Lecture 1.2 : Strings

Introduction

- A string is simply a *sequence* of characters. The string type is a *collection type* meaning it contains a number of objects (characters in this case) that can be treated as a single object. The string type is a particular type of collection type called a *sequence type*. This means each constituent object (here character) in the collection occupies a specific numbered location within it i.e. elements are *ordered*.
- Python strings are typically enclosed in single or double quotes. (Pick a style and be consistent throughout and across your programs.)

```python
name1 = "Jimmy Murphy"
name2 = 'Mary Kelly'
print(name1)
print(name2)
```

```
Jimmy Murphy
Mary Kelly
```

- What if you want to include quotes in your string? You have a couple of options: You can enclose your string in the other kind of quote or you can *escape* the quote with a backslash (nullifying its role as a string delimiter).

```python
name3 = "Nora O'Neill"
name4 = 'Sally O\'Brien'
print(name3)
print(name4)
```

```
Nora O'Neill
Sally O'Brien
```

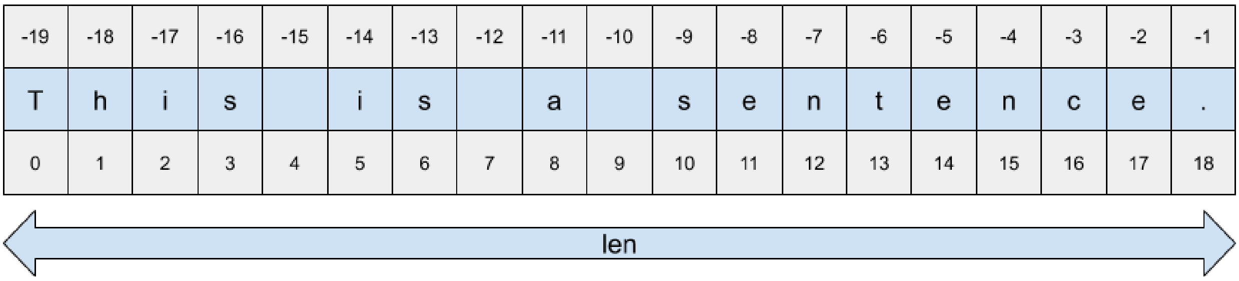
- Python strings can contain *non-printing* characters (such as a `\n` which causes a new line to be emitted when printing the string).

```python
rhyme = "Humpty Dumpty sat on a wall,\nHumpty Dumpty had a great fall."
print(rhyme)
```

```
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
```

String indexing

- As mentioned, a string is a *sequence* type. This means each member object (character in this case) occupies a numbered position in the collection and can be accessed by *indexing* the sequence at that index.
- For example, the characters of the string `'This is a sentence.'` reside at the indices indicated here:



-19	-18	-17	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1
T	h	i	s		i	s		a		s	e	n	t	e	n	c	e	.
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

- The length of a string is the number of characters it contains. The `len()` function returns the length of a string.
- We can extract individual characters by *indexing* the string at a given location. The first character in the string is located at index zero. If the length of the string is N, the final character is located at index N-1. Indexing outside string boundaries gives rise to an error.

```python
s = 'This is a sentence.'
print(len(s))
print(s[0])
print(s[1])
print(s[2])
print(s[18])
print(s[19])
```

```
19
T
h
i
.



---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [4], in <module>
      5 print(s[2])
      6 print(s[18])
----> 7 print(s[19])

IndexError: string index out of range
```

- In Python it is possible to index relative to the end of the string using negative indices: The last character is at index -1, the second last at index -2, the third last at index -3, etc.

```python
print(s[-1])
print(s[-2])
print(s[-3])
print(s[-19])
print(s[-20])
```

```
.
e
c
T
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Input In [5], in <module>
      3 print(s[-3])
      4 print(s[-19])
----> 5 print(s[-20])

IndexError: string index out of range
```

String slicing

- We can extract more than a single character from a string. We can extract subsequences or *slices* by specifying a range of indices separated by a colon. Writing `s[start:end]` will return a new string composed of the characters `s[start]`, `s[start+1]`, `s[start+2]`, `...`, `s[end-3]`, `s[end-2]`, `s[end-1]`.
- Note that the character located at `s[end]` is *not* returned.

```
s = 'This is a sentence.'
print(s[0:4])
print(s[5:7])
```

```
This
is
```

- If either the starting or ending indices are omitted their values default to the beginning and end of the string.

```
s = 'This is a sentence.'
print(s[:4])
print(s[10:])
print(s[:])
```

```
This
sentence.
This is a sentence.
```

- As usual, negative indices can be used to specify locations relative to the end of the string.

```
s = 'This is a sentence.'
print(s[:-10])
```

```
This is a
```

- Slicing is more forgiving in terms of indices than pure indexing. A start or end index outside of a string's boundaries is treated as the string boundary.

```
s = 'This is a sentence.'
print(s[:100])
print(s[-100:])
print(s[-100:100])
```

```
This is a sentence.
This is a sentence.
This is a sentence.
```

Extended slicing

- It is possible to specify a third parameter when slicing sequences. It indicates the *step size* to take along the sequence when extracting its elements. Writing `s[start:end:step]` will return a new string composed of `s[start]`, `s[start+step]`, `s[start+2*step]`, `s[start+3*step]`, etc. Extraction continues for as long as `start+i*step < end` where i = 0, 1, 2, 3, etc.
- As usual, if the start or end of the slice are omitted they default to the beginning and end of the string respectively.

```
s = 'This is a sentence.'
print(s[::1])
print(s[::2])
print(s[:10:3])
print(s[10::3])
```

```
This is a sentence.
Ti sasnec.
Tss
stc
```

- A negative step size is interpreted as a step backwards through the sequence. This is handy for reversing a string.
- For a negative step size, if the start or end of the slice are omitted, their values default to the end and beginning of the string respectively.

```python
s = 'This is a sentence.'
print(s[-1:-20:-1])
print(s[::-1])
print(s[::-2])
print(s[-18:-10:-1])
print(s[-10:-18:-1])
```

```
.ecnetnes a si sihT
.ecnetnes a si sihT
.censas iT

 a si si
```

String concatenation and replication

- We can use the `+` operator to concatenate strings.

```python
line1 = 'Humpty Dumpty sat on a wall'
line2 = 'Humpty Dumpty had a great fall'
print(line1 + ',\n' + line2 + '.')
```

```
Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall.
```

- We can use the `*` operator to replicate strings.

```python
s = 'apple '
print(s * 3)
```

```
apple apple apple
```

String comparison

- Strings can be tested for equality with the `==` operator.

```python
print('cat' == 'dog')
print('mouse' == 'mouse')
```

```
False
True
```

Strings are `True` or `False`

- The empty string `''` is interpreted as `False`.

```
s = ''
if s:
   print(True)
else:
   print(False)
```

```
False
```

- Any non-empty string is interpreted as `True`.

```
s = 'apple'
if s:
   print(True)
else:
   print(False)
```

```
True
```

Strings are iterable

- Because a string is an *iterable* sequence we can use a `for` loop to examine each of its characters in turn.

```
s = 'apple'
for c in s:
   print(c)
```

```
a
p
p
l
e
```

Strings are immutable

- Strings are *immutable*. This means they cannot be modified. If we try to modify a string we get an error.

```
s = 'apple'
s[0] = 'A'
print(s)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [18], in <module>
      1 s = 'apple'
----> 2 s[0] = 'A'
      3 print(s)

TypeError: 'str' object does not support item assignment
```

- If we want a “modify” a string we have to create a new one from the original.

```
s = 'apple'
s = 'A' + s[1:]
print(s)
```

```
Apple
```

Processing lines of text

- A common operation is to read in a line of text from a file, strip any surrounding whitespace and split the line into its constituent tokens.

```
line = 'This is a line of text'
tokens = line.strip().split()
print(tokens)
```

```
['This', 'is', 'a', 'line', 'of', 'text']
```

- By default, the `split()` method splits strings on whitespace but we can ask it to split on other characters.

```
line = 'This,is,a,line,of,text'
tokens = line.strip().split(',')
print(tokens)
```

```
['This', 'is', 'a', 'line', 'of', 'text']
```

- We can use the `join()` method to glue lists of words back together into a single string. We need to tell `join()` which glue character to use.

```python
line = 'This,is,a,line,of,text'
tokens = line.strip().split(',')
print(tokens)
newline = ' '.join(tokens)
print(newline)
newline = '-'.join(tokens)
print(newline)
```

```
['This', 'is', 'a', 'line', 'of', 'text']
This is a line of text
This-is-a-line-of-text
```

String membership

- We can use the `in` operator to check whether one string is a substring of another.

```python
s = 'This is a sentence.'
print('This' in s)
print('x' in s)
print('.' in s)
```

```
True
False
True
```

String methods

- Python comes with built-in support for a large set of common string operations. These operations are called `methods` and they define the things we can do with strings. We have looked only at a small number of Python's string methods.
- Calling `help(str)` in the Python shell or `pydoc str` in a Linux terminal outputs a list of these methods. We see the methods we can invoke on a string `s` include `capitalize()` (returns a capitalized version of `s`), `isdecimal()` (returns `True` if `s` contains only decimal characters), `lower()` (returns a new copy of `s` with all characters converted to lowercase), etc.
- Many editors and IDEs support autocompletion, whereby, if `s` is a string, placing your cursor after the dot in `s.` and hitting tab will provide you with a list of the string methods that can be invoked on `s`. Then simply select the one you want.
- Note that, because strings are immutable, calling a method on a string will *not* alter the string itself.

```python
s = 'apple'
print(s.capitalize())
print(s)
```

```
Apple
apple
```

- Whenever you find it necessary to carry out some string processing, first look up the list of built-in string methods. There may be one that will help you with your task. There is no point writing code that duplicates what a built-in string method can do for you already.

```
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 ar
 |      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 ther
 |      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 ther
 |      is at least one character in the string.
 |  
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier, False otherwise.
 |      
 |      Call keyword.iskeyword(s) to test whether string s is a reserved identifier,
 |      such as "def" or "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 a
 |      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 ther
 |      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 strin
 |      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 ar
 |      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 removed.
 |
 |      If chars is given and not None, remove characters in chars instead.
 |
 |  swapcase(self, /)
 |      Convert uppercase characters to lowercase and lowercase characters to uppercas
 |
 |  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 widt
 |
 |      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(...)
 |      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.
```