EEB 177 Lecture 7

Michael Alfaro

EEB 177 Lecture 7

Topics

- Passing arguments to shell scripts
- Python

Preliminaries

- Start **nano**: `$ nano` and save the file "classwork-Thursday-1-30.txt" to your class-assignments directory
- push this to your remote repository
- you can write answers to today's exercises in this file.

Your Previous Script

```
#!/bin/bash
ls -la
echo "Above are the directory listings for this folder"
pwd
echo "right now it is :"
date
```

Passing Arguments

- You can pass arguments to make your programs more generalizable
- For example, what you if you had many data files that you wanted to extract the body mass from?
- Your input in the terminal could look something like this:

```
$ ExtractBodyM.sh ~/path/to/your/file/here
```

Example

- You can access the contents of the arguments passed from the command line using the $ character
- For example, $1, $2, $3, etc.
- Let's revisit the `dir.sh` file, but let you do it for any provided directory
- In your code, save the contents of the `$n` to a variable name like this:

```
#!/bin/bash
INPUTDIR = $1
ls -la $INPUTDIR
echo "Above are the directory listings for this folder"
pwd
echo "right now it is :
date
```

Now, edit your `ExtractBodyM.sh` script...

- to allow for any input csv file,
- and to allow you to name the file that you want to save as output (instead of BodyM.csv)
- Hint: you'll need to store the values for two inputs

Solution

```
#!/bin/bash
INPUTFILE = $1
OUTPUTFILE = $2
tail -n +2 $INPUTFILE | cut -d ';'
-f 2-6 | tr -s ';' ' ' | sort -r -n -k 6 > $OUTPUTFILE
```

In the terminal you could now type something like this:

```
$ ExtractBodyM.sh ../data/Pacifici2013_data.csv BodyM.csv
```

^ or whatever other files you could have to *pass as input*

- remember, in the case of *this specific program*, we are expecting our input files to have a certain column layout to extract the rows
- this would therefore be useful in a situation where we had many data files all in the same format and layout, but with different data points

Programming languages

- There are over 2000
- There is no perfect language for all tasks
- You are already learning several: regular expressions, python, R
- This class does not cover fast, compiled languages like C. Useful for heavy computational tasks

Getting started with python

- Python should already be installed on your VM

```
$ ipython
IPython 2.2.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's feature
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for ext
In [1]: import scipy
```

to exit ipython: CTRL-D CTRL-D to exit

IPython interactive mode

You can use python and IPython from the command prompt.

```
In [1]: 2 + 2
Out[1]: 4

In [6]: 2 > 3
Out[6]: False

In [11]: "I'm fine, " + "thank you"
Out[11]: "I'm fine, thank you"
```

try it out. . . .

Python operators

```
+ Addition
- Subtraction
* Multiplication
/ Division
** Power
% Modulo
// Integer division == Equals
!= Differs
> Greater
>= Greater or equal &, and Logical and |, or Logical or
!, not Logical not
```

Variables

You typically manipulate variables in programming languages.

```
In [12]: x = 5
In [13]: x
Out[13]: 5
In [14]: x + 3
Out[14]: 8
In [15]: y = 8
In [16]: x + y
Out[16]: 13
In [17]: x = "My string"
In [18]: x + " is now longer"
Out[18]: "My string is now longer"
In [19]: x + y
TypeError: cannot concatenate "str" and "int" objects
```

modulus

The modulus operator % returns the remainder of an integer division

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

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.rpartition
```

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
```

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', 'Arabidopsis

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!

```
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]
```

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), ^; 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 "d
```

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.