Python Programming
Strings

Strings

- String basics.
- Special characters.
- Multi-line strings.
- Indexing and slicing strings.
- Common string operators and methods.
- Formatting strings.
- Built-in string functions.

Introduction

- According to the Python documentation, “Strings are immutable sequences of Unicode code points.” Less technically speaking, strings are sequences of characters. The term sequence in Python refers to an ordered set. Other common sequence types are lists, tuples, and ranges, all of will be covered.
- https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- Note that there is no “character” type in Python. A single character is just a string of length 1. So, when you index a string, you get multiple one-character strings (or strings of length 1).
-

Quotation Marks and Special Characters (Escaping)

- Strings can be created with single quotes or double quotes. There is no difference between the two.
- To create a string that contains a single quote, enclose the string in double quotes:

```
phrase = "Where'd you get the coconuts?"
```

- Vice versa for double quotes:

```
phrase = 'The soldier asked, "Are you suggesting coconuts migrate?"'
```

- Sometimes you will want to output single(double) quotes within a string denoted by single(double) quotes. Then use escaping with a backslash (\):

```
>>> phrase = "The soldier said, \"You've got two empty halves of a
coconut.\""
>>> print(phrase)
The soldier said, "You've got two empty halves of a coconut."
```

Special Characters

- backslash (\)

```
>>> phrase = "Use an extra backslash to output a backslash: \\"
>>> print(phrase)
Use an extra backslash to output a backslash: \
```

- Two other common special characters are the newline (\n) and horizontal tab (\t):

```
>>> print('Equation\tSolution\n 55 x 11\t 605\n 132 / 6\t 22')
Equations   Solution
55 x 11      605
132 / 6      22
```

Escape sequences

Escape sequence	Meaning
\'	Single Quote
\"	Double Quote
\\	Backslash
\n	Newline
\t	Horizontal tab

Raw Strings

- Sometimes, a string might have a lot of backslashes in it that are just meant to be plain old backslashes. The most common example is a file path. For example:

```
'C:\news\today.txt'
```

- If you assign this to a variable, the \n and \t characters get printed as a newline and a tab

- Using the “r” (for raw data) prefix on the string, we ensure that all backslashes are escaped

```
>>> my_path = r'C:\news\today.txt'
>>> print(my_path)
C:\news\today.txt
```

- Note that you cannot end a raw string with a single backslash:

```
my_path = r'C:\my\new\'
```

Triple quotes

- Triple quotes are used to create multi-line strings. You generally use three double quotes
- Demo 16: strings/Demos/triple_quotes.py
- Three single quotes will work as well, but double quotes are recommended. More information at https://www.python.org/dev/peps/pep-0008/#string-quotes.
- Note that quotation marks can be included within triple-quoted strings without being escaped with a backslash
- break up your code with a newline without having that newline show up in your output. Escape the actual newline with a backslash:
- Demo 17: strings/Demos/triple_quotes_newline_escaped.py

String Indexing

- Indexing is the process of finding a specific element within a sequence of elements through the element's position. Remember that strings are basically sequences of characters. We can use indexing to find a specific character within the string.
- the first character (the left-most) is at position 0.
- the first character (the right-most) is at position -1.

M	O	N	T	Y		P	Y	T	H	O	N
0	1	2	3	4	5	6	7	8	9	10	11
-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

- Demo 18: strings/Demos/string_indexing.py

Exercise 11

- Indexing Strings
  - 10-20 min

Slicing Strings

- Often, you will want to get a sequence of characters from a string (i.e., a substring). In Python, you do this by getting a slice of the string:

```
substring = orig_string[first_pos:last_pos]
```

- last_pos is excluded
- 'hello'[1:3] would return "el"
- 'hello'[:3] would return "hel"
- 'hello'[3:] would return "lo"
- Demo 19: strings/Demos/string_slicing.p

Exercise 12

- Slicing Strings
  - 10-20 min

Concatenation and repetition

- Concatenation is a means for stringing strings together
- it is often used to combine variables with literals
- Demo 20: strings/Demos/concatenation.py
- Repetition is the process of repeating a string some number of times.
- repetition is done with the * operator.
- Demo 21: strings/Demos/repetition.py

Exercise 13

- Repetition
  - 5-10 min

Combining Concatenation and Repetition

- Concatenation and repetition can be combined. Repetition takes precedence

```
>>> 'a' + 'b' * 3 + 'c'
'abbbc'
```

- force the concatenation to take place first by using parentheses:

```
>>> ('a' + 'b') * 3 + 'c'
'abababc'
```

- Demo 22: strings/Demos/combining_concatenation_and_repetition.py

Python Strings are Immutable

- use Python’s built-in id() function, to return the identity of an object:

```
>>> name = 'Nat'
>>> id(name)
1670060967728
>>> name += 'haniel'
>>> name 'Nathaniel'
>>> id(name)
1670060968240
```

- Notice that the id of name changes when we modify the string in the variable

Python Strings are Immutable

- the original string is unchanged. for example, the upper() method returns a string in all uppercase letters:

```
>>> name = 'Nat'
>>> name.upper() # Returns uppercase copy of 'Nat'
'NAT'
>>> name # Original variable is unchanged
'Nat'
```

- If you want to change the original variable, you must assign the returned value back to the variable:

```
>>> name = 'Nat'
>>> name = name.upper()
>>> name
'NAT'
```

Common String Methods

- str.capitalize() – Returns a string with only the first letter capitalized.
- str.lower() – Returns an all lowercase str
- str.upper() – Returns an all uppercase string.
- str.title() – Returns a string with each word beginning with a capital letter followed by all lowercase letters.
- str.swapcase() – Returns a string with the case of each letter swapped.

```
>>> 'hELLo'.capitalize()
'Hello'
>>> 'hELLo'.lower()
'hello'
>>> 'hELLo'.upper()
'HELLO'
>>> 'hello world'.title()
'Hello World'
>>> 'hELLo'.swapcase()
'HellO'
```

String replace

- A string with old replaced by new count times.
- ```
  >>> 'mommy'.replace('m', 'b')
  'bobby'
  >>> 'mommy'.replace('m', 'b', 2)
  'bobmy'
  ```
- str.replace(old, new[, count])
- count is optional, so you can nest optional parameters within optional parameters.

```
str.find(sub[, start[, end]])
```
```

Strip characters

- str.strip([chars]) – Returns a string with leading and trailing chars removed.
- str.lstrip([chars]) – Returns a string with leading chars removed.
- str.rstrip([chars]) – Returns a string with trailing chars removed.
- chars defaults to whitespace.

```
>>> ' hello '.strip()
'hello'
>>> 'hello'.lstrip('h')
'ello'
>>> 'hello'.rstrip('o')
'hell
```

String methods that return Boolean

- str.isalnum() – Returns True if all characters are letters or numbers.
- str.isalpha() – Returns True if all characters are letters.
- str.islower() – Returns True if string is all lowercase.
- str.isupper() – Returns True if string is all uppercase.
- str.istitle() – Returns True if string is title case.

```
>>> 'Hello World!'.isalnum()
False
>>> 'Hello World!'.isalpha()
False
>>> 'hello'.islower()
True
>>> 'HELLO'.isupper()
True
>>> 'Hello World!'.istitle()
True
```

String methods that return Boolean

- str.isspace()
- True if string is made up of only whitespace.

```
>>> ' '.isspace()
True
>>> ' hi '.isspace()
False
```

- The str.isdigit(), str.isdecimal(), and str.isnumeric() all check to see if a string has only numeric character
- All three will return True if the string contains only Arabic digits (i.e., 0 through 9)

```
'42'.isdigit() # True
'42'.isdecimal() # True
'42'.isnumeric() # T
```

String methods that return Boolean

- All three will return False if any character is non-numeric:

```
'4.2'.isdigit() # False
'4.2'.isdecimal() # False
'4.2'.isnumeric() # False
```

- '23'.isnumeric() and '23'.isdigit() return True, but '23'.isdecimal() returns False.
- '¼'.isnumeric() returns True, but '¼'.isdigit() and '¼'.isdecimal() return
- All the is... methods shown above return False for empty strings, meaning strings with zero length.

String methods

- str.startswith() and str.endswith()
  - str.startswith(prefix[, start[, end]]) – Returns True if string starts with prefix.
  - str.endswith(prefix[, start[, end]]) – Returns T
- Both of these methods start looking at start index and end looking at end index if start and end are specified.

```
>>> 'hello'.startswith('h')
True
>>> 'hello'.endswith('o')
True
```

String Methods that Return a Number

String Methods that Return a Position (Index) of a Substring

- str.find(sub[, start[, end]]) – Returns the lowest index where sub is found. Returns -1 if sub isn’t found.
- str.rfind(sub[, start[, end]]) – Returns the highest index where sub is found. Returns -1 if sub isn’t found.
- str.index(sub[, start[, end]]) – Same as find(), but errors when sub is not found.
- str.rindex(sub[, start[, end]]) – Same as rfind(), but errors when sub is not found.

```
>>> 'Hello World!'.find('l')
2
>>> 'Hello World!'.rfind('l')
9
>>> 'Hello World!'.index('l')
2
>>> 'Hello World!'.rindex('l')
9
```

String methods that return a number

- str.count(sub[, start[, end]])
- Returns the number of times sub is found. Start looking at start index and end looking at end index.

```
>>>"Hello World!".count('l')
3
```

String formatting

- Python includes powerful options for formatting strings.
- The format() Method

```
>>> '{0} is an {1} movie!'.format('Monty Python','awesome')
'Monty Python is an awesome movie!'

>>> '{movie} is an {adjective} movie!'.format(movie='Monty Python', …
adjective='awesome')
'Monty Python is an awesome movie!'

'{} is an {} movie!'.format('Monty Python', 'awesome')
```

- These examples really just show another form of concatenation and could be rewritten like this:

```
'Monty Python' + ' is an ' + awesome + ' movie!'
# or
movie = 'Monty Python'
adjective = 'awesome'
movie + ' is an ' + adjective + ' movie!'
```

String formatting

- format() can also be used to format the replacement strings with specification:

```
{field_name:format_spec}
#or
{:format_spec}
```

- format_spec is:

```
[[fill]align][sign][width][,][.precision][type]
```

- let’s break it down from right to left:

Type

`[[fill]align][sign][width][,][.precision][type]`

- type is a 1 letter specifier e.g.:

```
'{:s} is an {:s} movie!'.format('Monty Python','awesome')
```

- s is a string, d is a decimal integer, f is a float
- there are many more types (https://docs.python.org/3/library/string.html#formatspec)
- Demo 23: strings/Demos/formatting_types.py

Precision

[[fill]align][sign][width][,][.precision][type]

- The precision is specified before the type and is preceded by a decimal point, like this:

```
>>> import math
>>> 'pi equals {:.2f}'.format(math.pi)
'pi equals 3.14'
```

Separating Thousands

```
[[fill]align][sign][width][,][.precision][type]
```

- Insert a comma before the precision value to separate the thousands with commas, like this:

```
>>> '{:,.2f}'.format(1000000)
'1,000,000.00'
```

Width

[[fill]align][sign][width][,][.precision][type]

- The width is an integer indicating the minimum number of characters of the resulting string. If the passed-in string has fewer characters than the specified width, padding will be added. By default, padding is added after strings and before numbers

```
>>> '{:5}'.format('abc')
'abc  '
>>> '{:5}'.format(123)
'  123'
>>> '{:5.2f}'.format(123)
'123.00'
```

Sign

[[fill]align][sign][width][,][.precision][type]

- By default, negative numbers are preceded by a negative sign, but positive numbers are not preceded by a positive sign. To force the sign to show up, add a plus sign (+) before the precision, like this:

```
>>> 'pi equals {:+.2f}'.format(math.pi)
'pi equals +3.14'
```

Alignment

[[fill]align][sign][width][,][.precision][type]

- **<** left align **>** right align **=** padding (between sign/digits) **^** centered

```
>>> '{:>5}'.format('abc')
'  abc'
>>> '{:<5}'.format(123)
'123  '
>>> '{:^5}'.format(123)
' 123 '
>>> '{:=+5}'.format(123)
'+  123'
```

Fill

[[fill]align][sign][width][,][.precision][type]

- By default, spaces are used for padding, but this can be changed by inserting a fill character before the alignment option, like this (note the period after the colon):

```
>>> '{:.^10.2f}'.format(math.pi)
'...3.14...'
```

- And now with a dash:

```
>>> '{:-^10.2f}'.format(math.pi)
'---3.14---'
```

Percentage type

- As mentioned earlier, another option for type is percentage (%):

```
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> grade
0.72
>>> '{:.2f}'.format(grade)
'0.72'
>>> '{:.2%}'.format(grade)
'72.00%'
>>> '{:.0%}'.format(grade)
'72%'
```

Long lines of code

- The Python Style Guide23 suggests that lines of code should be limited to 79 characters. This can be difficult as each line of code is considered a new statement; however, it can usually be accomplished through some combination of:

1. Breaking method arguments across multiple lines.

2. Concatenation.

3. Triple-quoted multi-line string

- Demo 24: strings/Demos/long_code_lines.py

Exercise 14

- Playing with formatting
  - 10-20 min

Formatted String Literals (f-strings)

```
user_name = input("What is your name? "
greeting = f"Hello, {user_name}!"
```

- The curly braces within the f-string contain the variable name and optionally a format specification. The string literal is prepended with an f.
- Demo 25: strings/Demos/f_strings.py

Practice

- Everything you learned earlier about formatting can be applied to the f-string because the same formatting function is called. Practice with f-strings by running the following lines of code:

```
>>> import math
>>> movie = 'Monty Python'
>>> adjective = 'awesome'
>>> f'{movie} is an {adjective} movie!'
'Monty Python is an awesome movie!'
>>> low = 1
>>> high = 5
>>> rating = 6
>>> f'On a scale of {low} to {high},
{movie} is a {rating}.'
'On a scale of 1 to 5, Monty Python is a
6.'
```

```
>>> f'pi equals {math.pi:f}'
'pi equals 3.141593'
>>> f'pi equals {math.pi}'
'pi equals 3.141592653589793'
>>> f'pi equals {math.pi:.2f}'
'pi equals 3.14'
>>> f'{1000000:,.2f}'
'1,000,000.00'
>>> f"{'abc':20}"
'abc                  '
>>> f'{123:20}'
'                   123'
>>> f'{123:5.2f}'
'123.00'
```

(continue on next page)

Practice (continued)

```
>>> f'pi equals {math.pi:+.2f}'
'pi equals +3.14'
>>> f"{'abc':>20}"
'                 abc'
>>> f'{123:<20}'
'123                  '
>>> f'{123:=+20}'
'+                  123'
>>> f'pi equals {math.pi:.^10.2f}'
'pi equals ...3.14...'
>>> f'pi equals {math.pi:-^10.2f}'
'pi equals ---3.14---'
>>>
```

```
>>> questions = 25
>>> correct = 18
>>> grade = correct / questions
>>> f'{grade:.2f}'
'0.72'
>>> f'{grade:.2%}'
'72.00%'
>>> f'{grade:.2f}'
'0.72'
```

Built-in string functions

- str(object)
- convert to string datatype

```
>>> str('foo') # 'foo' is already a string
'foo'
>>> str(999)
'999'
>>> import math
>>> str(math.pi)
'3.141592653589793'
```

- len(string)

```
>>> len('foo')
3
```

- len() can also take other objects (which will be covered later)

Built-in string functions

- min() and max()

```
>>> min('w', 'e', 'b')
'b'
>>> min('a', 'B', 'c')
'B'
>>> max('w', 'e', 'b')
'w'
>>> max('a', 'B', 'c')
'c'
```

- min() and max() can also take iterables (which will be covered later)

Exercise 15

- outputting tab-delimited text
  - 25-40 min