



COMP2720: Automating Tools for New Media

Text as a medium, and
manipulating strings



New programming syntax and concepts

- Up until now, we've had a small set of programming elements we've worked with:
 - Assignment (`x= 42`), `print`, `for` (with and without `range()`), `if`
- We're halfway through the course, so we're going to start pulling back the curtains a little and show what's behind the scenes
- Two new concepts:
 - Dot notation for objects
 - Lists



Text

- Text is the universal medium
 - We can convert any other media to a text representation.
 - We can convert between media formats using text.
 - Text is simple.
- Like sound, text is usually processed in an *array* — a long line of characters.
- We refer to these long lines of characters as *strings*.

Strings

- Strings are defined with quote marks.
- **Python** actually supports three kinds of quotes:
 - >>> print 'this is a string'
this is a string
 - >>> print "this is a string"
this is a string
 - >>> print """this is a string"""
this is a string
- Use the right one that allows you to embed quote marks you want
 - >>> aSingleQuote = "'" # that is " ' " pushed together
 - >>> print aSingleQuote
 - '

Why would you want to use triple quotes?

- To have long quotations with new lines and such inside them.

```
>>> print aLongString()
```

```
This is a
```

```
long
```

```
string
```

```
>>>
```

```
def aLongString():
```

```
    return """This is a  
long  
string"""
```

Encodings for strings

- Strings are just *arrays* of characters.
- In most cases, characters are just single bytes.
 - The *ASCII* encoding standard maps between single byte values and the corresponding characters.
- More recently, characters are two bytes.
 - *Unicode* uses two bytes per characters so that there are encodings for glyphs (characters) of other languages (Japanese, Chinese, Thai, etc.)
 - *Java* uses Unicode. The version of Python we are using is based in Java, so our strings are actually using Unicode.
 - See <http://www.unicode.org> for more information.



There are more characters than we can type

- Our keyboards don't have all the characters available to us, and it's hard to type others into strings.
 - Backspace?
 - Return?
 - و?
- We use *backslash* escapes to get other characters in to strings.

Backslash escapes

- “\b” is backspace.
- “\n” is a newline (pressing the Enter key).
- “\t” is a tabulator.
- “\uXXXX” is a Unicode character, where XXXX is a code and each X can be 0-9 or A-F (*hexadecimal!*).
 - See: <http://www.unicode.org/charts/>
 - Must precede the string with “u” for Unicode to work.

Testing strings

```
>>> print "hello\tthere\nMark"
```

```
hello  there
```

```
Mark
```

```
>>> print u"\uFEED"
```

```
ؑ
```

```
>>> print u"\u03F0"
```

```
π
```

```
>>> print "This\bis\na\btest"
```

```
Thisis
```

```
atest
```

Manipulating strings

- We can add strings and get their lengths using the kinds of programming features we've seen previously.

```
>>> helloStr = "Hello"
>>> print len(helloStr)
5
>>> markStr = ", Mark"
>>> print len(markStr)
6
>>> print helloStr+markStr
Hello, Mark
>>> print len(helloStr+markStr)
11
```



Getting parts of strings

- We use the square bracket `[]` notation to get parts of strings.
- `string[n]` gives you the *n*th character in the string (but keep in mind the first one is the *zero-th*).
- `string[n:m]` gives you the *n*th up to (but not including) the *m*th character.

Getting parts of strings

```
>>> helloStr = "Hello"
```

```
>>> print helloStr[1]
```

```
e
```

```
>>> print helloStr[0]
```

```
H
```

```
>>> print helloStr[2:4]
```

```
ll
```

| | | | | |
|----------|----------|----------|----------|----------|
| H | e | l | l | o |
|----------|----------|----------|----------|----------|

0 1 2 3 4

Start and end assumed if not there

```
>>> print helloStr
```

```
Hello
```

```
>>> print helloStr[:3]
```

```
Hel
```

```
>>> print helloStr[3:]
```

```
lo
```

```
>>> print helloStr[:]
```

```
Hello
```

Objects: Dot notation

- All data in Python are actually *objects*.
- Objects not only store data, but they respond to special functions that only objects of the same type understand.
- We call these special functions *methods*.
 - Methods are functions known only to certain objects.
- To execute a method, you use dot notation
 - *Object.method()*

Capitalize is a method known only to strings

```
>>> test="this is a test."
```

```
>>> print test.capitalize
```

```
<builtin method 'capitalize'>
```

```
>>> print test.capitalize()
```

```
This is a test.
```

```
>>> print capitalize(test)
```

```
A local or global name could not be found.
```

```
NameError: capitalize
```

```
>>> print 'this is another test'.capitalize()
```

```
This is another test
```

```
>>> print 12.capitalize()
```

```
A syntax error is contained in the code -- I can't read it as Python.
```

```
Why?
```

Strings are sequences

```
>>> for i in "Hello":
```

```
...     print i
```

```
...
```

```
H
```

```
e
```

```
l
```

```
l
```

```
o
```

Useful string methods

- `startswith(prefix)` returns true if the string starts with the given suffix.
- `endswith(suffix)` returns true if the string ends with the given suffix.
- `find(findstring)` and `find(findstring,start)` and `find(findstring,start,end)` finds the findstring in the object string and returns the *index number* where the string starts. You can tell it what index number to start from, and even where to stop looking. It returns *-1* if it does not find the string.
- There is also `rfind(findstring)` (and variations) that searches from the end of the string toward the front.

Demonstrating startswith()

```
>>> letter = "Mr. Mark Guzdial requests the pleasure of your  
company..."
```

```
>>> print letter.startswith("Mr.")
```

```
1
```

```
>>> print letter.startswith("Mrs.")
```

```
0
```

Remember that Python sees "0" as false and anything else (including "1") as true

Demonstrating endswith()

```
>>> filename="barbara.jpg"
>>> if filename.endswith(".jpg"):
...     print "It's a picture"
...
It's a picture
```

Demonstrating find()

```
>>> print letter
```

```
Mr. Mark Guzdial requests the pleasure of your company...
```

```
>>> print letter.find("Mark")
```

```
4
```

```
>>> print letter.find("Guzdial")
```

```
9
```

```
>>> print len("Guzdial")
```

```
7
```

```
>>> print letter[4:9+7]
```

```
Mark Guzdial
```

```
>>> print letter.find("fred")
```

```
-1
```

replace() method

```
>>> print letter
```

```
Mr. Mark Guzdial requests the pleasure of your company...
```

```
>>> letter.replace("a","!")
```

```
'Mr. M!rk Guzdi!l requests the ple!sure of your comp!ny...'
```

```
>>> print letter
```

```
Mr. Mark Guzdial requests the pleasure of your company...
```

Useful methods to use with **lists** (not all work with strings)

- `append(something)` puts something in the list at the end.
- `remove(something)` removes something from the list, if it's there (otherwise returns an error).
- `sort()` puts the list in alphabetical order.
- `reverse()` reverses the list.
- `count(something)` tells you the number of times that something is in the list (**works with string too**).
- `max()` and `min()` are **global** functions (we've seen them before) that take a list as input and give you the maximum and minimum value in the list (**works with string too**).

Converting from strings to lists

```
>>> print letter.split(" ")
```

```
['Mr.', 'Mark', 'Guzdial', 'requests', 'the', 'pleasure', 'of', 'your',  
'company...']
```



Lists

- We've seen lists before — that's what `range()` returns.
- Lists are very powerful structures.
 - Lists can contain strings, numbers, even other lists.
 - They work very much like strings.
 - You get pieces out with `[]`.
 - You can “add” lists together.
 - You can use for loops on them.
 - We can use them to process a variety of kinds of data.

Demonstrating lists

```
>>> mylist = ["This","is","a", 12]
>>> print mylist
['This', 'is', 'a', 12]
>>> print mylist[0]
This
>>> for i in mylist:
...     print i
...
This
is
a
12
>>> print mylist + ["Really!"]
['This', 'is', 'a', 12, 'Really!']
```

Strings have no font

- Strings are only the characters of text displayed “WYSIWYG” (*What You See is What You Get*).
 - WYSIWYG text includes fonts and styles.
- The font is the characteristic look of the letters in all sizes.
- The style is typically the **boldface**, *italics*, underline, and ~~other effects~~ applied to the font.
 - In printer’s terms, each style is its own font.

Encoding font information

- Font and style information is often encoded as *style runs*.
 - A separate representation from the string.
 - Indicates bold, italics, or whatever style modification; start character; and end character.
- **The *old brown* fox runs.**
- Could be encoded as:
"The old brown fox runs."
[[bold 0 6] [italics 5 12]]



How do we encode all that?

- Is it a single value? Not really.
- Do we encode it all in a complex list? We could.
- How do most text systems handle this?
 - *As objects.*
 - Objects have data, maybe in many parts.
 - Objects know how to act upon their data.
 - Objects' methods may be known only to that object, or may be known by many objects, but each object performs that method differently.



What can we do with all this?

- Answer: Just about anything!
- *Strings* and *lists* are about as powerful as one gets in Python
 - By “powerful,” we mean that we can do a lot of different kinds of computation with them.
- Examples:
 - Pull up a Web page and grab information out of it, through the use of a function.
 - Find a nucleotide sequence in a string and print its name.
 - Manipulate a function’s source.
- But first, we have to learn how to manipulate files...