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.

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
True
>>> s2 is s3
True
```
Strings are sequences of characters. Below are some functions defined on sequence types, though not all supported on strings (e.g., sum).

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

Some functions that are available on strings:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(s)</td>
<td>return length of the string</td>
</tr>
<tr>
<td>min(s)</td>
<td>return char in string with lowest ASCII value</td>
</tr>
<tr>
<td>max(s)</td>
<td>return char in string with highest ASCII value</td>
</tr>
</tbody>
</table>

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

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

Strings are sequences of characters, which can be accessed via an index. Indexes are 0-based, ranging from [0 ... len(s)-1].

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\text{Monty Python} & & & & & & & & & & & \\
-12 & -11 & -10 & -9 & -8 & -7 & -6 & -5 & -4 & -3 & -2 & -1
\end{array}
\]

Indexing into Strings

Indexing into Strings
**Slicing**

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

**General Form:**

```
String[start : end]
```

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

```python
>>> 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,

```bash
> python MultiplicationTable.py
```

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
</tr>
</tbody>
</table>
```

which included:

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

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

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

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`

```python
>>> s1 = "xyz"
>>> s2 = "abcxyzr1s"
>>> 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.
- They can have equal contents; test with `==`.
- They can be literally the same object (same data in memory); test with `is`.

For immutable object classes such as strings and numbers, these are the same.

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

---

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.

```python
>>> "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).

```python
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"))
```

More on Strings

General Form:
```
for c in s:
    body
```

You can also iterate using the indexes:

```python
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

print(swapCase2("abCDefGH"))
```

What You Can't Do

```python
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"))
```

Strings are Immutable

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

```python
>>> s = "Pat"
>>> s[0] = 'R'
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
>>> 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!
Useful Testing Methods

Below are some useful methods.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.isalnum()</code></td>
<td>nonempty alphanumeric string?</td>
</tr>
<tr>
<td><code>s.isalpha()</code></td>
<td>nonempty alphabetic string?</td>
</tr>
<tr>
<td><code>s.isdigit()</code></td>
<td>nonempty and contains only digits?</td>
</tr>
<tr>
<td><code>s.isidentifier()</code></td>
<td>follows rules for Python identifier?</td>
</tr>
<tr>
<td><code>s.islower()</code></td>
<td>nonempty and contains only lowercase letters?</td>
</tr>
<tr>
<td><code>s.isupper()</code></td>
<td>nonempty and contains only uppercase letters?</td>
</tr>
<tr>
<td><code>s.isspace()</code></td>
<td>nonempty and contains only whitespace?</td>
</tr>
</tbody>
</table>

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

```python
>>> 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:

```python
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:

```python
# 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.

```python
>>> 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'
```

This is better:

```python
# 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?

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
> 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.endswith(s1)</code>:</td>
<td>does <code>s</code> end with substring <code>s1</code>?</td>
</tr>
<tr>
<td><code>s.startswith(s1)</code>:</td>
<td>does <code>s</code> start with substring <code>s1</code>?</td>
</tr>
<tr>
<td><code>s.find(s1)</code>:</td>
<td>lowest index where <code>s1</code> starts in <code>s</code>, -1 if not found</td>
</tr>
<tr>
<td><code>s.rfind(s1)</code>:</td>
<td>highest index where <code>s1</code> starts in <code>s</code>, -1 if not found</td>
</tr>
<tr>
<td><code>s.count(s1)</code>:</td>
<td>number of non-overlapping occurrences of <code>s1</code> in <code>s</code></td>
</tr>
</tbody>
</table>

```python
>>> 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`.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.capitalize()</code>:</td>
<td>return a copy with first character capitalized</td>
</tr>
<tr>
<td><code>s.lower()</code>:</td>
<td>lowercase all letters</td>
</tr>
<tr>
<td><code>s.upper()</code>:</td>
<td>uppercase all letters</td>
</tr>
<tr>
<td><code>s.title()</code>:</td>
<td>capitalize all words</td>
</tr>
<tr>
<td><code>s.swapcase()</code>:</td>
<td>lowercase letters to upper, and vice versa</td>
</tr>
<tr>
<td><code>s.replace(old, new)</code>:</td>
<td>replace occurrences of old with new</td>
</tr>
</tbody>
</table>

So remember 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.

```python
>>> 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

```python
>>> "abcDEfg".upper()
'ABCDEFG'
>>> "abcDEfg".lower()  # only letters
'abcdefg'
>>> "abc123".upper()  # only letters
'ABC123'
>>> "abcdef".capitalize()
'Abcdef'
>>> "abcdef".swapcase()  # only letters
'ABCdef'
```

Stripping Whitespace

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.lstrip()</td>
<td>return copy with leading whitespace removed</td>
</tr>
<tr>
<td>s.rstrip()</td>
<td>return copy with trailing whitespace removed</td>
</tr>
<tr>
<td>s.strip()</td>
<td>return copy with leading and trailing whitespace removed</td>
</tr>
</tbody>
</table>

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

Strip User Input

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

```python
>>> 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
```
Recall from Slideset 3, our functions for formatting strings. The \texttt{str} class also has some formatting options:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{s.center(w)}:</td>
<td>returns a string of length ( w ), with ( s ) centered</td>
</tr>
<tr>
<td>\texttt{s.ljust(w)}:</td>
<td>returns a string of length ( w ), with ( s ) left justified</td>
</tr>
<tr>
<td>\texttt{s.rjust(w)}:</td>
<td>returns a string of length ( w ), with ( s ) right justified</td>
</tr>
</tbody>
</table>

\begin{verbatim}
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 '
\end{verbatim}

In Slideset 5, we had code to compute and print a multiplication table up to \( \text{LIMIT} - 1 \).

```bash
> 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 the following code to center the title:

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

A better way would be:

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

### Multiplication Table Revisited

With \( \text{LIMIT} = 10 \):

```bash
> 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 \( \text{LIMIT} = 13 \):

```bash
> 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 |
```

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,Francis,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 \texttt{split} method), which we'll cover soon.

But suppose you needed to write your own functions to extract the data from a line.
Later we’ll explain how to process files. For now, let’s process a line.

In file FieldToComma2.py:

```python
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?

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

Next stop: Lists.