Lecture 2: Python Introduction

Goals for Today:

Python:

- Data Structures
- Control Statements
- Comprehensions

Data Structures

In any practical application it will quickly become inconvenient to work with objects on an individual basis and we will want to collect objects together--that's what **data structures** are for:

• **List**: lists are the structures that store data in **order**. Lists have built-in methods for adding more data, removing data, modifying data, and accessing data in the list. You can modify objects in the list without creating a whole new list, for this reason we say that lists are **mutable**. Defined using `[]` notation.

• **Tuple**: tuples are structures that store data in **order** but are **immutable**. That is, tuples are like lists, except that you can't add or remove from them without creating a whole new object in memory. Sometimes you need things that are immutable (e.g., the next two structures require immutable objects!). Defined using `()` notation.

• **Set**: sets are like lists but they are **unordered** and no duplicates are allowed. Unordered? Yes, sets collect objects but don't keep track of where they were put. On a technical side, it is actually more accurate to say that sets are hyperspecific about where they've put things (internally) and so you as the user cannot modify that order. This makes sets very efficient at calculating membership (e.g., is "1" in our set?) and intersections (common objects in multiple sets). Defined using `{}` or `set(list)` notation.

• **Dictionary**: dictionaries are structures that store objects in **pairs**. Each pair consists of a **key** and a **value**. In analogy, a phone book is a dictionary, where the key is a person's name, and the value their phone number. You could accomplish the same thing using two ordered lists, but dictionaries have the advantage that their keys are stored...as...a...**set**! Defined with `{}` notation with `key:value` pairs.

Data Structures: List Examples

Lists can be defined using `[]` notation:

```
# example 1 - creating a list of ints
a = [1,2,3,4]
print("a:"+str(a))
```

```
a:[1, 2, 3, 4]
```

You can add elements to existing lists using the + operator:

```
# example 2 - adding an element and creating a new list
b = a + [100]
print("b:"+str(b))
```

```
b:[1, 2, 3, 4, 100]
```

You can remove elements from a list using the `.pop()` method or `del`*:

```
# example 3 - removing an element from the list and storing it
 c = b.pop(0)
print("c:"+str(c)+", b:"+str(b))
```

```
c:1, b:[2, 3, 4, 100]
```

*the `del` keyword does not return the deleted object (e.g., `del b[0]`)

Data Structures: List Indexing

Objects within Lists can be accessed using slice [start:end:stride] syntax*:

```
# example 4 - grabbing the first item in a list (without removal)
e = a[0]
print("e:"+str(e))
```

```
e:1
```

(-) indexing counts from the end:

```
# example 5 - grabbing the last object (without removal)
f = a[-1]
print("f:"+str(f))
```

```
f:4
```

Multiple objects can be returned (start=0,end=-1 by default):

```
# example 6 - grabbing every other object without removal)
j = a[::2]
print("j:"+str(j))
```

```
j:[1, 3]
```

*Python is a 0-indexed language (first object is index 0 in ordered structures).

*start=0, end=start+1 and stride=1 by default. Inclusive of start, exclusive of end

Data Structures: Slice Notation

Since it is so important, here's a quick summary of slice notation:

- `list[start:stop:step]` returns a list of objects between the indices `start` through `stop` (excluding stop).
- `list[start:]` returns objects `start` through the end of the list.
- `list[:stop]` returns objects from the beginning of the list through the object at index `stop` (*excluding the object at* `stop`)
- `list[:]` returns all objects
- `list[::step]` returns objects every step apart.

You can use slice notation for accessing objects from any ordered data structure (e.g., tuple).

Since lists are **mutable** you can also reassign objects using slice notation:

```
# example 11 - reassign the first element to 100000.0
a[0] = 100000.0
print("a:"+str(a))
```

```
a:[100000.0, 2, 3, 4]
```

Data Structures: Tuple Examples

Tuples can be defined using `()` notation:

```
# example 1 - creating a tuple
a = (1,2,3,4)
print("a:"+str(a))
```

```
a:(1, 2, 3, 4)
```

You can combine tuples using the + operator:

```
# example 2 - extending a tuple
b = a + a
print("b:"+str(b))
```

```
b:(1, 2, 3, 4, 1, 2, 3, 4)
```

You can use slice notation on tuples (because they are ordered):

```
# example 4 - returning the last two objects
d = a[-2:]
print("d:"+str(d))
```

```
d:(3, 4)
```

*the `del` keyword does not return the deleted object (e.g., `del b[0]`)

Data Structures: Tuple Examples

You **cannot** modify elements of tuples individually (**immutable**):

```
# example 5 - try to reassign the first element to 100000.0
a[0] = 100000.0
```

```
TypeError
```

Python also has built-in functions like `min()` and `max()` that find the minimum and maximum values in data structures*:

```
# example 6 - finding the min of a tuple
a = (1,2,3,4)
b = min(a)
print(”b:"+str(b))
```

```
b:1
```

```
# example 7 - finding the min of a list
a = [1,2,3,4]
b = min(a)
print(”b:"+str(b))
```

```
b:1
```

*these functions work on sets and dictionary keys as well.

Data Structures: Set Examples

Sets can be defined using `{ }` notation:

```
# example 1 - creating a set
a = {1,2,3,4}
print("a:"+str(a))
```

```
a:{1, 2, 3, 4}
```

Sets are unordered (which also implies no duplications):

```
# example 2 - sets do not have order or duplicates
b = {4,4,4,4,3,2,1}
print("a==b: "+str(a==b))
```

```
a==b: True
```

You can add to sets using the `set.add()` method:

```
# example 3 - extending a set
a.add(5.0)
print("a:"+str(a))
```

```
a:{1, 2, 3, 4, 5.0}
```

Data Structures: Set Examples

The `in` operator can be used to determine if a value is in a data structure*:

```
# example 5 - checking if a number is in the set
print("2 in a: "+str(2 in a))
```

```
2 in a: True
```

Sets also have specialized methods for determining logical overlaps:

```
# example 6 - finding the union of two sets
b = {3,4,5,6,7}
print("b:"+str(b))
print("union of a and b:"+str(a.union(b)))
print("intersection of a and b:"+str(a.intersection(b)))
```

```
b:{3, 4, 5, 6, 7}
union of a and b:{1, 2, 3, 4, 5.0, 6, 7}
intersection of a and b:{3, 4, 5}
```

Notice the example of upcasting (`5=5.0`) for the comparison.

*`in` works on lists, tuples, and dictionary keys as well. More efficient on sets and dictionaries.

Data Structures: Dictionary Examples

Dictionaries can be defined using `{key:value}` notation:

```
# example 1 - creating a dictionary
a = {1:"one",2:"two",3:"three",4:"four"}
print("a:"+str(a))
```

```
a:{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
```

You can access values in a dictionary by key

```
# example 2 - accessing a dictionary value by key
print("a[3]: "+a[3])
```

```
a[3]: three
```

You can add to dictionaries using `dictionary[key] = value` notation:

```
# example 3 - adding a key:value pair to a dictionary
a[99] = "ninety-nine"
print("a:"+str(a))
```

```
a:{1: 'one', 2: 'two', 3: 'three', 4: 'four', 99: 'ninety-
nine'}
```

Data Structures: Dictionary Examples

Dictionaries can also hold other dictionaries

```
# example 4 - dictionary of dictionaries
a = {"benzene": {"MW": 78.11, "MF":"C6H6"},\
     "ethane": {"MW":30.07, "MF":"C2H6"}}
print(a["benzene"]["MW"])
```

```
78.11
```

All of the structures shown so far can hold other structures (not just integers, floats, and strings).

- Sequential `[][]…[]` notation can be used to call nested objects:

```
# example 5 - list of lists
a = [[1,2,3],[4,5,6],[7,8,9]]
print(a[1][2])
```

```
6
```

- Nested lists of lists are rudimentary tensors.

- We've mentioned **methods** several times (e.g., `list.add()`). We'll come back to this when we discuss classes. Here just think of them as functions attached to structures.

Control Statements

We motivated the introduction of data structures by saying that it is difficult to do anything useful if you are limited to dealing with individual data.

Similarly, it is difficult to construct useful programs if you have to hard code every individual action you want to do. We will often want to:

- perform the same action many times (**for/while** loops)
- perform an action only under certain circumstances (**if/elif/else** logic)
- attempt something and if it fails implement a fallback plan (**try/except** logic)
- some combination of all three.

The tools that accomplish these feats are **control statements**. These are built into python, and they are so useful that every language has them in some form. We'll quickly run through the definitions of each of these then demonstrate their usefulness with examples.

Control Statements: `for` Loops

You will use `for` whenever you need to do something "*for every* `x` *in* `y`", where `x` is an object and `y` is something like a list that potentially holds many objects.

The standard construction of a `for` loop is:

```
for variable in iterable:
---->indented commands…
```

where `variable` is a convenient name for the objects returned by the **iterable**, and `iterable` is the **pointer** to an iterable object or an iterable defined directly in the for statement.

```
# example 1: loop over list objects
for i in [1,2,3]:
    print(i)
```

```
1
2
3
```

you can use `for` on any object that is **iterable**, which means the object has rules for what to do when for operates on it.

Control Statements: `for` Loops

many built-in python objects are **iterable**, including strings (`for` loops over their individual characters), although their iteration rules might not be obvious without experimentation.

```
# example 2: loop over list object using pointer
a = [1,2,3]
for dummy in a:
    print(dummy)
```

```
1
2
3
```

Looping over the pointer to an iterable is the same as looping directly over the iterable. You can also use slicing:

```
# example 3: loop over partial list
a = [1,2,3,4,5]
for i in a[::2]:
    print(i)
```

```
1
3
5
```

Control Statements: `for` Loops

You can modify objects within for loops:

```
# example 4: modify objects within loop (cumulative summation)
a = [1,2,3,4,5]
cum_sum = [a[0]]
for i in a[1:]:
    cum_sum = cum_sum + [i+cum_sum[-1]]
print(cum_sum)
```

```
[1, 3, 6, 10, 15]
```

You can also break out of loops using the keyword `break`:

```
# example 5: break out of loop early (also if/else use!)
a = [1,2,3,None,5,6,7]
for i in a:
    if i is None:
        break
    else:
        print(i)
```

```
1
2
3
```

Control Statements: `while` Loops

You will use `while` whenever you need to do something repeatedly "while x is True”.

The standard construction of a `while` loop is:

```
while condition:
---->indented commands…
```

where `condition` is an expression that evaluates to a **boolean**. As long as `condition` evaluates to `True`, the loop will run again.

```
# example 1: increment an integer
i=1
while i < 4:
    print(i)
    i += 1
```

```
1
2
3
```

`while` loops and `for` loops are interchangeable for most tasks, but some things are easier with one or the other.

Control Statements: `while` Loops

Looping over objects is more cumbersome with `while` loops:

```
# example 2: loop over list object using pointer
a = [1,2,3]
i=0
max=len(a)
while i<max:
    print(a[i])
    i += 1
```

```
1
2
3
```

`while` can be useful when objects change length in the loop:

```
# example 3: pop items from list
a = [1,2,3]
while a:
    print(a.pop(0))
```

```
1
2
3
```

Note: iterables (e.g., `a` above) evaluate to `True` when non-empty.

Control Statements: `while` Loops

`while` can also be useful when the termination condition isn't obvious:

```
# example 4: modify objects within loop (multiples of 2)
print("\nexample 4:")
a = [1]
while a[-1] < 100:
    a += [a[-1]*2]
print(a[:-1])
```

```
[1, 2, 4, 8, 16, 32, 64]
```

`while` can be useful when you need infinite loops:

```
# example 5: break out of loop early (also if/else use!)
a = [1,2,3,None,5,6,7]
i = 0
while True:
    if a[i] is None:
        break
    else:
        print(a[i])
    i += 1
```

```
1
2
3
```

Note: `True` is always `True.` Make sure you include a break condition!

Control Statements: `if/elif/else` cases

You will use `if` (and possibly `elif` and `else`) when you want to do something "only if x is True", where x is a condition (e.g., `x is False`) and True is a boolean.

The standard construction of an `if` statement is:

```
if condition1:
---->indented commands…
elif condition2:
---->indented commands…
elif condition3:
…
else:
---->indented commands…
```

where `conditionX` are expressions that evaluate to a boolean, `elif` means “else if”, and `else` has no condition.

`elif` and `else` statements are optional. `else` is always last if it is used.

`else` always runs if the preceding conditions are `False`.

Each condition is evaluated sequentially and only the first condition that evaluates to `True` is executed.

Control Statements: `if/elif/else` cases

`if/else` can be used with other control statements:

```
# example 1: if else, find smallest divisor of a besides 1.
a = 1890843693043
i = 2
while i < a:
    if a % i == 0:
        break
    else:
        i += 1
print("{} is the smallest divisor of {} greater than 1.".forma
t(i,a))
```

```
8641 is the smallest divisor of 1890843693043 greater than 1.
```

`if` is evaluated first, `elif` second, `else` last:

```
# example 3: if elif else
a = 2.1
if a % 2 == 0:
    print("a is even")
elif a % 1 == 0:
    print("a is odd")
else:
    print("a is fractional")
```

```
a is fractional
```

Control Statements: `try/except` cases

You will use `try/except` when you want to do something that might lead to an error.

The standard construction of a `try/except` statement is:

```
try:
---->indented commands…
except error1:
---->indented commands…
except error2:
…
except:
---->indented commands…
```

where `errorX` are specific errors that might be encountered while evaluating the try block (e.g., `TypeError`). At least one except must be present.

"***It is better to ask forgiveness than permission***" is a common Python axiom, meaning that it is sometimes faster to try something than to check in advance that is works.

`except:` without an error listed will always get run when it is encountered.

`finally:` can be used at the end to run something afterward in all cases.

Control Statements: `try/except` cases

`try` can be used without specific exceptions:

```
#Example 1: try with a general except condition
try:
    print(doesnt_exist)
except:
    print("that didn't work!")
```

```
That didn't work!
```

`try` can be used with specific exceptions:

```
#Example 2: try with a specific except condition
try:
    print(doesnt_exist)
except NameError:
    print("Caught a NameError")
```

```
Caught a NameError
```

`pass` can be used to do nothing in the except block:

```
#Example 3: try with a do nothing exception
try:
    print(doesnt_exist)
except:
    pass
```

```

```

Control Statements: `try/except` cases

`try` can be used with both specific and non-specific exceptions:

```
# Example 4: try with a specific exception and general excepti
on
print("\nExample 4:")
a="string"
try:
    a = a/2
except TypeError:
    print("'a' doesn't support division")
except:
    print("something besides a type error occurred")
```

```
'a' doesn't support division
```

`try/except` is logically equivalent to `if/else` except you don't test the condition in advance. Thus, you can get by without using it, but it is an asset of the language (e.g., one of your homework problems can be done very simply with it).

`finally:` and `else:` can be used at the end of try/except chains to excecute commands regardless of which try/except case was run.

Structure Comprehensions

`For` and `if/else` loops are used so often with data structures that a shorthand syntax called **list/set/dictionary comprehension** exists for implementing them in a single line.

The basic syntax for a list comprehension is:

```
[ f(variable) for variable in iterable ]
```

where `variable` is a name for the objects returned in each loop of the iterable (like a `for` loop), `iterable` is any object with rules for iteration, and `f(variable)` is the statement for what you want to do with the `variable`.

It is easier to show than to tell:

```
a = list(range(10)) # Note: range is a built in function for
                    #       generating lists of integers
b = [ i*10 for i in a ]
print(b)
```

```
[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

We've looped over a list `[0,1,2,…,9]` and multiplied each element by `10`.

Comprehensions for sets and dictionaries are similar but we use `{}` instead of `[]`

Structure Comprehensions: List examples

`if/else` logic can be applied to each element during the iteration:

```
a = list(range(10))
c = [ i*10 for i in a if i % 2 == 0 ]
print(c)
```

```
[0, 20, 40, 60, 80]
```

`if/else` logic can be applied to the return statement also:

```
a = list(range(10))
d = [ i*10 if i % 2 == 0 else i for i in a ]
print(d)
```

```
[0, 1, 20, 3, 40, 5, 60, 7, 80, 9]
```

`for` loops can be nested:

```
a = [[1,2],[3,4]]
f = [ j for i in a for j in i ]
```

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

Comprehensions for sets and dictionaries are similar but we use `{}` instead of `[]`

Structure Comprehensions: Set/Dict examples

Set comprehensions use `{}` instead of `[]`:

```
a = { i/10.0 for i in range(10) if i % 2 == 0 }
print(a)
```

```
{0.0, 0.4, 0.6, 0.8, 0.2}
```

Dictionary comprehensions use `{}` and `key:value` pairs:

```
b = { str(i):i for i in range(6) }
```

```
{'0': 0, '1': 1, '2': 2, '3': 3, '4': 4, '5': 5}
```

Tuple comprehensions use `tuple()` :

```
c = tuple( i/10.0 for i in range(10) if i % 2 == 0 )
print(c)
```

```
(0.0, 0.2, 0.4, 0.6, 0.8)
```

`()` is already reserved for syntactically for functions.