Lecture 4 - Python 3 Strings

Computer Programming

Robert D. Vincent and Samia Hilal

Marianopolis College

February 9, 2022

String literals

- Recall: String literal constants are just text surrounded by either single, double, or "triple" quotes:

```
'This is "single-quoted" string'
"Here's a double-quoted string"
"""This is a triple-quoted string"""
```

- Special characters can be introduced with a backslash:

```
"A newline after the message\n"
'F\tC'  # tab character between F and C
"C:\\Windows\\Path\\Name"
```

String characteristics

- Strings represent text of any length.
- The Python string type is called `str`.
- Strings can be thought of as a sequence of characters.
- A string has a known, constant length.
- Strings can be used with both operators and functions.
- A string value is *immutable* - it cannot be changed.

String operators

- We can *concatenate* strings with the + operator:

```
>>> x = "meer"
>>> y = "kat"
>>> x+y
'meerkat'
```

- We can repeat strings with the * operator:

```
>>> 'Fa' * 4
'FaFaFaFa'
>>> 10 * "-" # Works in either order.
'----------'
```

- These operators are *overloaded* - they do different things for operands of different types.

String comparison operators

- We can compare strings to one another.
- Python considers 'A' different from 'a':

```
>>> x = "Fred"
>>> x == "Alice"
False
>>> x == "Fred"
True
>>> x == "fred"
False
>>> x != "fred"
True
```

String comparison operators

- We can also use the >, <, <=, and >= operators.
- These **almost** perform alphabetic ordering:

```
>>> "aardvark" < "zebra"
True
>>> "aardvark" < "armadillo"
True
>>> "Alice" > "Bob"
False
>>> "alice" > "Bob"
True                # What happened here?
```

String comparisons

- Sometimes called *lexicographic* ordering.
- Compares each element (character) in turn.
- Remember from Lecture 2:
  - Upper-case 'A' = 65 decimal
  - Lower-case 'a' = 97 decimal
- Therefore 'a' > 'A' !!
- To ignore case, convert to all upper or all lower case before comparison.

String length

- The built-in `len()` function computes the number of characters in a string:

```
>>> text = "apple"
>>> len(text)
5
>>> len('')
0
>>> len("A\tB\n") # A, tab, B, newline
4
```

- We *call* a function by typing its name, followed by a list of *arguments* in parentheses.
- The function call will return a value and/or take an action.

String indexing

- An *index* expression extracts a single character:

```
>>> text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> text[0]    # First character
'A'
>>> text[1]    # Second character
'B'
>>> text[25]   # Final character
'Z'
>>> text[26]
IndexError:  string index out of range
```

- The index must be present in the string.
- Index values range from 0 to `len(text) - 1`

String indexing, continued

- Negative offsets are interpreted relative to the end of the string:

```
>>> text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> text[-1]   # Final character
'Z'
>>> text[-2]   # Second-to-last
'Y'
>>> text[-26] # First character
'A'
>>> text[-27]
IndexError:  string index out of range
```

- Again, the element must be present.

String indexing, continued

- Indexing is very important!
- Used when we have a single name or value that contains multiple elements in a fixed order.
- The index starts at zero, so the first element is at position zero.
- With strings, the result is a string of length one.

```
>>> text = "abcde"
>>> len(text)
5
>>> n = 1
>>> c = text[n+1] # Extract at index 2
>>> print(c, len(c))
c 1
```

String slicing

- A slice expression may extract zero or more characters.
- A colon separates parts of the slice expression.

```
>>> text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> text[1:4] # 4 not included
'BCD'
>>> text[:7]  # Zero assumed!
'ABCDEFG'
>>> text[9:]  # From 9 to end.
'JKLMNOPQRSTUVWXYZ'
>>> text[30:] # Nonexistent, but ok
''
```

- Slice expressions **may** request start or end positions that do not exist.

String slicing with different increments

You can specify a third argument to a slice expression that tells Python how many positions to skip:

```
>>> text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> text[0:4]      # Default increment is 1
'ABCD'
>>> text[0:4:2]    # Every second character
'AC'
>>> text[::3]      # Every third character
'ADGJMPSVY'
>>> text[1:100:4] # OK if 2nd too big
'BFJNRVZ'
>>> text[100:]     # OK if 1st too big
''
```

Slicing with negative offsets

Negative offsets are computed relative to the **end** of the string.

```
>>> text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> text[-2:]    # From Y until end.
'YZ'
>>> text[-10:]   # From Q until end.
'QRSTUVWXYZ'
>>> text[:-20]   # From A until F.
'ABCDEF'
>>> text[1:-20] # From B until F.
'BCDEF'
```

If start or end is negative, a positive increment is still used.

Slicing with negative increment

A negative increment reverses the direction of the slice:

```
>>> text = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
>>> text[::-1]    # Reverse the entire string
'ZYXWVUTSRQPONMLKJIHGFEDCBA'
>>> text[-10::-1]  # From Q until beginning.
'QPONMLKJIHGFEDCBA'
>>> text[-10::-2]  # Skip every other.
'QOMKIGECA'
```

The default start and end values change if the increment is negative.

Review of string slicing and indexing

- An index expression:
  - `string[ index ]`
  - Uses one integer expression in square brackets.
  - returns a *single-character* string.
- A slice expression:
  - `string[ start:end:increment ]`
  - Uses a colon to separate the start, end, and increment.
  - returns a string of *zero or more* characters.
  - Default values are used for missing components.
- In both cases, a negative `start` or `end` value counts from the end of the string.

The `in` operator

- The `in` operator searches for a string within a string:

```
>>> text = "Earth"
>>> "art" in text
True
>>> text in "Earth, Wind, and Fire"
True
>>> "Art" in text
False
>>>
```

- Later we'll see that `in` has other uses.

Character type tests

Check the kinds of characters in a string:

```
>>> text1 = "0019"
>>> text2 = "Canada"
>>> text1.isalpha()
False
>>> text1.isdigit()
True
>>> text2.isalpha()
True
>>> text2.isdigit()
False
>>> text2.isalnum()
True
```

Character type tests, continued

- `isalpha()` - Returns `True` if all of the characters in the string are alphabetic.
- `isdigit()` - Returns `True` if all of the characters in the string are digits (0-9).
- `isalnum()` - Returns `True` if all of the characters are either digit or alphabetic.
- `isupper()` - Returns `True` if all of the characters are upper-case alphabetic.
- `islower()` - Returns `True` if all of the characters are lower-case alphabetic.
- All return `False` for an empty string!

String Searching - index

Use `index` to check the for the position of a substring that occurs in a string.
`index` raises an error if substring is not found:

```
>>> text = "ABCDEFGH"
>>> text.index("A")  # Find first 'A'
0
>>> text.index('CD') # Find substring
2
>>> text.index('DEF') # Longer substring
3
>>> text.index('CDG')
ValueError:  substring not found
```

String Searching - find

Using `find` is very similar to `index`.
Use `find` when you do not know if substring occurs.
returns -1 if substring not found.

```
>>> text = "ABCDEFGH"
>>> text.find("A")  # Find first 'A'
0
>>> text.find('CD') # Find substring
2
>>> text.find('DEF') # Longer substring
3
>>> text.find('CDG')
-1
```

More string searching

```
>>> text = 'Hello, World!'
>>> text.endswith('d!')
True
>>> text.endswith('z!')
False
>>> text.index("o") # Find first 'o'
4
>>> text.rindex("o") # Find last 'o'
8
>>> text.rfind("o") # Find last 'o' or -1
8
```

More about function calls

- There are two kinds of function calls in Python:
  - Normal functions like `len()`, `print()`, `input()`:

    `function(argument)`
  - *Method* functions like many string operations:

    `argument.function()`
- A method function cannot be called like a normal function.
- In method functions, the first argument to the function is the value appearing before the period.
- More details when we cover *object-oriented* programming.

Strings are immutable

Other string methods change the case of strings:

```
>>> text.upper()
'APPLE'
>>> text.lower()
'apple'
>>> text.capitalize()
'Apple'
>>> text
'apple'     # Original is unchanged!
```

Python strings are *immutable* - we never change them, but we can make a copy with new properties.

```
>>> new_text = text.upper()
>>> new_text
'APPLE'
```

Strings are immutable

We can't change individual values in strings:

```
>>> text = 'Apple'
>>> x = text[0]
>>> print(x)
'A'
>>> text[0] = 'B'
# ERROR!
```

Python strings are *immutable* - we never change them, we just make altered copies!

```
>>> text = 'ABBBB'
>>> new_text = 'B' + text[1:]
>>> new_text
'BBBBB'
```

Replacing text

We can replace or remove text in a copy of a string.
These methods return an updated copy of the string.
Original string remains unchanged:

```
>>> text = "in all thy sons command"
>>> text.replace("thy sons", "of us")
'in all of us command'
>>> text.replace("o", "-")
'in all thy s-ns c-mmand'
>>> text.replace("s", '')
'in all thy on command'
# Remove leading and trailing spaces
>>> text2 = " abcd e "
>>> text2.strip()
'abcd e'
```

Converting other types to `str`

- The `str()` function will convert other types to `str`.

```
>>> x = 100
>>> y = str(x + 1)
>>> y
'101'
>>> y + 2
TypeError:  can't convert 'int' object to str...
>>> y + '2'
'1012'
>>> 'xyz' + y
'xyz101'
```

Summary

- A string is an *immutable* sequence of characters.
- Some operators are *overloaded* to work with strings.
- The `len()` function gives the number of characters in a string.
- We can extract a single character using an *index* expression.
- We can extract a series of characters using a *slice* expression.