Lecture #11: Strings, Mutable Data

Strings: A Specialized Type of Sequence

- Strings are sequences of characters, with a good deal of special syntax.
- Rather odd property: the base cases are circular. Characters are themselves strings of length 1!
- The usual operations on tuples apply also to strings:

```python
>>> "abcd"[0]
'a'
>>> len("abcd")
4
>>> "abcd"[1:3]
'bc'
>>> "ab" + "cd"  
'abcd'
>>> "x" * 5
"xxxxx"
>>> for c in "abcd":
    print(c, end=" 
 a, b, c, d,
```

Modified Operations

- Membership is not quite the same for strings as for tuples:

```python
>>> 'b' in ('a', 'b', 'c', 'd')  # A sequence, not a string  
True
>>> 'bc' in ('a', 'b', 'c', 'd')  # But...
False
>>> 'b' in 'abcd'  
True
>>> 'bc' in 'abcd'  # in Finds substrings
True
```

- The substring is generally more important than the character, in other words.

Numerous Functions and Methods

- The calls `str(x)` and `x._str_()` convert values of any type into strings that depict them:

```python
>>> str(3+7)
'10'
```

- The methods reflect common manipulations from "real life":

```python
>>> "i can’t find my shift key".capitalize()
'I can’t find my shift key'
>>> "chAIlGe".upper() + " CasE".lower() + " randOMLY".swapcase()
'CHArGE case RANDOMLY'
>>> "1234".isnumeric() and "abcd".isalpha()  
True
>>> 'sNakeeyes'.upper().endswith('YES')
True
>>> '{x} + {y} = {answer}'.format(answer=7, x=3, y=4)
'3 + 4 = 7'
```

A Cast of Thousands

- Python3 uses Unicode its basic character set: an international standard comprising most alphabets (dead and alive).
- Characters have standard numbers (indicating position in the character set) and names. The Python `ord` and `chr` convert from character to number and back.
- Getting your computer to actually render them all properly, however, is another matter entirely, which is outside Python.
- The character codes from 0-127 (7-bit codes) are known as ASCII (American Standard Code for Information Interchange). Everything you typically type uses this subset.
- Nice property: 1 byte (8 bits) per character.
- This is lost with Unicode, but since there is an extra bit, we can encode larger character codes (UTF-8).

Denoting Characters and Strings

- You've seen string literals all along. Python has 8 (!) styles. Consider the string

```python
\begin{quote}
  I'd rather be in Philadelphia.
\end{quote}
```

which we can write:

```python
>>> "\begin{quote}
  I'd rather be in Philadelphia.
\end{quote}"
>>> "\begin{quote}I'd rather be in Philadelphia.\end{quote}"
>>> "\begin{quote}I'd rather be in Philadelphia."\end{quote}"
>>> "\begin{quote}I'd rather be in Philadelphia."\end{quote}"
>>> "\begin{quote}I'd rather be in Philadelphia."\end{quote}"
>>> "\begin{quote}I'd rather be in Philadelphia."\end{quote}"
```
Escapes

- The \ escape allows us to introduce special, non-graphical characters: `newline \n, tab \t`
- Or to insert quoting characters.
- Or Unicode characters:
  ```
  "kαβγζב\u05d0聱聲"
  "☺☹"
  ```
  [See demo].

Strings as Sequences

- Most string operations are variations on the sequence operations we've seen.
- Example: take a string, break it into lines, indent the lines by \N spaces, glue the lines back together, and return the result
  ```python
  def indent_lines(s, n):
      return "\n".join(map(lambda line: " " * n + line, s.split('\n')))
  ```
  Use it to indent a file:
  ```python
  print(indent_lines(open("afile").read(), 4))
  ```
- An even more general manipulation: regular expressions:
  ```python
  import re
  def indent_lines(s, n):
      return re.sub(r'(?m)^', ' ' * n, a)
  ```

Further exploration left to the reader.

Immutable Values

- The last weeks have concentrated on immutable data: Values, once created, are not changed.
- For example:
  ```
  >>> X, Y = (1, 2, 3), (3, 4, 5)
  >>> Z = (X, Y)
  >>> X = (0, -1)
  >>> Z
  ((1, 2, 3), (3, 4, 5))
  ... just as you'd expect for X and Y integers.
  ```

Local Variables

- What we have changed are local variables.
- But our uses of local variables have generally been such that we could replace all of them with parameters that we don't assign to.
- So instead of:
  ```python
  def sum_every_other(A):
      S = 0
      for i in range(0, len(A), 2):
          S += A[i]
  ```
  Alternative:
  ```python
  def sum_every_other(A):
      def sum(i, S):
          if i >= len(A): return S
          else return sum(i+2, S+A[i])
      return sum(0, 0)
  ```
  Further exploration left to the reader.

Referential Transparency

- This discipline of not changing things once they are created leads to the property of referential transparency: One may freely substitute a value for a variable having an equal value without changing the meaning of a program.
- When we can change data after creation, this property is lost.
- For example, in Python, tuples are immutable, so that these two fragments are indistinguishable, regardless of the contents of `...`:
  ```
  x = (1, 2, 3)  x = (1, 2, 3)
  y = (1, 2, 3)  y = x
  ...
  ```
- But we can change lists in Python:
  ```
  x = [1, 2, 3]  x = [1, 2, 3]
  y = [1, 2, 3]  y = x
  y[0] = 0  y[0] = 0
  print x[0]  print x[0]
  ```
  print two different things (1 vs. 0).

Mutation and Functions

- Let's work from an example:
  ```python
  def make_counter(start, limit):
      def next():
          """Increment the counter value, and return previous value. Returns None if counter is at the limit."""
          nonlocal start
          if start == limit:
              return None
          start += 1
          return start-1
      return next
  ```
  ```
  nonlocal start
  if start == limit:
      return None
  start += 1
  return start-1
  ```
  The new nonlocal statement says "Assignments to start in this function do not create a new local variable. Rather, they refer to the existing start defined outside (in make_counter)."
Using Counters

• I can now write a loop like this:

```python
>>> c = make_counter(0, 10)
>>> while True:
...     k = c()
...     if k is None:
...         break
...     print(c, end=",”)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

• Each call to `c` returns a different value: referential transparency clearly does not apply.