CS303E: Elements of Computers and Programming

More on Strings

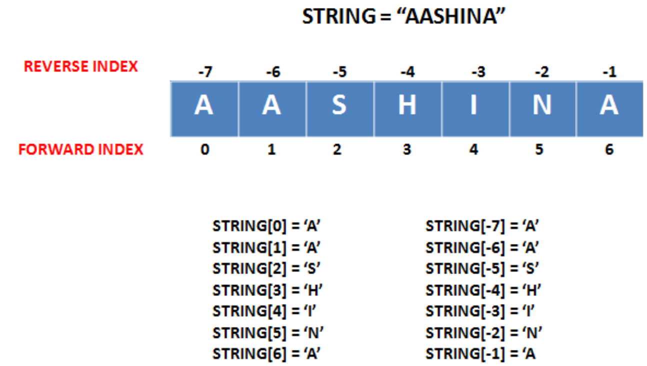
Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: August 27, 2024 at 14:25

The `str` Class

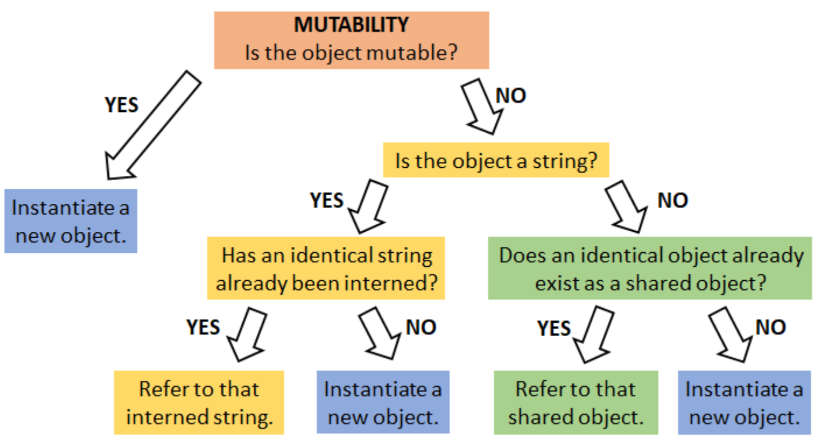
One of the most useful Python data types is the *string* type, defined by the `str` class. Strings are actually sequences of characters.

Strings are *immutable*, meaning you can't change them after they are created.



Object Creation/Instantiation

All immutable objects with the same content are stored as one object.



Creating Strings

Strings have some associated special syntax:

```
>>> s1 = str("Hello")     # using the constructor function
>>> s2 = "Hello"          # alternative syntax
>>> id(s1)                # strings are unique
139864255464424
>>> id(s2)
139864255464424
>>> s3 = str("Hello")
>>> id(s3)
139864255464424
>>> s1 is s2              # are these the same object?
True
>>> s2 is s3
True
```

Sequence Operations

Strings are *sequences of characters*. Below are some functions defined on sequence types, though not all supported on strings (e.g., `sum`).

Function	Description
`x in s`	x is in sequence s
`x not in s`	x is not in sequence s
`s1 + s2`	concatenates two sequences
`s * n`	repeat sequence s n times
`s[i]`	ith element of sequence (0-based)
`s[i:j]`	slice of sequence s from i to j-1
`len(s)`	number of elements in s
`min(s)`	minimum element of s
`max(s)`	maximum element of s
`sum(s)`	sum of elements in s
`for loop`	traverse elements of sequence
`<, <=, >, >=`	compares two sequences
`==, !=`	compares two sequences

Functions on Strings

Some functions that are available on strings:

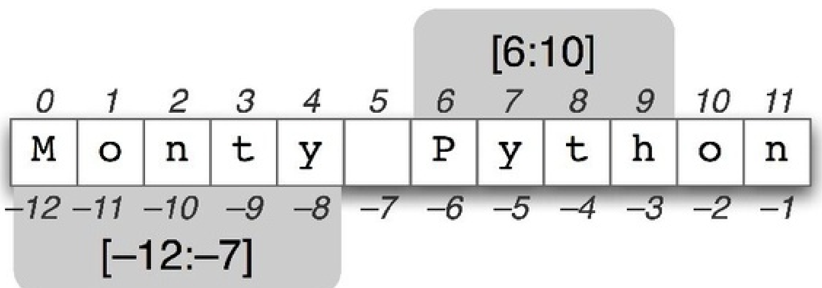
Function	Description
`len(s)`	return length of the string
`min(s)`	return char in string with lowest ASCII value
`max(s)`	return char in string with highest ASCII value

```
>>> s1 = "Hello, World!"
>>> len(s1)
13
>>> min(s1)
' '
>>> min("Hello")
'H'
>>> max(s1)
'r'
```

Why does it make sense for a blank to have lower ASCII value than any letter?

Indexing into Strings

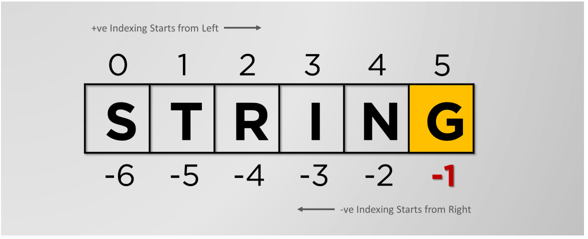
Strings are sequences of characters, which can be accessed via an index.



Indexes are 0-based, ranging from [0 ... len(s)-1].

You can also index using negatives, `s[-i]` means `s[len(s)-i]`.

Indexing into Strings




```
>>> s = "Hello, World!"
>>> s[0]
'H'
>>> s[6]
' '
>>> s[-1]
'!'
>>> s[-6]
'W'
>>> s[-6 + len(s)]
'W'
```

Slicing

Slicing means to select a contiguous subsequence of a sequence or string.

General Form:

`String[start : end]`

```
>>> s = "Hello, World!"
>>> s[1 : 4]            # substring from s[1]...s[3]
'ell'
>>> s[ : 4]             # substring from s[0]...s[3]
'Hell'
>>> s[1 : -3]           # substring from s[1]...s[-4]
'ello, Wor'
>>> s[1 : ]             # same as s[1 : s(len)]
'ello, World!'
>>> s[ : 5]             # same as s[0 : 5]
'Hello'
>>> s[:]                # same as s
'Hello, World!'
>>> s[3 : 1]            # empty slice
''
```


Concatenation and Repetition

General Forms:

```
s1 + s2
s * n
n * s
```

`s1 + s1` means to create a new string of `s1` followed by `s2`.
`s * n` or `n * s` means to create a new string containing `n` repetitions of `s`

```
>>> s1 = "Hello"
>>> s2 = ", World!"
>>> s1 + s2              # + is not commutative
'Hello, World!'
>>> s1 * 3               # * is commutative
'HelloHelloHello'
>>> 3 * s1
'HelloHelloHello'
```

Notice that concatenation and repetition *overload* two familiar operators.

Looking Back

In Slideset 5, we had code to compute and print a multiplication table up to LIMIT - 1,

```
> python MultiplicationTable.py
              Multiplication Table
     |   1   2   3   4   5   6   7   8   9
------------------------------------------
   1 |   1   2   3   4   5   6   7   8   9
   2 |   2   4   6   8  10  12  14  16  18
    ....
   9 |   9  18  27  36  45  54  63  72  81
```

which included:

```
    print("------------------------------------------")
```

That works well for LIMIT = 10, but not otherwise. How could you fix it?

```
    print("------" + "----" * (LIMIT - 1) )
```


in and not in operators

The `in` and `not in` operators allow checking whether one string is a *contiguous* substring of another.

General Forms:

```
s1 in s2
s1 not in s2
```

```
>>> s1 = "xyz"
>>> s2 = "abcxyzrls"
>>> s3 = "axbyczd"
>>> s1 in s2
True
>>> s1 in s3
False
>>> s1 not in s2
False
>>> s1 not in s3
True
```

Aside: Equality of Objects

There are two senses in which objects can be equal.

1. They can have equal contents; test with ==.
2. They can be literally the same object (same data in memory); test with `is`.

For elementary immutable object classes such as strings and numbers, these are the same. That's not necessary true for complex objects like lists or tuples.

For user-defined classes, `(o1 == o2)` is False unless `(o1 is o2)` or you've overloaded == by defining `__eq__` for the class.

Equality of Objects

```
>>> s1 = "xyzabc"
>>> s2 = "xyz" + "abc"
>>> s3 = str("xy" + "za" + "bc")
>>> s1 is s2                        # s1, s2, s3 are all
True                                # the same object in
>>> s2 == s3                        # memory
True
>>> s1 == s2
True
>>> from Circle import *
>>> c1 = Circle()                   # circle with radius 1
>>> c2 = Circle()                   # circle with radius 1
>>> c1 == c2                        # they're different
False
>>> c3 = c2                         # c3 is new pointer to c2
>>> c2 == c3                        # they're the same object
True
```


Equality of Objects

If two objects satisfy `(x is y)`, then they satisfy `(x == y)`, but not always vice versa.

```
>>> from Circle import *
>>> c1 = Circle()
>>> c2 = Circle()
>>> c3 = c2
>>> c1 is c2
False
>>> c3 is c2
True
>>> c1 == c2
False
>>> c2 == c3
True
```

If you define a class, you can override == and make any equality comparison you like.

Comparing Strings

In addition to equality comparisons, you can order strings using the relational operators: `<`, `<=`, `>`, `>=`.

For strings, this is *lexicographic* (or alphabetical) ordering using the ASCII character codes.

```
>>> "abc" < "abcd"
True
>>> "abcd" <= "abc"
False
>>> "Paul Jones" < "Paul Smith"
True
>>> "Paul Smith" < "Paul Smithson"
True
>>> "Paula Smith" < "Paul Smith"
False
```

Iterating Over a String

Sometimes it is useful to do something to each character in a string, e.g., change the case (lower to upper and upper to lower).

```
DIFF = ord('a') - ord('A')

def swapCase (s):
    result = ""
    for ch in s:
        if ( 'A' <= ch <= 'Z' ):
            result += chr(ord(ch) + DIFF )
        elif ( 'a' <= ch <= 'z' ):
            result += chr(ord(ch) - DIFF )
        else:
            result += ch
    return result

print(swapCase( "abCDefGH" ))
```

```
> python StringIterate.py
ABcdEFgh
```


Iterating Over a String

General Form:

```
for c in s:
    body
```

You can also iterate using the indexes:

```
def swapCase2 (s):
    result = ""
    for i in range(len(s)):
        ch = s[i]
        if ( 'A' <= ch <= 'Z' ):
            result += chr(ord(ch) + DIFF )
        elif ( 'a' <= ch <= 'z' ):
            result += chr(ord(ch) - DIFF )
        else:
            result += ch
    return result
```


What You Can't Do

```
def swapCaseWrong (s):
    for i in range(len(s)):
        if ( 'A' <= s[i] <= 'Z' ):
            s[i] = chr(ord(s[i]) + DIFF )
        elif ( 'a' <= s[i] <= 'z' ):
            s[i] = chr(ord(s[i]) - DIFF )
    return s

print(swapCaseWrong( "abCDefGH" ))
```

```
> python StringIterate.py
Traceback (most recent call last):
  File "StringIterate.py", line 38, in <module>
    print(swapCaseWrong( "abCDefGH" ))
  File "StringIterate.py", line 35, in swapCaseWrong
    s[i] = chr(ord(s[i]) - DIFF )
TypeError: 'str' object does not support item assignment
```

What went wrong?

Strings are Immutable

You can't change a string, by assigning at an index. You have to create a new string.

```
>>> s = "Pat"
>>> s[0] = 'R'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s2 = 'R' + s[1:]
>>> s2
'Rat'
```

Whenever you concatenate two strings or append something to a string, you create a new value. *Don't forget to save it!*

Let's Take a Break

Useful Testing Methods

Below are some useful methods.

Function	Description
s.isalnum():	nonempty alphanumeric string?
s.isalpha():	nonempty alphabetic string?
s.isdigit():	nonempty and contains only digits?
s.isidentifier():	follows rules for Python identifier?
s.islower():	nonempty and contains only lowercase letters?
s.isupper():	nonempty and contains only uppercase letters?
s.isspace():	nonempty and contains only whitespace?

Notice that these are methods of class `str`*, not functions, so must be called on a string s.*

```
>>> islower("xyz")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'islower' is not defined
```


Useful Testing Methods

```
>>> s1 = "abc123"
>>> s1.isalnum()
True
>>> s1.isalpha()
False
>>> "abcd".isalpha()
True
>>> "1234".isdigit()
True
>>> "abcd".islower()
True
>>> "abCD".isupper()
False
>>> "".islower()
False
>>> "".isdigit()
False
>>> "\t\n  \r".isspace()    # contains tab, newline, return
True
>>> "\t\n  xyz".isspace()   # contains non-whitespace
False
```


Example: Recognizer for Integers

Suppose you want to know if your string input represents a decimal integer, which may be signed. You might write the following:

```
def isInt( s ):
    return  s.isdigit()  \
        or (      (s[0] == '-' or s[0] == '+') \
              and s[1:].isdigit() )
```

Notice that this allows some peculiar inputs like +000000, but then so does Python.

Better Error Checking

When your program accepts input from the user, it's always a good idea to "validate" the input.

Earlier in the semester, we wrote:

```
# See if an integer entered is prime.
num = int( input("Enter an integer: ") )
< code to test if num is prime >
```

What's 'wrong' with this code?

Better Error Checking

When your program accepts input from the user, it's always a good idea to "validate" the input.

Earlier in the semester, we wrote:

```
# See if an integer entered is prime.
num = int( input("Enter an integer: ") )
< code to test if num is prime >
```

What's 'wrong' with this code?

If the string entered does not represent an integer, `int` might fail.

```
>>> num = int (input ("Enter an integer: "))
Enter an integer: 3.4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '3.4'
```


Better Error Checking

This is better:

```
# See if an integer entered is prime.
while (True):
   # recall that input returns a string
   stringInput = input("Enter a positive integer: ")
   if ( stringInput.isdigit() ):
      break
   else:
      print("Invalid input: not a positive integer.", \
            " Try again!")
# At this point, do we know that stringInput represents
# a positive integer? Any positive integer?
num = int( stringInput )
< code to test if num is prime >
```

This still isn't quite right. Can you see what's wrong?

Better Error Checking

This is better:

```
# See if an integer entered is prime.
while (True):
   # recall that input returns a string
   stringInput = input("Enter a positive integer: ")
   if ( stringInput.isdigit() ):
      break
   else:
      print("Invalid input: not a positive integer.", \
            " Try again!")
# At this point, do we know that stringInput represents
# a positive integer? Any positive integer?
num = int( stringInput )
< code to test if num is prime >
```

This still isn't quite right. Can you see what's wrong?

It doesn't allow +3, but does allow 0. How would you fix it?

Testing Our Code

```
> python IsPrime4.py
Enter a positive integer: -12
Invalid input: not a positive integer.  Try again!
Enter a positive integer: abcd
Invalid input: not a positive integer.  Try again!
Enter a positive integer: 57
57 is not  prime
```


Substring Search

We already saw that `in` and `not in` work on strings.

Python provides some other string methods to see if a string contains another as a substring:

Function	Description
`s.endswith(s1):`	does s end with substring s1?
`s.startswith(s1):`	does s start with substring s1?
`s.find(s1):`	lowest index where s1 starts in s, -1 if not found
`s.rfind(s1):`	highest index where s1 starts in s, -1 if not found
`s.count(s1):`	number of non-overlapping occurrences of s1 in s

Substring Search

```
>>> s = "Hello, World!"
>>> s.endswith("d!")
True
>>> s.startswith("hello")        # case matters
False
>>> s.startswith("Hello")
True
>>> s.find('l')                  # search from left
2
>>> s.rfind('l')                 # search from right
10
>>> s.count('l')
3
>>> "ababababa".count('aba')     # nonoverlapping occurrences
2
```


Converting Strings

Below are some additional methods on strings. Remember that strings are *immutable*, so these all make a new copy of the string. *They don't change s.*

Function	Description
`s.capitalize():`	return a copy with first character capitalized
`s.lower():`	lowercase all letters
`s.upper():`	uppercase all letters
`s.title():`	capitalize all words
`s.swapcase():`	lowercase letters to upper, and vice versa
`s.replace(old, new):`	replace occurences of old with new

So remember to save the result!

Don't Forget to Save the Result

A very common error is to forget what it means to be immutable: no operation changes the original string. If you want the changed result, you have to save it.

```
>>> s1 = "abCDefGH"
>>> s1.swapcase()
'ABcdEFgh'
>>> s1                    # s1 didn't change
'abCDefGH'
>>> s2 = s1.swapcase()    # save the result
>>> s2
'ABcdEFgh'
>>>
```

BTW: what happens to the result if you don't save it?

String Conversions

```
>>> "abcDEfg".upper()
'ABCDEFG'
>>> "abcDEfg".lower()
'abcdefg'
>>> "abc123".upper()          # only letters
'ABC123'
>>> "abcDEF".capitalize()
'Abcdef'
>>> "abcDEF".swapcase()       # only letters
'ABCdef'
>>> book = "introduction to programming using python"
>>> book.title()              # doesn't change book
'Introduction To Programming Using Python'
>>> book2 = book.replace("ming", "s")
>>> book2
'introduction to programs using python'
>>> book2.title()
'Introduction To Programs Using Python'
>>> book2.title().replace("Using", "With")
'Introduction To Programs With Python'
```


Stripping Whitespace

It's often useful to remove whitespace at the start, end, or both of string input. Use these functions:

Function	Description
`s.lstrip()`:	return copy with leading whitespace removed
`s.rstrip()`:	return copy with trailing whitespace removed
`s.strip()`:	return copy with leading and trailing whitespace removed

```
>>> s1 = "   abc   "
>>> s1.lstrip()              # new string
'abc   '
>>> s1.rstrip()              # new string
'   abc'
>>> s1.strip()               # new string
'abc'
>>> "a b c".strip()
'a b c'
```


Strip User Input

It's typically a good idea to strip user input to remove extraneous white space!

```
>>> ans = input("Please enter YES or NO: ")
Please enter YES or NO:   NO
>>> ans
'  NO  '
>>> ans == 'YES' or ans == 'NO'
False
>>> ans = input("Please enter YES or NO: ").strip()
Please enter YES or NO:     YES
>>> ans
'YES'
>>> ans == 'YES' or ans == 'NO'
True
>>>
```

Formatting Strings

Recall from Slideset 3, our functions for formatting strings. The `str` class also has some formatting options:

Function	Description
`s.center(w)`:	returns a string of length w, with s centered
`s.ljust(w)`:	returns a string of length w, with s left justified
`s.rjust(w)`:	returns a string of length w, with s right justified

```
s = "abc"
>>> s.center(10)              # new string
'   abc    '
>>> s.ljust(10)               # new string
'abc       '
>>> s.rjust(10)               # new string
'       abc'
>>> s.center(2)               # new string
'abc'
```


Looking Back (Again)

In Slideset 5, we had code to compute and print a multiplication table up to LIMIT - 1.

```
> python MultiplicationTable.py
          Multiplication Table
    |   1   2   3   4   5   6   7   8   9
-----------------------------------------
  1 |   1   2   3   4   5   6   7   8   9
      ...
```

which included the following code to center the title:

```
print("          Multiplication Table")
```

A better way would be:

```
print("Multiplication Table".center(6 + 4 * (LIMIT-1)))
```


Multiplication Table Revisited

With `LIMIT = 10`:

```
> python MultiplicationTable.py
         Multiplication Table
    |   1   2   3   4   5   6   7   8   9
-----------------------------------------
  1 |   1   2   3   4   5   6   7   8   9
  2 |   2   4   6   8  10  12  14  16  18
      ...
  9 |   9  18  27  36  45  54  63  72  81
```

With `LIMIT = 13`:

```
> python MultiplicationTable.py
               Multiplication Table
    |   1   2   3   4   5   6   7   8   9  10  11  12
-----------------------------------------------------
  1 |   1   2   3   4   5   6   7   8   9  10  11  12
  2 |   2   4   6   8  10  12  14  16  18  20  22  24
      ...
 12 |  12  24  36  48  60  72  84  96 108 120 132 144
```


String Example: CSV Files

A comma-separated values (csv) file is a common way to record data. Each line has multiple values separated by commas. For example, I can download your grades from Canvas in csv format:

```
Name,EID,HW1,HW2,Exam1,Exam2,Exam3
Possible,,10,10,100,100,100
Jones;Bob,bj123,10,9,99,60,45
Riley;Frank,fr498,4,8,72,95,63
Smith;Sally,ss324,5,10,100,75,80
```

Suppose you needed to process such a file. There's an easy way to extract that data (the Python string `split` method), which we'll cover soon.

But suppose you needed to write your own functions to extract the data from a line.

String Example: Line of csv Data

Later we'll explain how to process files. For now, let's process a line.

In file FieldToComma2.py:

```
def SplitOnComma ( str ):
    """ Given a string possibly containing a comma,
    return the initial string (before the comma) and
    the string after the comma. If there is no comma,
    return the string and the empty string. """
    if (',' in str):
        index = str.find(",")
        # Note: returns a pair of values
        return str[:index], str[index+1:]
    else:
        return str, ""
```

Notice that this returns a *pair* of values. How would you split on something other than a comma?

String Example: Line of csv Data

```
>>> from FieldToComma2 import *
>>> line = " abc  , def ,ghi, jkl  "
>>> first, rest = SplitOnComma( line )
>>> first
' abc  '
>>> rest
' def ,ghi, jkl  '
>>> first, rest = SplitOnComma(rest)
>>> first
' def '
>>> rest
'ghi, jkl  '
```


String Example

```
def SplitFields( line ):
    """ Iterate through a csv line to extract and print
    the values, stripped of extra whitespace. """
    rest = line.strip()
    i = 1
    while (',' in rest):
        next, rest = SplitOnComma( rest )
        print("Field", i, ": ", next.strip(), sep = "")
        i += 1
    print("Field", i, ": ", rest.strip(), sep = "")
```

```
>>> from FieldToComma2 import *
>>> csvLine = "  xyz , 123  ,a, 12, abc    "
>>> SplitFields( csvLine )
Field1: xyz
Field2: 123
Field3: a
Field4: 12
Field5: abc
```


Next stop: Lists.