Prerequisites

- To learn how to write and run real programs we need:
- A decent text editor to write Python code.
  - Text editor is an app that allows you to easily edit and save simple text files.
  - Recommended editors:
    - Notepad++
    - Sublime Text

- Python 3X interpreter properly installed
  - Here are instructions for windows and for mac
  - To check: open a command prompt/terminal and type *`python`* (or *`python3`* for mac) – this should open Python shell. To exit shell type `exit()`
  - If you are unable to do this, please use help from tutors or ITS

Python program structure Running python scripts from command line

Workout 05.04

by Marina Barsky

Writing code in text editor

- So far we have played around with Python commands in the Python IDLE shell, or were running python code using PyCharm graphical user interface.

- We want to write a "serious" Python now.

- In a text editor of your choice (or in IDLE->new file), type the following Python code:

```
print ('Very serious program')
```

- Save the file as *serious.py*.

Running script from command line

- Let's assume our script *serious.py* is saved in a subdirectory *my_scripts* under the home directory of user *monty*.

- Then open a terminal or a command-line window and type:

```
monty@python:~$ cd my_scripts
monty@python:~/my_scripts$ python serious.py
Very serious program
monty@python:~/my_scripts$
```

- **Do not go to the next step until you were able to run your script using Python interpreter from command-line.**

Byte count in serious.py

- In file *bc.py* (byte-count) write a program that reads the text in file *serious.py*, and counts the number of bytes in this file:

```
name = 'serious.py'
handle = open(name, 'r')
text = handle.read()
print(name, 'has', len(text), 'bytes')
```

Structuring programs

```
name = 'serious.py'
handle = open(name, 'r')
text = handle.read()
print(name, 'has', len(text), 'bytes')
```

- In many programming languages (e.g. Java, C and C++), it is not possible to simply have statements sitting alone at the bottom of the program
- They are required to be part of a special function that is automatically invoked by the operating system when the program is executed from command-line. This special function is called ***main***.
- Although this is not required by the Python programming language, it is a good idea that we can incorporate into the logical structure of our program.
- The four lines of the code above are logically related to one another in that they provide the main tasks that the program will perform.
- Since functions are designed to allow us to break up a program into logical pieces, it makes sense to call this piece ***main***.

Adding main

- Change your *bc.py* code to:

```
def main():
    name = 'serious.py'
    handle = open(name, 'r')
    text = handle.read()
    print(name, 'is', len(text), 'bytes')
```

- We have defined a new function named *main* that doesn't have any parameters.
- The four lines of main processing are now placed inside this function.

```
main()
```

- Finally, in order to execute that main processing code, we need to **call** the *main* function (last line). When you run your script again, you will see that the program works the same way as before.

Recommended program structure

- First, import any modules that will be required.
- Second, define any functions that will be needed.
- Third, define a *main* function that will get the process started.
- Finally, call the *main* function (which will in turn call the other functions as needed).

Note: In Python there is nothing special about the name *main*. We could have called this function anything we wanted. We chose *main* just to be consistent with some of the other languages.

- Modify your script *bc.py* to have a proper program structure with *main*

Importing functionality

```
import random
```

This is preferred – so you always know where the function is coming from

- To use:

```
random.choice (["rock", "paper", "scissors")
```

```
from random import *
```

- To use:

```
choice (["rock", "paper", "scissors")
```

Importing our own modules

File ***my_func.py*** implements simple functions

```
1.    def squareit(n):
2.        return n * n
3.
4.    def cubeit(n):
5.       return n*n*n
6.
7.    def main():
8.       anum = 15
9.       print(squareit(anum))
10.      print(cubeit(anum))
11.
12.
13.   main()
```

Importing our own modules

File ***my_func.py***

```
1.    def squareit(n):
2.        return n * n
3.
4.    def cubeit(n):
5.       return n*n*n
6.
7.    def main():
8.       anum = 15
9.       print(squareit(anum))
10.      print(cubeit(anum))
11.
12.
13.   main()
```

File ***test.py***

```
1. from my_func import *

2. anum = int(input("Enter number"))
3.    print(squareit(anum))
```

What do you notice when you run test.py?

Conditional invocation of *main()*

- Before the Python interpreter executes your program, it defines a few special variables
- One of those variables is called **__name__** and it is automatically set to the string value **"__main__"** when the program is being executed by itself in a standalone fashion
- On the other hand, if your code is being imported by another program, then the **__name__** variable is set to the name of your module
- This means that we can know whether the program is being run by itself or whether it is being used by another program and based on that observation, we may or may not choose to execute some of the code that we have written.

Function *main*

File ***my_func.py*** implements simple functions

```
1.    def squareit(n):
2.        return n * n
3.
4.    def cubeit(n):
5.       return n*n*n
6.
7.    def main():
8.       anum = int(input("Please enter a number"))
9.       print(squareit(anum))
10.      print(cubeit(anum))
11.
12.   if __name__ == "__main__":
13.      main()
```

This is executed only if we run module ***my_func.py***, not if:

```
from my_func import *
import my_func
```

Word count program

- Using an example from the lecture slides, write a Python script which counts the number of all real alpha-words in file *spam.txt*.

- First, define function *count_words*, next define function *main* which invokes *count_words* with filename **as a parameter**. Finally, invoke main by passing the name of file as a parameter:

```
if __name__ == "__main__":
    main('spam.txt')
```

- Save your code in file *wc.py* (word-count), test it by running it from command-line.

What if we want to create a general word-count program that would count words in any file?

Running script with command-line arguments

- There is a special module which has all the functions needed for the interaction of your program with the operating system:

```
import sys
```

- For every invocation of Python script, ***sys.argv*** is automatically a list of strings representing the arguments (as separated by spaces) on the command-line.

- To understand how it works, let's create a simple script *count_args.py*:

```
import sys
print(sys.argv, len(sys.argv))
```

- Now, run this script from command-line:

```
monty@python:~/my_scripts$ python count_args.py
monty@python:~/my_scripts$ python count_args.py foo and bar
```

General word-counting program

- Modify your *wc.py* code to count words in a file whose name is passed as a command-line parameter.
- Here is an example of how to use the filename passed as a command-line parameter:

```
import sys


name = sys.argv[1]
handle = open(name, 'r')
text = handle.read()
print(name, 'has', len(text), 'bytes')
```

- Run a new version passing different existing file names as parameters. Take care of the case when the file does not exist. Submit your *wc.py* to Google classroom.

Vocabulary counting program

- In a new file *vocab.py*, use a dictionary data structure to count the number of **distinct** words in a file, the name of which is passed as a parameter to the Python script.
- You can start from a sample code in lecture slides.
- Keep the style very similar to your previous code, with *main* function and its conditional invocation. Pass the file name as a command-line parameter.
- Output the total number of unique words (author's vocabulary)

Collect counts for each word

- In file *top_words.py*, write a program that collects the count of how many times each unique word has appeared in the file

- For example, running your program with *spam.txt* as a parameter:

```
monty@python:~/my_scripts$ python top_words.py spam.txt
```

- should produce the following output:

```
{'holidays': 1, 'poptarts': 3, 'for': 1, 'spam': 3,
'Will': 1, 'I': 3, 'and': 3, 'get': 1, 'like': 2}
```

Most frequent words

- Modify top_words.py to extract the top 5 words with the largest counts
- For this, collect dictionary tuples in form of (value, key) into a new list, sort the resulting list in reverse order, and output the top-5 slice
- Submit *top_words.py*

In total submit:

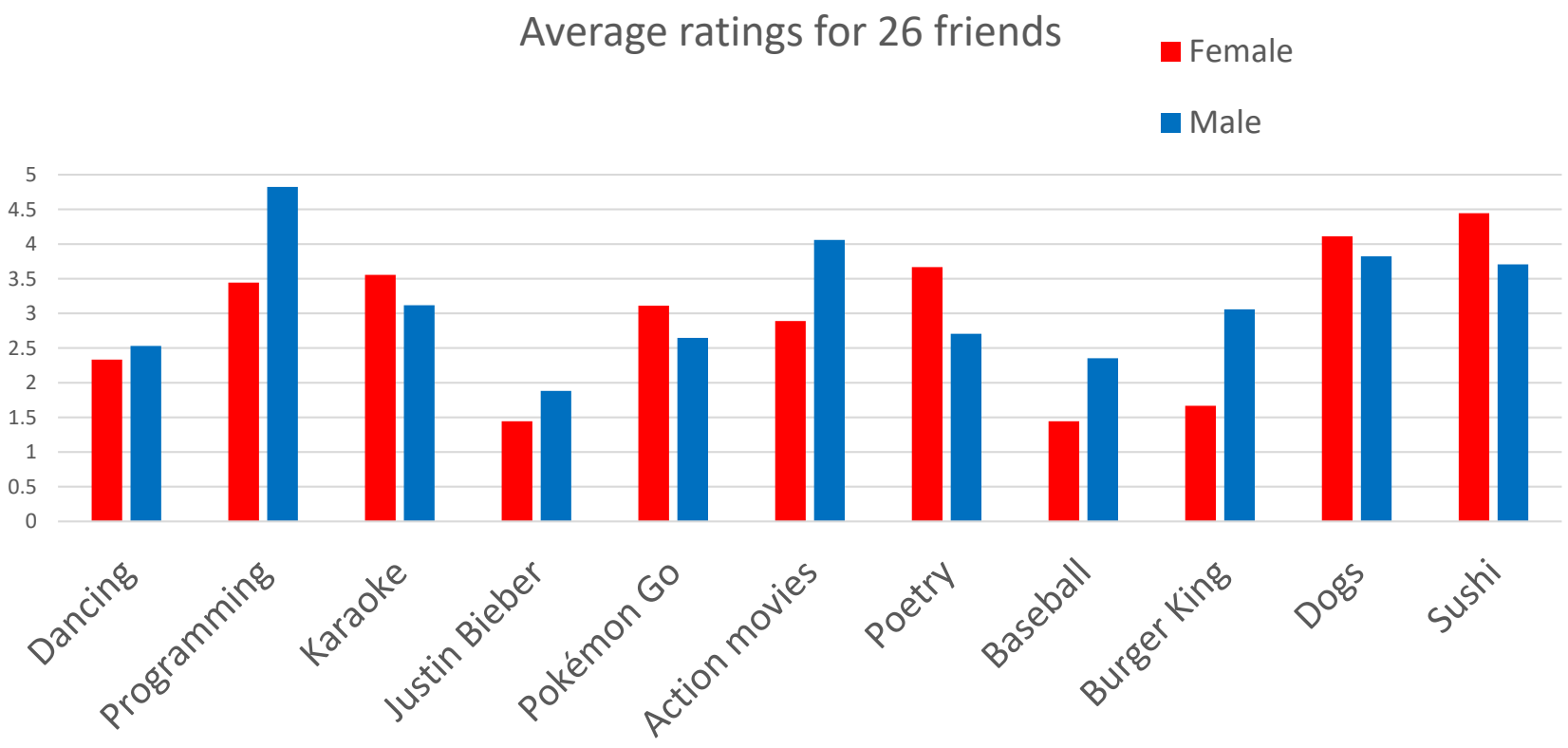
- *bc.py*
- *wc.py*
- *vocab.py*
- *top_words.py*

Homework 4

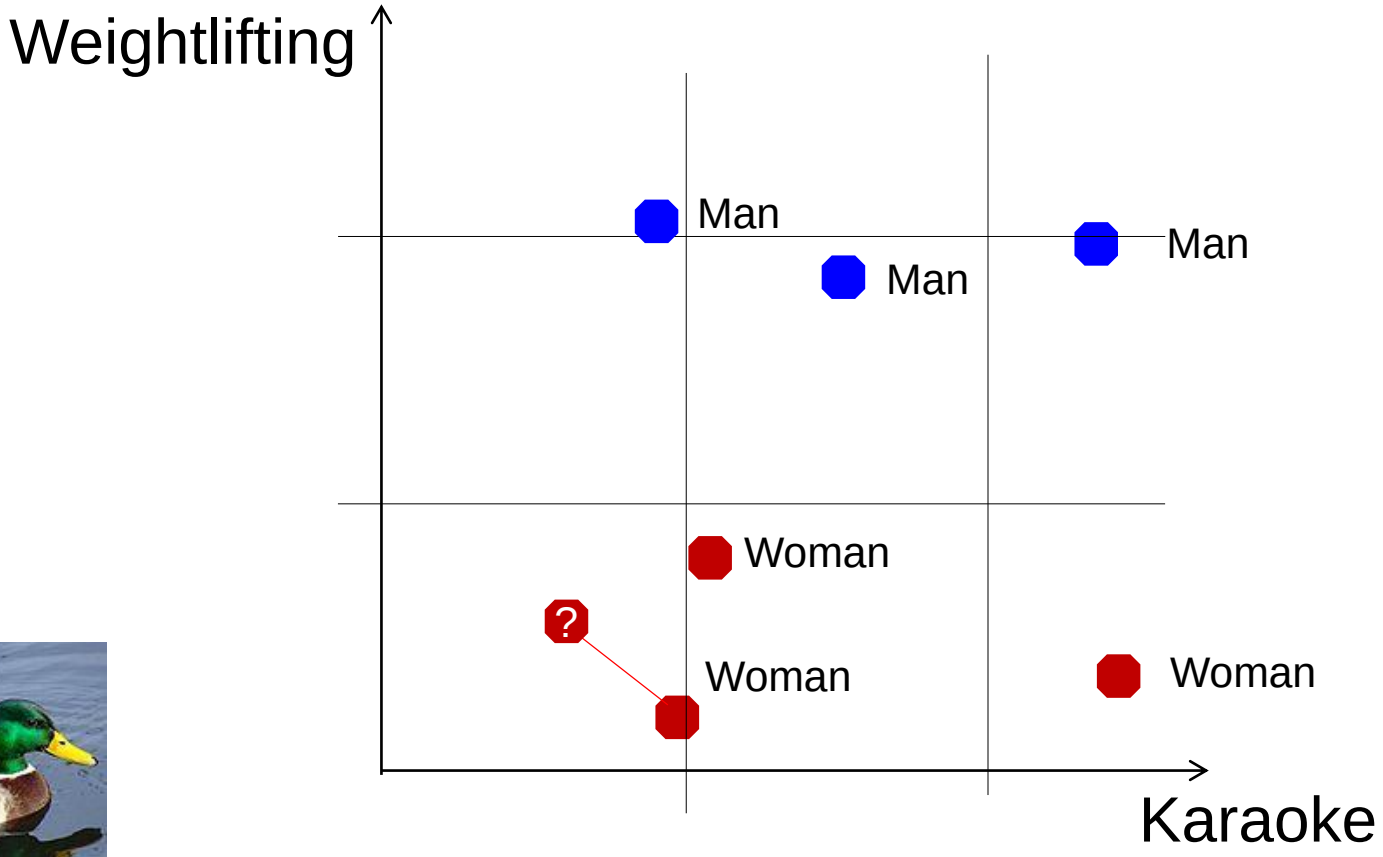
Using files and dictionaries to find friends

Demo data app (not the demo of homework, just an example):
http://hope.simons-rock.edu/~mbarsky/intro18/lectures/data/predict/

My friends dataset

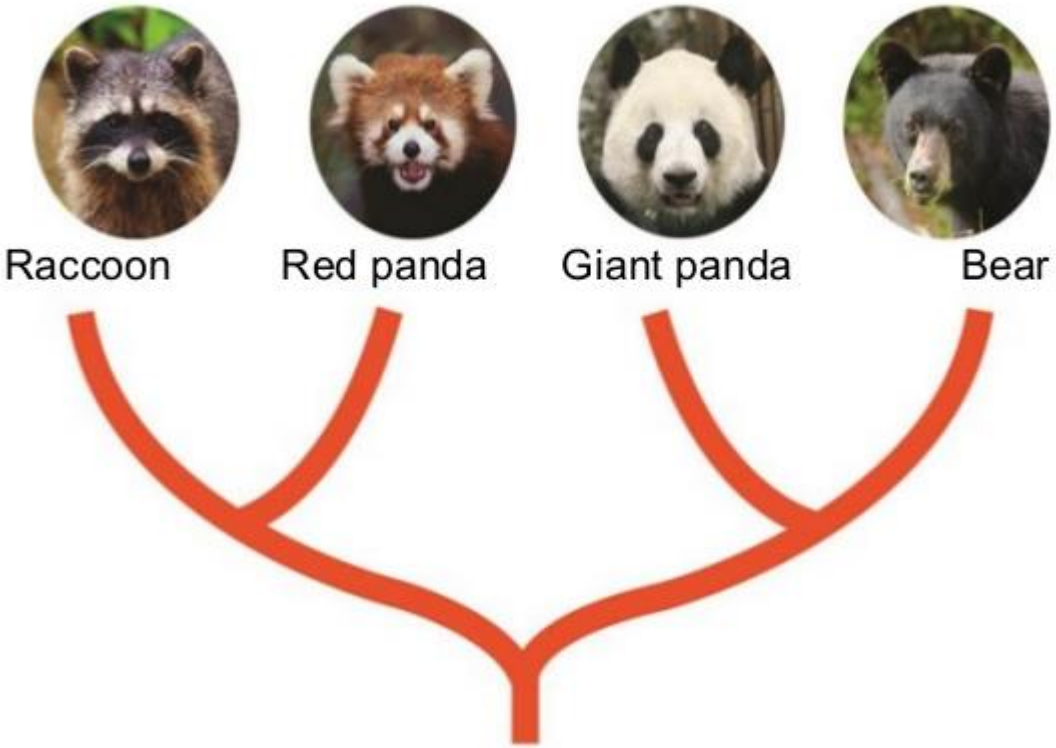


Classification by proximity to nearest neighbors



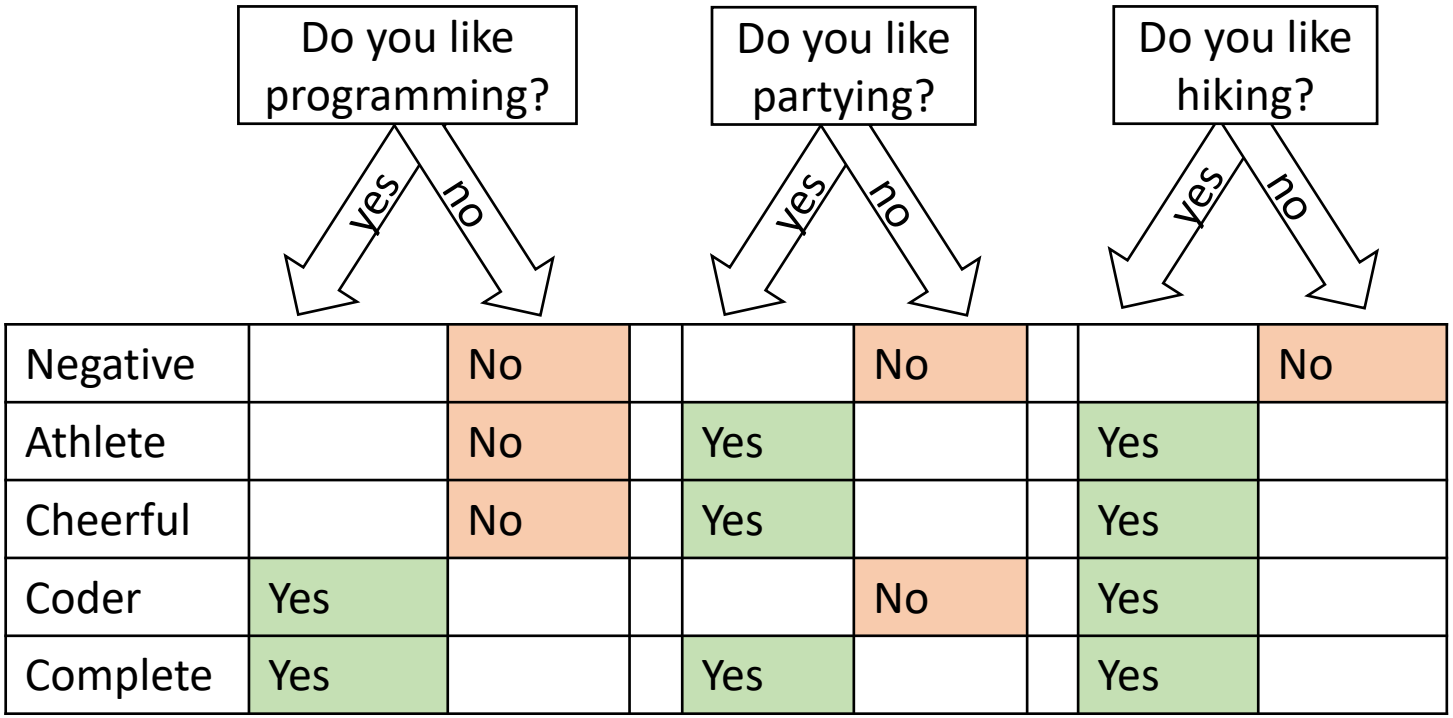
Red Panda: a Bear or a Cat?

Red Panda: a Bear or a Cat



Flynn, J. J.; Nedbal, M. A.; Dragoo, J. W.; Honeycutt, R. L. (2000). "Whence the Red Panda?" Molecular Phylogenetics and Evolution.

Collect preferences from users and store them in a file



Negative		No			No		No
Athlete		No		Yes		Yes	
Cheerful		No		Yes		Yes	
Coder	Yes				No	Yes	
Complete	Yes			Yes		Yes	

Who is preferred friend for *Athlete*?

Athlete		No		Yes			Yes	
Cheerful		No		Yes			Yes	

OR

Athlete		No		Yes			Yes	
Coder	Yes				No		Yes	

?