling scheme to use for encoding errors.
 |        The default is 'strict' meaning that encoding errors raise a
 |        UnicodeEncodeError.  Other possible values are 'ignore', 'replace' and
 |        'xmlcharrefreplace' as well as any other name registered with
 |        codecs.register_error that can handle UnicodeEncodeErrors.
 |
 |  endswith(...)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      suffix can also be a tuple of strings to try.
 |
 |  expandtabs(self, /, tabsize=8)
 |      Return a copy where all tab characters are expanded using spaces.
 |
 |      If tabsize is not given, a tab size of 8 characters is assumed.
 |
 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  format(...)
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions from args and
kwargs.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  format_map(...)
 |      S.format_map(mapping) -> str
 |
 |      Return a formatted version of S, using substitutions from mapping.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  index(...)
 |      S.index(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  isalnum(self, /)
 |      Return True if the string is an alpha-numeric string, False otherwise.
 |
 |      A string is alpha-numeric if all characters in the string are alpha-
numeric and
 |      there is at least one character in the string.
 |
 |  isalpha(self, /)
 |      Return True if the string is an alphabetic string, False otherwise.
 |
 |      A string is alphabetic if all characters in the string are alphabetic and
there
 |      is at least one character in the string.
 |
 |  isascii(self, /)
 |      Return True if all characters in the string are ASCII, False otherwise.
 |
 |      ASCII characters have code points in the range U+0000-U+007F.
 |      Empty string is ASCII too.
 |
 |  isdecimal(self, /)
 |      Return True if the string is a decimal string, False otherwise.
 |
 |      A string is a decimal string if all characters in the string are decimal
and
 |      there is at least one character in the string.
 |
 |  isdigit(self, /)
 |      Return True if the string is a digit string, False otherwise.
 |
 |      A string is a digit string if all characters in the string are digits and
there
 |      is at least one character in the string.
 |
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier, False otherwise.
 |
 |      Use keyword.iskeyword() to test for reserved identifiers such as "def"
and
 |      "class".
 |
 |  islower(self, /)
 |      Return True if the string is a lowercase string, False otherwise.
 |
 |      A string is lowercase if all cased characters in the string are lowercase
and
 |      there is at least one cased character in the string.
 |
 |  isnumeric(self, /)
 |      Return True if the string is a numeric string, False otherwise.
 |
 |      A string is numeric if all characters in the string are numeric and there
is at
 |      least one character in the string.
 |
 |  isprintable(self, /)
 |      Return True if the string is printable, False otherwise.
 |
 |      A string is printable if all of its characters are considered printable
in
 |      repr() or if it is empty.
 |
 |  isspace(self, /)
 |      Return True if the string is a whitespace string, False otherwise.
 |
 |      A string is whitespace if all characters in the string are whitespace and
there
 |      is at least one character in the string.
 |
 |  istitle(self, /)
 |      Return True if the string is a title-cased string, False otherwise.
 |      
 |      In a title-cased string, upper- and title-case characters may only
 |      follow uncased characters and lowercase characters only cased ones.
 |  
 |  isupper(self, /)
 |      Return True if the string is an uppercase string, False otherwise.
 |      
 |      A string is uppercase if all cased characters in the string are uppercase
and
 |      there is at least one cased character in the string.
 |  
 |  join(self, iterable, /)
 |      Concatenate any number of strings.
 |      
 |      The string whose method is called is inserted in between each given
string.
 |      The result is returned as a new string.
 |      
 |      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
 |  
 |  ljust(self, width, fillchar=' ', /)
 |      Return a left-justified string of length width.
 |      
 |      Padding is done using the specified fill character (default is a space).
 |  
 |  lower(self, /)
 |      Return a copy of the string converted to lowercase.
 |  
 |  lstrip(self, chars=None, /)
 |      Return a copy of the string with leading whitespace removed.
 |      
 |      If chars is given and not None, remove characters in chars instead.
 |  
 |  partition(self, sep, /)
 |      Partition the string into three parts using the given separator.
 |      
 |      This will search for the separator in the string.  If the separator is
found,
 |      returns a 3-tuple containing the part before the separator, the separator
 |      itself, and the part after it.
 |      
 |      If the separator is not found, returns a 3-tuple containing the original
string
 |      and two empty strings.
 |  
 |  replace(self, old, new, count=-1, /)
 |      Return a copy with all occurrences of substring old replaced by new.
 |
 |        count
 |          Maximum number of occurrences to replace.
 |          -1 (the default value) means replace all occurrences.
 |
 |      If the optional argument count is given, only the first count occurrences
are
 |      replaced.
 |
 |  rfind(...)
 |      S.rfind(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Return -1 on failure.
 |
 |  rindex(...)
 |      S.rindex(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notation.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  rjust(self, width, fillchar=' ', /)
 |      Return a right-justified string of length width.
 |
 |      Padding is done using the specified fill character (default is a space).
 |
 |  rpartition(self, sep, /)
 |      Partition the string into three parts using the given separator.
 |
 |      This will search for the separator in the string, starting at the end. If
 |      the separator is found, returns a 3-tuple containing the part before the
 |      separator, the separator itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containing two empty
strings
 |      and the original string.
 |
 |  rsplit(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the delimiter
string.
 |
 |        sep
 |          The delimiter according which to split the string.
 |          None (the default value) means split according to any whitespace,
 |          and discard empty strings from the result.
 |        maxsplit
 |          Maximum number of splits to do.
 |          -1 (the default value) means no limit.
 |
 |      Splits are done starting at the end of the string and working to the
front.
 |
 |  rstrip(self, chars=None, /)
 |      Return a copy of the string with trailing whitespace removed.
 |
 |      If chars is given and not None, remove characters in chars instead.
 |
 |  split(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the delimiter
string.
 |
 |      sep
 |        The delimiter according which to split the string.
 |        None (the default value) means split according to any whitespace,
 |        and discard empty strings from the result.
 |      maxsplit
 |        Maximum number of splits to do.
 |        -1 (the default value) means no limit.
 |
 |  splitlines(self, /, keepends=False)
 |      Return a list of the lines in the string, breaking at line boundaries.
 |
 |      Line breaks are not included in the resulting list unless keepends is
given and
 |      true.
 |
 |  startswith(...)
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      prefix can also be a tuple of strings to try.
 |
 |  strip(self, chars=None, /)
 |      Return a copy of the string with leading and trailing whitespace remove.
 |
 |      If chars is given and not None, remove characters in chars instead.
 |
 |  swapcase(self, /)
 |      Convert uppercase characters to lowercase and lowercase characters to
uppercase.
```

```
 |
 |  title(self, /)
 |      Return a version of the string where each word is titlecased.
 |
 |      More specifically, words start with uppercased characters and all
remaining
 |      cased characters have lower case.
 |
 |  translate(self, table, /)
 |      Replace each character in the string using the given translation table.
 |
 |        table
 |          Translation table, which must be a mapping of Unicode ordinals to
 |          Unicode ordinals, strings, or None.
 |
 |      The table must implement lookup/indexing via __getitem__, for instance a
 |      dictionary or list.  If this operation raises LookupError, the character
is
 |      left untouched.  Characters mapped to None are deleted.
 |
 |  upper(self, /)
 |      Return a copy of the string converted to uppercase.
 |
 |  zfill(self, width, /)
 |      Pad a numeric string with zeros on the left, to fill a field of the given
width.
 |
 |      The string is never truncated.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  maketrans(x, y=None, z=None, /)
 |      Return a translation table usable for str.translate().
 |
 |      If there is only one argument, it must be a dictionary mapping Unicode
 |      ordinals (integers) or characters to Unicode ordinals, strings or None.
 |      Character keys will be then converted to ordinals.
 |      If there are two arguments, they must be strings of equal length, and
 |      in the resulting dictionary, each character in x will be mapped to the
 |      character at the same position in y. If there is a third argument, it
 |      must be a string, whose characters will be mapped to None in the result.
```

So, we can see from the output it gives us a lot of information regarding the datatype including built-in methods we can use to operate on data of that particular type (i.e. Strings).

Note: *If you type the* `help()` *function in your terminal and the output is longer than the window, you can press the letter* `q` *to exit back to normal operation.*

Holy cow that's a lot of words. If we scroll down to the word capitalize, things begin to make more sense. For example, for capitalize, this is what it says:

```
capitalize(...)

    S.capitalize() -> str
    Return a capitalized version of S, i.e. make the first character
    have upper case and the rest lower case.
```

Our next step is to use our formula of datatype-dot-method name-parentheses, and see what happens next.

```
"smithers".capitalize()
```

```
'Smithers'
```

Tips going forward

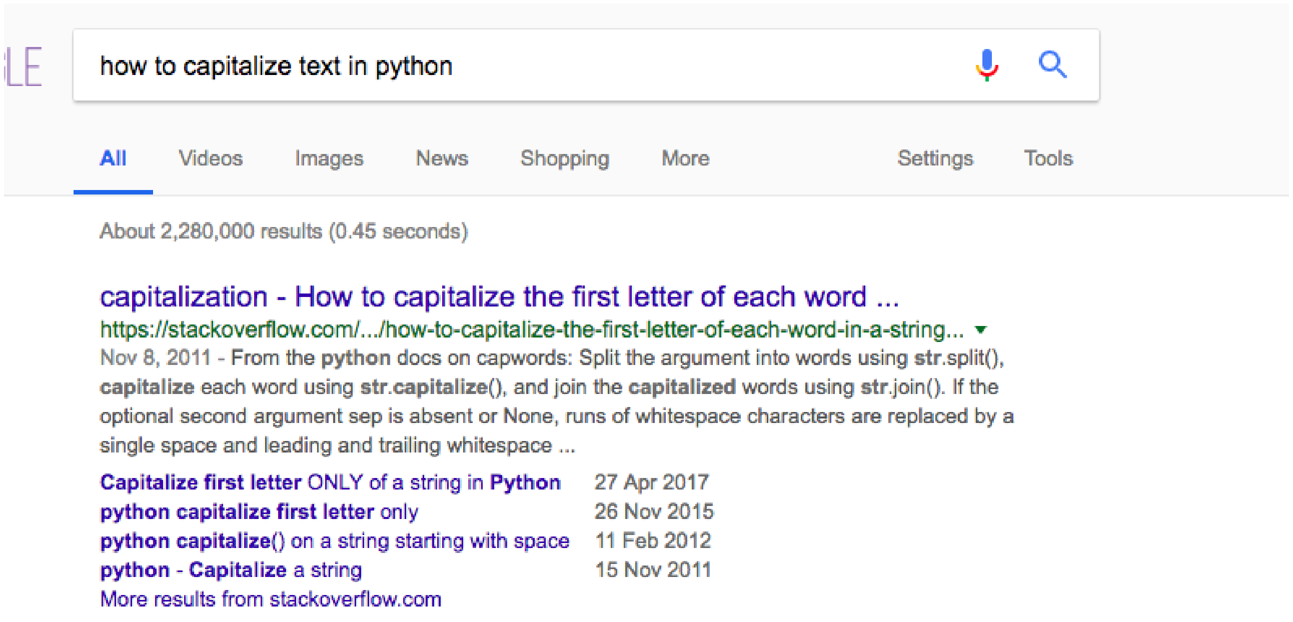
There are many other examples of methods to use with strings in Python. In fact, they are all listed when you called `help(str)` above. What is important when learning how to code in Python is to think of the right question to ask when you are stuck and learn how to read the documentation. There is a cycle to getting comfortable with Python functions and methods.

In this lesson, we went through that cycle:

- Guess: We just tried something and looked to the error message for clues as to what to do next.
- help(str): We saw a nice way to learn about new methods, then we took a guess to test our understanding
- Following a pattern: We started with a simple method like calling upper, took a moment to break this down into a pattern, and then tried this pattern again to call

other methods

Here is one more method of discovery: **just ask Google**. For example, look what happens when we ask Google about capitalization.



how to capitalize text in python

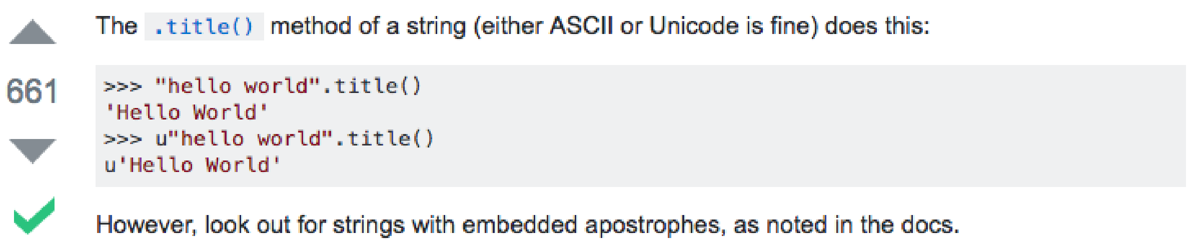
About 2,280,000 results (0.45 seconds)

capitalization - How to capitalize the first letter of each word ...
https://stackoverflow.com/.../how-to-capitalize-the-first-letter-of-each-word-in-a-string...
Nov 8, 2011 - From the **python** docs on capwords: Split the argument into words using **str**.split(), **capitalize** each word using **str.capitalize**(), and join the **capitalized** words using **str**.join(). If the optional second argument sep is absent or None, runs of whitespace characters are replaced by a single space and leading and trailing whitespace ...

Capitalize first letter ONLY of a string in **Python**	27 Apr 2017
python capitalize first letter only	26 Nov 2015
python capitalize() on a string starting with space	11 Feb 2012
python - **Capitalize** a string	15 Nov 2011

More results from stackoverflow.com

A great link with a detailed answer.



The `.title()` method of a string (either ASCII or Unicode is fine) does this:

```
>>> "hello world".title()
'Hello World'
>>> u"hello world".title()
u'Hello World'
```

However, look out for strings with embedded apostrophes, as noted in the docs.

Then we try this new method out ourselves, to see if this user on StackOverFlow is right (they normally are).

```
"hello world".title()
```

```
'Hello World'
```

And our work is done.

Feel free to look at other common string operations here.

Summary

In this lesson, we learned about our first datatype in Python: the string. A string is just text. We indicate to Python that we are writing a string by surrounding our content with quotation marks. Once we do this, we can operate on this string by calling methods like `upper()` or `endswith()` . We identified a general pattern for calling methods on datatypes: 'instance of a datatype-dot-method name-parentheses'.

The second thing we learned was different mechanisms for learning about methods. We saw the importance of guessing and experimentation, and how doing so can give us error messages, which provide clues. We also saw how to ask questions about a datatype by calling `help()` followed by the datatype name like `help(str)` . Finally, we saw we can ask Google. This mechanism of exploration is a skill we'll build up over time and this course will provide guidance and practice all along the way.