EEB 177 Lecture 4

Topics

- Python Overview
- Strings
- Collections

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 #3

Due this Wednesday by 5:00 PM

Projects

What can I do for my project?

Any biology-grounded topic that requires you to use programming tools to solve a problem can work. Projects must....

- perform operations in the shell
- use python, R, or shell to manipulate text data
- use python to read/write text files
- require three or more novel functions
- include a visualization of data in R
- include a final report in markdown or latex with commented code and embedded figures
- show a history of version control.

Project Suggestions

If you are in a research lab, talk to your PIs.

If you need a suggestion...see the handout I sent around.

If you are thinking of pursuing this idea for the Friday deadline, download occurrence data for three (or more) groups (we can't all work on dinosaurs...) and use your `tail`, `cut`, `sort` and `uniq` commands to find the number of unique species in the group. Write a text file called "paleo-project-idea.txt" and rank your preference for the three groups. Report the shell command and the result (species count) for each group and commit this to a new repo called eeb-174-final-project.

Project ideas are due Friday, Feb 3rd.

Preliminaries

- Create a new jupyter notebook and save the file "classwork-Tuesday-1-31.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
```

Variables

Variables allow you to assign a name to a value or object that is stored in memory. Once you create the variable you can use it in operations.

```
In [10]: x
Out[10]: "The cell grew"
In [11]: y # this is an integer Out[11]: 8
In [12]: x + " " + str(y) + " nm"
Out[12]: 'The cell grew 8 nm'
In [13]: z = "88" # this is a string In [14]: x + z
Out[15]: 'The cell grew88'
In [16]: y + int(z)
Out[16]: 96
```

dynamic typing

Programming languages like `C` and `Fortran` are statically typed, meaning that you need to define the type of a variable when you create it. Python does not require this and automatically determines type. You can see the type of a variable with the `type` function.

```
In [22]: xx = 2

In [23]: type(xx)
Out[23]: int

In [24]: xx = "two"

In [25]: type  (xx)
Out[25]: str
```

strings

Python is an excellent language for bioinformatics in part because it provides many built-in functions for manipulating strings. You can see these methods by typing the name of a string followed by a period and then TAB.

```
In [27]: xx.
xx.capitalize  xx.format      xx.isupper     xx.rindex
xx.center      xx.index       xx.join        xx.rjust
xx.count       xx.isalnum     xx.ljust       xx.rpartitio
```

Remember you can get help on any function with `help`

```
In [4]: help(xx.center)
Help on built-in function center: [...]
```

here are some string functions

```
# replace characters
 In [5]: astring.replace("T", "U")
 Out[5]: ’AUGCAUG’
 # position of first occurrence
 In [6]: astring.find("C")
 Out[6]: 3
 # count occurrences
 In [7]: astring.count("G")
 Out[7]: 2
 In [8]: newstring = " Mus musculus "
 # split the string (using spaces by default)
 In [9]: newstring.split()
 Out[9]: [’ Mus’, ’musculus ’]
 # specify how to split
 In [10]: newstring.split("u")
 Out10]: [’ M’, ’s m’, ’sc’, ’l’, ’s ’]
 # remove leading and trailing white space
 In [11]: newstring.strip()
 Out[11]: ’Mus musculus’
 In [1]: astring = "ATGCATG"
# return the length of the string
In [2]: len(astring)
Out[2]: 7
```

You can also use string functions by creating the string on the fly with quotation marks and calling method from the new string.

```
# make upper case
In [12]: "atgc".upper()
Out[12]: ’ATGC’
# make lower case
In[13]: "TGCA".lower()
Out[13]: ’tgca’
```

concatenating strings with `+` and `join`

```
In [14]: genus = "Rattus"
In [14]: species =
"norvegicus"
In [16]: genus + " " + species
Out[16]: ’Rattus norvegicus’

# join requires a list of strings as input;
#more on lists below
In [17]: human = ["Homo", "sapiens" , "sapiens"]
In [18]: " ".join(human)
Out[18]: ’Homo sapiens sapiens’
# specify any symbol as delimiter
```

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.

append, sort, and count are methods that work on lists

```
In [33]: MyList[0] = "blue"
In [34]: MyList
Out[34]: ["blue", 2.44, "blue", True]
In [35]: MyList.append("a new item")
In [36]: MyList
Out[36]: ["blue", 2.44, "blue", True, "a new item"]
In [37]: MyList.sort()
In [38]: MyList
Out[38]: [True, 2.44, "a new item", "blue", "blue"]
In [39]: MyList.count("blue")
Out[39]: 2
```

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'
```

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']
```

Dictionaries are collections of key-value pairs. Very useful for data without a natural order. Defined by `{}`.

```
In [1]: GenomeSize={"Homo sapiens": 3200.0,
  "Escherichia coli": 4.6, "Arabidopsis thaliana": 157.0}
In [2]: GenomeSize
Out[2]:
{"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
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}
```

Creating dictionaries

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

```
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 '
```

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": "-...".

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
- Nevercallavariablel 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).