1 Administrative Topics

- Return Quiz 2

2 Strings

2.1 Creating String Literals

We have seen how to create string literals, by placing some text between pairs of single quotes or pairs of double quotes, e.g. “hi” and ‘ho’. Once a string has been created in Python, Python “forgets” which quotes were used to delimit it. In other words, the quotes are just used to create it and don’t determine how the string is stored. For example, if we type

```python
myString = "hi"
```

at the Python interpreter prompt (i.e. we use double quotes), then type

```python
myString
```

it will print out ‘hi’ (single quotes).

However, if we actually call `print`, it won’t print the quotes at all, e.g.

```python
print myString
```

results in hi.
2.1.1 Escape Characters

Strings can have special characters like tabs and carriage returns encoded with *escape sequences*. Escape sequences begin with a backslash. Table ?? shows a few that are useful when reading text that has been read in from a file.

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'</td>
<td>single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote</td>
</tr>
<tr>
<td>\t</td>
<td>tab</td>
</tr>
<tr>
<td>\n</td>
<td>newline (return in Unix)</td>
</tr>
<tr>
<td>\r</td>
<td>carriage return</td>
</tr>
</tbody>
</table>

An escape sequence is considered as a single character – e.g. `len('h\th')` is 3.

Carriage returns, newlines, tabs and spaces are all considered whitespace.

2.2 String Operations

We have seen how to concatenate two strings, e.g. “hi” + “ho” is “hiho”.

There are several other operators that apply to strings shown in Table ??.

Here are some examples illustrating the operations (with the command typed at the Python prompt and the result that is displayed):

- "man" in 'human' ⇒ True
- "woman" not in 'human' ⇒ True
- "hi" = "ho" ⇒ 'hiho'
- "ho " * 3 ⇒ 'ho ho ho '
Table 2: String Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>True if an item of s is equal to x, else False</td>
</tr>
<tr>
<td>x not in s</td>
<td>False if an item of s is equal to x, else True</td>
</tr>
<tr>
<td>s + t</td>
<td>the concatenation of s and t</td>
</tr>
<tr>
<td>s * n, n * s</td>
<td>n shallow copies of s concatenated</td>
</tr>
<tr>
<td>s[i]</td>
<td>i\textsuperscript{th} item of s, origin 0</td>
</tr>
<tr>
<td>s[i:j]</td>
<td>slice of s from i to j (but not including the j\textsuperscript{th} item)</td>
</tr>
<tr>
<td>s[i:j:k]</td>
<td>slice of s from i to j with step k (but not including the j\textsuperscript{th} item)</td>
</tr>
<tr>
<td>len(s)</td>
<td>length of s</td>
</tr>
</tbody>
</table>

• 'blast'[0] ⇒ 'b'
• 'blast'[1:5] ⇒ 'last'
• 'blast'[0:5:2] ⇒ 'bat'
• len('blast') ⇒ 5
• min('blast') ⇒ 'a'
• max('blast') ⇒ 't'

And all of these examples produce identical results if they are applied to a variable containing a string, e.g. after the line

myString = 'blast'

we reproduce the same results:

• myString[0] ⇒ 'b'
• myString[1:5] ⇒ 'last'
• myString[0:5:2] ⇒ 'bat'
• len(myString) ⇒ 5
This is true because the operations are performed on the value itself – symbols just let us get to the values.

Sub: When I write myString[0], I pronounce it “myString sub 0”. Sub is an abbreviation of subscript and is related to the way we refer to elements of sets, vectors, and matrices in math.

### 2.3 String Methods

Guess what? Strings are objects, so they have methods.¹

One of the most useful string methods is find, which searches for the first occurrence of one string within another and returns its location. Its syntax, along with that of several other string methods is in Table ??.

Here are several examples using the string methods:

- "ho ho ho".upper() ⇒ "HO HO HO"
- "ho ho ho".split() ⇒ ['ho', 'ho', 'ho']
- "ho,ho,ho".split(’,’) ⇒ ['ho', 'ho', 'ho']
- "ho,ho,ho".split(’,’,1) ⇒ ['ho', 'ho,ho']

For a complete list of string methods, see [http://docs.python.org/library/stdtypes.html#string-methods](http://docs.python.org/library/stdtypes.html#string-methods)

### 2.4 Strings Methods Return New Strings

Notice that all of these methods return a new string with the result. None of them are changing the original string. This is because strings are immutable in Python. Once a string has been made, it will always be the same. But that doesn’t mean we can’t just reuse a variable, e.g.

---

¹In fact, all data types in Python are actually objects. Yes, even numeric types are objects, but there are no commonly used methods on numeric types. For the purposes of CS151, there is no need to think of numbers as objects, so we ignore that fact. New in Python 2.6 is a method called hex which returns the hexadecimal representation of a float
### Table 3: Selected String Methods

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s.upper()</code></td>
<td>returns a copy of <code>s</code> in upper case</td>
</tr>
<tr>
<td><code>s.lower()</code></td>
<td>returns a copy of <code>s</code> in lower case</td>
</tr>
<tr>
<td><code>s.title()</code></td>
<td>returns a copy of <code>s</code> in title case (the first letter of each “word” is upper case)</td>
</tr>
<tr>
<td><code>s.strip()</code></td>
<td>Return a copy of the string with the leading and trailing whitespace characters removed</td>
</tr>
<tr>
<td><code>s.strip(chars)</code></td>
<td>Return a copy of the string with the leading and trailing characters removed where the characters are specified by the string <code>chars</code></td>
</tr>
<tr>
<td><code>s.split()</code></td>
<td>Returns a list of words in the string, using whitespace as the delimiter string: runs of consecutive whitespace are regarded as a single separator, and the result will contain no empty strings at the start or end if the string has leading or trailing whitespace</td>
</tr>
<tr>
<td><code>s.split(sep)</code></td>
<td>Return a list of the words in the string, using <code>sep</code> as the delimiter string</td>
</tr>
<tr>
<td><code>s.split(sep, maxsplit)</code></td>
<td>Return a list of the words in the string, using <code>sep</code> as the delimiter string with at most <code>maxsplit</code> done</td>
</tr>
<tr>
<td><code>s.find(findstr)</code></td>
<td>If <code>findstr</code> is in <code>s</code>, then returns the index of the first character of <code>findstr</code> within <code>str</code>, else returns -1</td>
</tr>
<tr>
<td><code>s.find(findstr, start)</code></td>
<td>The same as above, but doesn’t begin looking in <code>s</code> until the <code>start</code>’th character</td>
</tr>
<tr>
<td><code>s.find(findstr, start, end)</code></td>
<td>The same as above, but doesn’t look past the <code>end</code>’th character</td>
</tr>
</tbody>
</table>
str = 'defenestration of prague'
str = str.title()

Now `str` contains the string 'Defenestration Of Prague'.

3 Converting strings to other types

In certain situations, such as when we are using command line arguments, we have strings that represent numbers. To use the numeric value of the string, we need to convert it. We saw this in Project 3.

- Converting to an integer value. Use the function `int()` For example, if you have a variable called `istr` and it contains a string representing an integer value (e.g. "3"), then you would call `int(istr)`.

- Converting to a floating point value. Using the function `float()`. For example, if you have a variable called `fstr` and it contains a string representing an integer value (e.g. "3.1"), then you would call `float(fstr)`.

- Converting to a boolean value. There is no equivalent function. If you have a variable called `bstr` and it contains a string representing a boolean value (e.g. "True") and you want to put its boolean value in a variable called `truthiness`, The best thing to do is this:

  ```python
  truthiness = bstr == "True"
  ```

  and if you want to be case insensitive, then you could use this code:

  ```python
  truthiness = bstr.title() == "True"
  ```

4 Quiz Yourself

1. How would you use the slice operation to retrieve the substring “can” from the string “catatonic” (recall that the square brackets are used to slice strings)?
2. How would you separate the string ”2.3e-9” into two strings – one containing the mantissa and the other containing the exponent?

3. Familiarize yourself with the String Methods at http://docs.python.org/library/stdtypes.html. How would you remove the trailing spaces from a string?

4. How would you check to see if all characters in a string are numbers?

5. How would you count the number of times “ACCT” appears in “AC-CTTGGCACCT”?

5 Default parameters and keyword arguments

We are returning to a topic we covered a couple of weeks ago. A couple of weeks ago, we talked about default parameters. Today, I will reintroduce that and add a discussion of keyword arguments.

When you are defining a function, you identify parameters, and it is possible to provide a default value for that parameter. When you call that function, you can choose to provide a value for that parameter if you want to.

```python
def sayHi(name = 'You '):
    print "Hey ,", name

sayHi()
sayHi("Stephanie")
sayHi("You")
```

will print out

Hey , You
Hey , Stephanie
Hey , You

When you call a function, the value you provide as input to the function is called an argument. In the above case Stephanie is an argument. There are two ways to provide arguments. One is using positional notation. This is what we have been using so far. The first argument is the value for the first parameter. The second argument is the value for the second parameter.
And so on and so forth. But there is another way to do it, using *keyword* notation. In this case, you use the name of the parameter and the value of the argument together in something that looks like an assignment statement.

```
sayHi( name = "George" )
```

For functions with many parameters, you can provide the first set of arguments by position and the last set by keyword, but once you start using keywords, you can’t stop.

In the next example, we examine code that shows the power of both default parameter values and keyword arguments. Consider the following function which has a lot of parameters, some of which have default values:

```
def circle(x, y, scale, fill=True, fcolor = 'black', pcolor = 'black', psize = 1):
    # lots of code here.
```

Then the following calls are legal:

```
circle( 100 , 0, 1.0 , psize = 3 )
circle(x=100 , y=0 , scale=1.0 , pcolor='red' )
circle( 100 , 0, 1.0 )
circle( scale = 1.0 , x = 1, y = 2 )
```

And the following calls are ILLEGAL:

```
1 circle( 100 , 0, 1, pcolor='red', 1 )
2 circle( 100 , 0 )
3 circle( 100 , 0, fill=False )
```

Line 1 is illegal because it uses a positional argument (1) after a keyword argument.

Line 2 is illegal because it fails to provide a value for `scale`. Since `scale` does not have a default value, the caller must supply an argument for it.

Line 3 is illegal for the same reason.