EEB 177 Lecture 8

Thursday Feb 2nd, 2017

Topics

- Collections
- Flow statements

Office Hours

Tues 11-12

Wednesday 10-11

Hershey Courtyard (inside Hershey Hall)

Terasaki 2149 on rainy days

Hacking Sessions

Tuesday from 6-8:30 in LS 3209

Homework #4

Will be assigned Friday.

Due next Wednesday by 5:00 PM

Projects Deadline #1

Due Friday by 5:00 PM. To complete this assignment:

- create your eeb-174-final-project repo on github
- add a file called project-idea.txt that describes your project idea including:
    - conceptual basis of the project (I am interested in the effect of the KT extinction on fossil mammals...)
    - discussion of data and computational challenges (I plan to use occurrence data from the PBDB to reconstruct fossil ranges...I will need to extract fossil age and locality data, format it for quantitative analyses, and visualize the results...
    - If you are using the PBDB, you must also report the # of unique species for 3 groups. Rank these groups in order of preference and show the shell commands you used to extract unique names.

Preliminaries

- Create a new jupyter notebook and save the file "classwork-Thursday-2-1.ipynb" to your class-assignments directory
- push this to your remote repository
- you can write answers to today's exercises in this file.

The modulus operator `%` returns the remainder of an integer division

```
In [17]: 15 % 7
Out[17]: 1
In [20]: 13 % 5
Out[20]: 3
```

String challenge

Do the following

1. Initialize the string s = "WHEN on board H.M.S. Beagle, as naturalist".
2. Apply a string method to count the number of occurrences of the character b.
3. Modify the command such that it counts both lower and upper case bs.
4. Replace WHEN with When.

Collections

Python has variables that are collections of other objects. lists are collections of ordered values and are defined by `[]`.

```
# Anything starting with # is a comment
In [26]: MyList = [3,2.44,"green",True]
In [27]: MyList[1]
Out[27]: 2.44
In [28]: MyList[0] # NOTE: FIRST ELEMENT -> 0
Out[28]: 3
In [29]: MyList[3]
Out[29]: True
In [30]: MyList[4]
IndexError: list index out of range
In [31]: MyList[2] = "blue"
In [32]: MyList
Out[32]: [3, 2.44, "blue", True]
```

note that indexing in python starts at 0.

slicing

We can subset or take a slice of a list using the `:` operator.

```
In [10]: my_list
Out[10]: ['blue', 2.44, 'green', True]
In [11]: my_list[0:1] # element 0 to 1 (non-inclusive) Ou
In [12]: my_list[1:3] # element 1 to 3 (non-inclusive) Ou
In [13]: my_list[:] # take the whole list
Out[13]: ['blue', 2.44, 'green', True]
In [14]: my_list[:3] # from start to element 3 (non-inclu
In [15]: my_list[3:] # from element 3 to the end
Out[15]: [True]
```

If you don't specify the first index position, `:` returns all of the indices from the start to the specified point. If you specify the first position but not last, `:` returns elements up to but not including that position.

indexing from the end

You can use negative numbers to grab elements from the end of a list.

```
In [16]: my_list[-2]
Out[16]: 'green'
```

copy

```
In[18]: my_list = ['blue', 2.44, 'green', True, 25]
In [19]: new_list = my_list.copy()
In [19]:new_list
Out[20]: ['blue', 2.44, 'green', True, 25]
```

clear

removes all elements from a list

```
In [21]: my_list.clear()
In [22]: my_list
Out[22]: []
```

pop

remove the last element of the list and return it.

```
In [23]: seq = list("TKAAVVNFT")
In [26]: seq2 = seq.pop()
In [27]: seq
Out[27]: [’T’, ’K’, ’A’, ’A’, ’V’, ’V’, ’N’, ’F’]
In [28]: seq2
Out[28]: ’T’
```

Subsetting lists

You can make a new list by telling python the elements of the original list to include

```
ranks = ["kingdom","phylum", "class", "order", "family"]
In [81]: ranks
Out[81]: ['kingdom', 'phylum', 'class', 'order', 'family'
lower_ranks = ranks[2:5]
In [83]: lower_ranks
Out[83]: ['class', 'order', 'family']
```

Note how python indexing works:numbers are inclusive at the start and exclusive at the end.

Appending to lists

Try this

```
apes = ["Homo sapiens", "Pan troglodytes",
"Gorilla gorilla"]
print("There are " + str(len(apes)) + " apes")
apes.append("Pan paniscus")
print("Now there are " + str(len(apes)) + " apes")
```

append() changes the variable in place!

+ and extend

Try this

```
apes = ["Homo sapiens", "Pan troglodytes",
"Gorilla gorilla"]
monkeys = ["Papio ursinus", "Macaca mulatta"]
primates = apes + monkeys
print(str(len(apes)) + " apes")
print(str(len(monkeys)) + " monkeys")
print(str(len(primates)) + " primates")

platyrrhines = ["Cebus", "Sapajus", "Aotus"]
monkeys.extend(platyrrhines)
print monkeys
['Papio ursinus', 'Macaca mulatta',
'Cebus', 'Sapajus', 'Aotus']
```

More list operations

Try these

```
In [71]: apes = ["Homo sapiens", "Pan troglodytes",
"Gorilla gorilla"]
In [72]: #how long is list?
In [73]: len(apes)
Out[73]: 3
In [74]: #index elements with brackets
In [75]: apes[1] # what element will this be
Out[75]: 'Pan troglodytes'
In [76]: apes.reverse() #lists are ordered
In [77]: apes[0]
Out[77]: 'Gorilla gorilla'
#if you know the element
but do not know position, use index
In [78]: apes.index("Homo sapiens")
Out[78]: 2
#get the last element of any list with [-1]
In [79]: apes[-1]
Out[79]: 'Homo sapiens'
```

Dictionaries

Dictionaries are collections of unordered lists where elements (*values*) are accessed by *keys*. We separate *key-value* pairs using a comma.Dictionaries are defined using `{}`

```
# create an empty dictionary
In [1]: my_dict = {}
# strings, floats or even lists can be values in a dictio
4]}
In [3]: my_dict
Out[3]: {'a': 'test', 'b': 3.14, 'c': [1, 2, 3, 4]}
In [4]: GenomeSize = {"Homo sapiens": 3200.0, "Escherichi
coli": 4.6, "Arabidopsis thaliana": 157.0}
# A dictionary has no natural order (i.e., the order of k
value input does not matter)
In [5]: GenomeSize
Out[5]:
{'Arabidopsis thaliana': 157.0,
'Escherichia coli': 4.6,
'Homo sapiens': 3200.0}
```

You can access the value of a dictionary by suppling the key for that pair.

```
In [3]: GenomeSize["Arabidopsis thaliana"]
Out[3]: 157.0
```

You can also add entries by supplying a new key-value pair.

```
In [4]: GenomeSize["Saccharomyces cerevisiae"] = 12.1
In [5]: GenomeSize
Out[5]:
{"Arabidopsis thaliana": 157.0,
 "Escherichia coli": 4.6,
 "Homo sapiens": 3200.0,
 "Saccharomyces cerevisiae": 12.1}
```

Adding keys and changing values in dictionaries

```
# ALREADY IN DICTIONARY!
In [6]: GenomeSize["Escherichia coli"] = 4.6
In [7]: GenomeSize
Out[7]:
{"Arabidopsis thaliana": 157.0,
 "Escherichia coli": 4.6,
 "Homo sapiens": 3200.0,
 "Saccharomyces cerevisiae": 12.1}
In [8]: GenomeSize["Homo sapiens"] = 3201.1
In [9]: GenomeSize
Out[9]:
{"Arabidopsis thaliana": 157.0,
 "Escherichia coli": 4.6,
 "Homo sapiens": 3201.1,
 "Saccharomyces cerevisiae": 12.1}
```

copy

```
In [13]: GS = GenomeSize.copy()
In [14]: GS
Out[14]:
{'Arabidopsis thaliana': 157.0, 'Escherichia coli': 4.6,
'Homo sapiens': 3201.1, 'Saccharomyces cerevisiae': 12.1}
```

clear

removes all elements

```
In [15]: GenomeSize.clear()
In [16]: GenomeSize
Out[16]: {}
```

get

gets a value from key

```
In [67]: GenomeSize.get("Homo sapiens")
Out[67]: 3200.0
```

this function is very useful for initializing the dictionary, or to return a special value when the key is not present.

```
In [68]: GenomeSize.get("Mus musculus", 10)
Out[68]: 10
```

items

returns key value pairs. Can be used to print contents of a dictionary.

```
for k,v in GS.items():
    print(k, v)
('Homo sapiens', 3200.0)
('Escherichia coli', 4.6)
('Arabidopsis thaliana', 157.0)
```

keys, values

These functions return lists of the keys or values of the dictionary.

```
In [72]: GS.keys()

Out[72]: ['Homo sapiens', 'Escherichia coli', 'Arabidopsi

In [74]: GS.values()

Out[74]: [3200.0, 4.6, 157.0]
```

Creating dictionaries

You will often create a dictionary and then populate it. Try this!

```python
enzymes = {}
enzymes['EcoRI'] = r'GAATTC' # r before the string
#tells python to automatically escape every character
enzymes['AvaII'] =  r'GG(A|T)CC'
enzymes['BisI'] =  r'GC[ATGC]GC'
enzymes.keys()
enzymes.values()
```

You can use zip() to turn two lists into a dictionary

```
keys = ('name', 'age', 'food')
values = ('Monty', 42, 'spam')
zip(keys, values) #makes a list of tuples
my_new_dict = dict(zip(keys, values))
my_new_dict
```

tuples contain sequences that are immutable and are defined by ().

```
In [12]: FoodWeb=[("a","b"),("a","c"),("b","c"),("c","c")
In [13]: FoodWeb[0]
Out[13]: ("a", "b")
In [14]: FoodWeb[0][0]
Out[14]: "a"
# Note that tuples are "immutable"
In [15]: FoodWeb[0][0] = "bbb"
TypeError: "tuple" object does not support item assignmen
In [16]: FoodWeb[0] = ("bbb","ccc")
In [17]: FoodWeb[0]
Out[17]: ("bbb", "ccc")
```

Use tuples when order matters.

sets are lists without duplicate elements

```
In [1]: a = [5,6,7,7,7,8,9,9]
In [2]: b = set(a)
In [3]: b
Out[3]: set([8, 9, 5, 6, 7])
In [4]: c=set([3,4,5,6])
In [5]: b & c
Out[5]: set([5, 6])
In [6]: b | c
Out[6]: set([3, 4, 5, 6, 7, 8, 9])
In [7]: b ^ c
Out[7]: set([3, 4, 7, 8, 9])
```

The operations are: Union | (or); Intersection & (and); symmetric difference (elements in set b but not in c and in c but not in b), ˆ; and so forth.

You can concatenate similar collection elements with +

```
In [1]: a = [1, 2, 3]
In [2]: b = [4, 5]
In [3]: a + b
Out[3]: [1, 2, 3, 4, 5]
In [4]: a = (1, 2)
In [5]: b = (4, 6)
In [6]: a + b
Out[6]: (1, 2, 4, 6)
In [7]: z1 = {1: "AAA", 2: "BBB"}
In [8]: z2 = {3: "CCC", 4: "DDD"}
In [9]: z1 + z2
     ---------------------------------------------
     TypeError  Traceback (most recent call last)
     ----> 1 z1 + z2
TypeError: unsupported operand type(s) for +: "dict" and '
```

collections recap

```
# round brackets --> tuple
In [1]: type((1, 2))
Out[1]: tuple
# square brackets --> list
In [2]: type([1, 2])
Out[2]: list
# curly brackets, sequence of values --> set
In [3]: type({1, 2})
Out[3]: set
# curly brackets, key-value pairs --> dictionary
In [4]: type({1: "a", 2: "b"})
Out[4]: dict
```

Challenge

Do this:

- Define a list a = [1, 1, 2, 3, 5, 8].
- Extract [5, 8] in two different ways.
- Add the element 13 at the end of the list.
- Reverse the list.
- Define a dictionary m = {"a": ".-", "b": "-...-", "c": ‘-.-.’}.
- Add the element "d": "-..".
- Update the value "b": "-...".

program flow in python

In its simplest form, a program is just a series of instructions (statements) that the computer executes one after the other. In Python, each statement occupies one line (i.e., it is terminated by a newline character). Other programming languages use special characters to terminate statements (e.g., ; is used in C).

Lets demonstrate statements that control flow in pyton with a simple program.

Python programming best practices

We will try to instill as many standard practices as we can at the beginning.

- Wrap lines so that they are less than 80 characters long. You can use parentheses () or signal that the line continues using a "backslash" .
- Use 4 spaces for indentation, no tabs.
- Separate functions using a blank line.
- When possible, write comments on separate lines.
- Use docstrings to document how to use the code, and comments to explain why and how the
code works.

best practices continued

- Naming conventions:
- _internal_global_variable
- a_variable
- SOME_CONSTANT
- a_function
- Never call a variable l 1 or O or o
- use two or more letters for variables: xx better than x
- Use spaces around operators and after commas:
  a = func(x, y) + other(3, 4).