Python Programming for Data Science

Week 36, Friday

Sequential types

- Lists
- A bit more on strings

Loops

Reading/Writing to files

Recap: datatypes — a popquiz

- How many different numeric types are there in Python?
- What are booleans used for?
- How do you create an empty string?
- What does "\n" mean?
- What does the * do for strings?
- Can you add a string to a number?

Lists

Types: Lists

Lists are a *collection* type. They allow you to gather several values and manipulate them as one.
Lists are ordered collections of items, accessible by an index.

- lines from a file
- multiple sequence alignment
- results from a series of experiments
- ...

A list is created using square brackets:

```
l = [1, 2.0, "three", 4]    # Lists can contain values of different types
l = [1,2,3,4,5,6]
```

Types: Lists — indexing

You can index in lists using square brackets:

```
listname[n]
```

this will return the n+1'th element in the list.

```
my_list = ['a', 'b', 'c']
print(my_list[1])
```

```
output
'b'
```

Note: sequences types in python are zero indexed. The first element has index 0.

Use a negative index to count from the end of the list:

```
print(my_list[-1])       # will print 'c'
```

Indexing: lists vs strings

Lists and strings are both *sequence types*, and have similar properties. Indexing in strings works exactly as for lists:

```
my_str = "hello"
print(my_str[2])
```

```
output

'l'
```

One difference is that in lists, you can *assign* to an index:

```
my_list = ['a', 'b', 'c']
my_list[2] = 'd'
print(my_list)
```

```
output

['a', 'b',
'd']
```

You cannot do this to strings (they are *immutable*).

Types: Strings & lists: slicing

Remember: Lists and strings are both *sequence* types
Sequence types support *slicing* to extract subsequences:

```
sequence[m:n]
```

This returns a sequence with the elements from index m up to n-1. (note: endpoint not included)

```
s = "hello"
print(s[1:4])
```

```
output

'ell'
```

Optionally, you can provide a third argument to a slice, specifying a step-size:

```
print(s[1:5:3])
```

```
output
'eo'
```

Types: Strings & lists: slicing (2)

Range parameters can be omitted, indicating that everything in that direction should be included:

```
s = "hello"
print(s[:4])
print(s[1:])
print(s[:])
```

```
output

'hell'
'ello'
'hello'
```

And for lists:

```
l = [1,2,3,4,5]
print(l[:4])
print(l[1:])
print(l[:])
```

```
output

[1,2,3,4]
[2,3,4,5]
[1,2,3,4,5]
```

Types: Lists — operators

Lists have exactly the same operators as strings:

```
+, +=, *, *=, [], ==, !=, <, <=, >, >=, in
```

Examples:

```
l = [1,2,3,4]
print(l + [5,6])
l += [5,6]
print(l)
print(3 in l)
print(l > [5,6])
print([5,6] * 3)
```

output

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 4, 5, 6]
True
False
[5, 6, 5, 6, 5, 6]
```

Data types have associated functionality

Each type has its own functionality.
For instance, it would be convenient if:

- strings had functionality that turns letters in string into capital letters
- lists had functionality that adds an element to the end of a list.

How is this done in Python? To explain this, I must briefly introduce the concepts of function and method.
We'll cover this in much more detail later in the course.

Interlude: functions and methods

A *function* is a piece of code that has a name, and can be *called* from anywhere in the program as:

```
functionname(argument-list)
```

Example:

```
len('hello') # function taking 1 argument
```

```
output
5
```

A *method* is similar to a function, but it belongs to an *object*:

```
object.methodname(argument-list)
```

```
l = 'hello'
print(l.replace('llo', 'y')) # method taking 2 arguments
```

```
output
'hey'
```

Note how methods make it easy to control which objects have which functionality: different types have different methods.

String methods

Commonly used string methods:

`capitalize`	capitalize string	`rstrip`	Remove traling whitespace
`lstrip`	removing leading whitespace	`join`	join list of strings
`center`	center justify	`split`	split into list of smaller strings
`replace`	replace substring	`ljust`	left justify
`count`	count substring occurrences	`strip`	`lstrip` and `rstrip`
`rjust`	right justify	`lower`	convert to lowercase
`find`	find index of substring	`upper`	convert to uppercase

Examples:

```
s = 'hello'
print(s.upper())
print('hello'.find('ll'))
print('hello'.replace('ll'
, 'ct'))
print('one two
three'.split(' '))
```

```
output

'HELLO'
2
'hecto'
['one', 'two', 'three']
# we'll use this one a
lot!
```

List methods

Commonly used list methods:

`append`	add element to the end of a list
`count`	Count occurrences of value
`index`	Return index of first occurrence of value
`insert`	Insert element at specified index
`pop`	Remove and return element from (end of) list
`remove`	remove first occurrence of specified value
`reverse`	reverse order of elements in list (in place)
`sort`	sort elements in list (in place)

Examples:

```
l =
[1,2,3,4,5,6]
l.append(7)
l.reverse()
print(l)
```

```
output


[7, 6, 5, 4, 3, 2,
1]
```

Lists — exercise

1. Create a variable containing the following string:

```
Horse sense is the thing a horse has which keeps it
from betting on people. W.C. Fields
```

2. Convert this string to a list of words.
3. Find the index of the word ”Horse” in the list.
4. Sort the list.
5. Print out the first 3 elements of this sorted list.
6. Bonus exercise: Try to reverse the list of words using the slice operation.

Lists — exercise — solution

1. Create a variable containing the string:

```
quote = '''Horse sense is the thing a horse has which
keeps it
... from betting on people. W.C. Fields'''
```

2. Convert this string to a list of words.

```
word_list = quote.split()
print(word_list)
```

```
output

['Horse', 'sense', 'is',
'the', 'thing', 'a',
'horse', 'has', 'which',
'keeps', 'it', 'from',
'betting', 'on',
'people.', 'W.C.',
'Fields']
```

3. Find the index of the word ”Horse” in the sequence.

```
print(word_list.index("Hor
se"))
```

```
output
0
```

Lists — exercise — solution (2)

4. Sort the list.

```
word_list.sort()
```

5. Print out the first 3 elements of this sorted list.

```
print(word_list[:3])
```

```
output

['Fields', 'Horse',
'W.C.']
```

6. Bonus exercise: Can you come up with a way to use slicing to reverse the order of the word list

```
print(word_list[::-1])
```

Basic looping

The while loop

A while loop repeats a piece of code for as long as its condition is true:

```
while condition:
     code block
```

```
i = 1
while (i < 4):
    print(i)
    i += 1

print("hello")
# i=4    (note change in output)
```

```
output
1
2
3
hello
```

Q: What would happen if we left out the `i += 1`?

A: It would loop forever. This is called an *infinite loop*

The for loop

Most of the time when writing a loop, it will be to iterate through a sequential container, like a list or a string. The for command makes this easy:

```
for variable in container-object:
     code block
```

```
container = ["a", "b", "c"]
for val in container:
    print(val)
```

```
output

a
b
c
```

Interrupting a loop

There are several ways to interrupt a loop from within:

- **break** - stop loop immediately
- **continue** - immediately start on the next iteration.

```
for i in [1,2,3,4]:
    if i==3:
        continue  # start on next iteration (skips
print)
    print(i)
```

```
output
1
2
4
```

What would happen if I wrote `break` instead of `continue`? `break` and `continue` work for both while and for loops.

Nice to know: the pass statement

Since Python relies on correct indentation, you have to write *something* inside an `if`-statement or loop
Sometimes, you just want to do nothing.
There is a statement that can be used as a placeholder, since it does literally nothing:

```
if x < 0:
    pass    # Haven't decided what to do here yet
elif x > 0:
    print("positive")
```

Loops - exercise

1. Create the following list: ["hello","how","are","you"].
2. Write a while loop that prints each element in the list.
3. Write a for loop that prints each element in the list.
4. Write a for loop that prints only words starting with the letter "h"

Loops - exercise - solution

1. Create the following list: ["hello","how","are","you"].

```
word_list = ["hello","how","are","you"]
```

2. Write a while loop that prints each element in the list.

```
i = 0
while i < len(word_list):
    print(word_list[i])
    i += 1
```

3. Write a for loop that prints each element in the list.

```
for entry in word_list:
    print(entry)
```

Loops - exercise - solution (2)

4. Write a loop that prints only words starting with the letter "h"

```
for word in word_list:
    if word[0] == "h"
        print(word)
```

Files

The file type

The file type is used to represent a file in a program
You can open a file using the open function:

```
open(filename, mode)
```

where mode can be:

```
'r'     - read
'w'     - write
'r+w'   - read and write
'a'     - append
```

The open function returns a File object. Remember to save this object to a variable.

```
input = open('hello.txt', 'r')
output = open('outputfile.txt', 'w')
```

Files - methods

Commonly used methods:

`readline`	Read one line as a string
`readlines`	Read file as a list of strings
`write`	Write string to file
`writelines`	Write list to file (as lines)
`close`	Close file

Example:

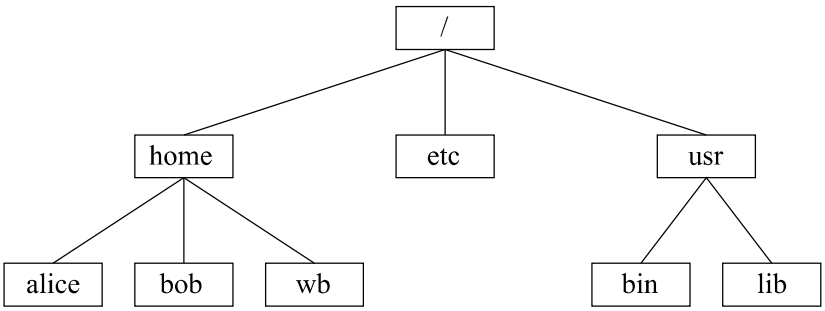
```
input = open('/home/wb/hello.txt', 'r')
lines = input.readlines()
first_line = lines[0]
input.close()
```

The `first_line` variable is now a string with whatever text was in the first line of the file - **including a new line at the end**, e.g. `"Hello, world!\n"`.

Remember, you can remove the newline with the `strip()` method.

Btw, what does `/home/wb/hello.txt` mean?

Files and Directories

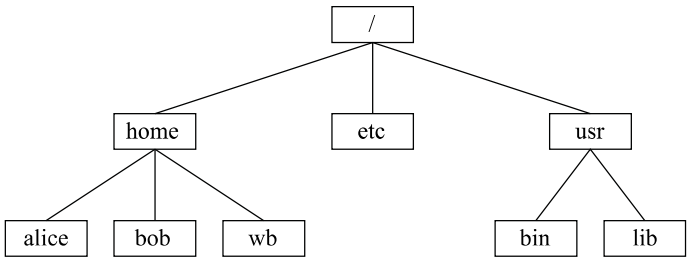


- Files are organized similar to Windows, OSX, ...
- The *current directory* is where you are in the tree right now.
- The root directory is the directory at the highest level of the file system hierarchy.

  Unix/Linux: `/`

  Windows: Name of drive - e.g. `C:\`

Absolute and Relative paths



Directories (folders) and files can be referred to by using either an **absolute** or a **relative** path:

absolute:

`/home/wb/exam_results.html` (Unix)

`C:\Users\wb\exam_results.html` (Windows)

relative: (assuming I am in `/home`)

`wb/exam_results.html` (Unix)

`wb\exam_results.html` (Windows)

Types - Files are streams

Important: you can only read a file once.
The file is a stream, you *consume* the stream when you read it

```
input_file = open('hello.txt', 'r')
file_as_str = input_file.read()
file_as_str = input_file.read()           # file_as_str
will now be empty
```

There are ways to reset a file stream, so you can read it again...
...but the simplest (and most efficient) is to just read once, and save the information that you need into a variable.

Files — exercise

1. In VS Code, create a new file (not a python file, just a normal text file), called `hello.txt`, and write 5 lines of text. A single word on each line is enough.
2. Now, create a new program called `file_test.py`. In this program, use the `open` function to open the `hello.txt` file.
3. Read its content into a list of strings
4. Print the last two lines to screen

Files — exercise — solution

1. Create a new program called `file_test.py`. In this program, use the `open` function to open the `hello.txt` file.

```
file_test.py
my_file = open("hello.txt")
```

2. Read its content into a list of strings

```
file_test.py
lines = my_file.readlines()
```

3. Print the last two lines to screen

```
file_test.py
print(lines[-2:])
```

Speaker notes