BMI-312 Modeling Biomedical Data

Quick Python

***T*here's *M*ore *T*han *O*ne *W*ay *T*o *D*o *I*t**

TMTOWTDI

(pronounced Tim Today)

For Extra Help

- Free Python course on Udacity

  https://www.udacity.com/course/introduction-to-python--ud1110

- Interactive Python Tutorials

  https://www.learnpython.org/

- Run basic code online

  http://www.pythontutor.com/visualize.html#mode=edit

Courtesy slides: Dr. Paea LePendu

Object-Oriented Programming (OOP)

Class

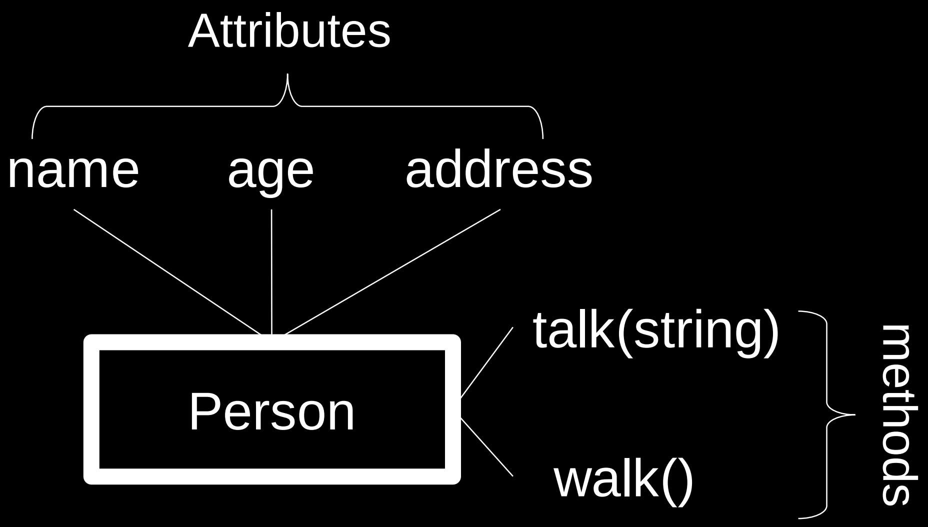
An abstract data type

- Defines attributes
- Defines methods

Object

A concrete instance of a class

- Attributes have values.
- Performs methods



```
p1 = Person(name='Mary',
age=22);

p1.talk('hello')
p1.talk('bye’)
p1.age = 23;

p2 = Person(name='J. Smith',
age=25);
p2.talk('good morning’)
p2.walk()
```

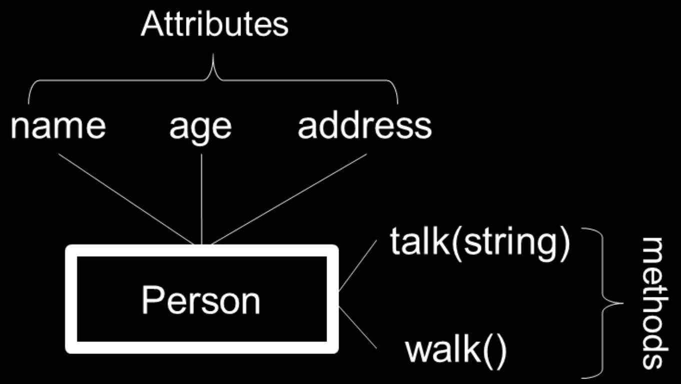
How about play?

If the class does not define play(), a person cannot play.

Object-Oriented Programming (OOP)

- Python is an OOP.
  - It offers many built-in classes.
  - Additional packages (a.k.a., libraries) are available containing other useful classes.

- Application Programming Interface (API)
  - Access points to definitions of classes and their attributes and methods.



Atomic (Basic) Data Types

- Like in other languages: int, float, bool, string, None
- Unlike in Java or C/C++, Python infers data types and casts them automatically.
  - Be careful. If needed, check types using the **type()** method.
- Naming conventions: letter, numbers, _
  - Do not use "." in names. ("." is reserved for accessing attributes and calling methods. After all, it is OOP).
  - Meanings of leading and trailing underscores

```
a = 2     # No claim of type. Python infers it is an integer.
b = 2.0   # has the same precision as a floating number.
c = True   # vs. False
d = "hello world"
e = None # same as null or NA in other languages

one_ratio = 2/1; type(one_ratio)
another_ratio = 2//1; type(another_ratio)
```

String

- String manipulations are important.
- You shall ***master*** the art of string manipulations.

```
label = "office"  # double quote
same_label = 'office'  # single quote
empty_label = ""  # empty string
label_single_quote = "Gev's office" # single quote inside
label_multi_line = """Address line 1
Address line 2"""  # string spanning multiple lines

s = "hello world"
s  # on-screen output, suppressed if ends with a ;
s[0:4] # "hell"
s[6:] # "world"
s[:-6]  # "hello"
# counting from end
```

Python is 0-indexed.

left-close right-open interval

String

- String functions ***do not modify*** the value of the original string.
  - A copy of the string with the new value is returned.
- Google before writing a new string function.

```
s = "    hello world   "

s.replace("world", "universe") ## s does not change.
s1 = s.replace("world", "universe") ## save the changed string by assigning the returned value to a new variable.
s.strip()
s.split()
s.split("o")
s.upper()  ## useful when inputs have mixed cases.
s.strip().upper()  ## concatenate methods
s.strip().index("o")  ## the first occurrence of letter o
```

API

❖ How to find the attributes and methods of an object?

```
dir(s)
## dir(objectName)
## displays attribute names and method names of
the specific object


help(s.replace)
## help(methodName)
## displays descriptions of the arguments and
returned values of a specific method.
```

Complex Data Types

❖ Data containers:

- Classes contain atomic or complex data types in an organized manner.

Types	Defined by	Notes
Set	set( )	**Unordered** ***unique*** elements
List	[ ]	**Ordered** elements
Tuple	( )	**Unmodifiable** list
Dictionary	{ }	Key-value **pairs**, **keys are unique**

```
a_list_of_names = ['Kay', 'John', 'John']
a_set_of_names = set(['Kay', 'John', 'John'])
a_tuple_of_names = ('Kay', 'John', 'John')
a_dic_of_names = {'Kay': 20, 'John': 22 , 'John': 30}

a=a_list_of_names[0]
b=a_set_of_names[0]  # error: set does not support indexing
c=a_tuple_of_names[0]  # you can access tuple element
a_tuple_of_names[0] = 'May' # error: you cannot update tuple
d=a_dic_of_names['Kay'] # search by key, return the value
```

Control Flow – Conditional

- if … elif … else

- Logical operators
  - ***boolean:*** and, or, not (used on boolean values)
  - ***bitwise:*** &, |, ^ (used on integers or booleans)
  - **To be safe, always put conditions inside parenthesis**

✓

```
a = 100
b = 200
c = 300
print("The smallest
is");
if a < b and a < c:
    print("a")
elif b < a and b < c:
    print("b")
else:
   print("c")
```

✗

```
a = 100
b = 200
c = 300
print("The smallest
is");
if a < b & a < c:
    print("a")
elif b < a & b <c:
    print("b")
else:
   print("c")
```

✓

```
a = 100
b = 200
c = 300
print("The smallest
is");
if (a < b) & (a < c):
      print("a")
elif (b < a) & (b < c):
      print("b")
else:
   print("c")
```

Control Flow – Repetitive

❖ for vs. while

- Use “for” loop if the number of iterations is known in priori.
- Use “while” loop otherwise. You’ll need it when
    - getting user input from keyboards,
    - reading a file line by line,
    - receiving data from database connections.
- In addition, use “continue” and “break” if necessary.

```
names = ['Kay', 'John']
for i in names:
  print('Hello ' + i);
```

```
name = input("What's your name?");
while len(name) > 0:
  print('Hello ' + name + '\n');
  name = input("What's your name?");
print('Good Bye.');
```

Colon & Indentation

Spacing Matters!

❖ Leading whitespace

- determines the grouping of statements
- tab vs. spaces
    - ***Keep it consistent***

✓
```
friendly = True
if friendly:
    print("Hello World!")
```

✗
```
friendly = True
if friendly:
print("Hello World!")
```

Short Hands

- ❖ Make the code less cumbersome.
- ❖ Avoid using short hands that are not intuitive.
- ❖ Maintainability:
  - One would be able to read, understand, and modify the longer code faster and with fewer errors than the shorter one!

```
a=1; b=2;

if a > b:
  x=10
else:
  x=11

x=10 if a > b else 11  ## okay

x = a > b and 10 or 11 ## huh?
```

Short Hands – Comprehensions

- Comprehensions are short hands to create and populate a list or a dictionary.
- It executes faster than loops.
- You need to master comprehensions
  - We will see more examples when discussing data structures

```
 # comprehension of lists
 b=2;
 a=[1, 5, 7];
 y=[x+1 for x in a if x > b]
```

```
y = [];
for x in a:
  if x > b:
     x = x + 1
     y.append(x)
```

```
# comprehension syntax:
[expression for item in list if conditional ]
```

Functions

- ❖ Defining functions to ***capture repetitive tasks*** is a good practice.

If you ever find yourself copying and pasting code, stop and think: should I make a function?

- ❖ Every function has a returned value. If you do not specify it, it returns None.
  - To return multiple values, organize them into a data structure.

```
def raise_mod(x, n=2): # n is optional, default=2
    return pow((x+1), n)

print(raise_mod(2,3)) # 3^3 = 27
print(raise_mod(2)) # 3^2 = 9
```

Functions – *Lambda*

- ❖ Allows multiple arguments but ***only one expression***
- ❖ Expression vs. statement:
  - An expression evaluates to a value, e.g., x + y, a < b,
  - A statement does not, e.g., if statement, for loop statement

```
f = lambda x: x*x    # lamba is a function
y = f(10)

f = lambda x, y: x*y    # lamba with two arguments
y = f(10, 4)

f = lambda x, y=3: x*y    # arguments with default values
y = f(10)
```

Functions – *Lambda*

- ❖ Unnamed (anonymous) function.
  - Useful when passing a one-off function as an argument to another function.
  - Advanced users apply lambdas often in large datasets.
  - We'll see more examples when discussing data structures.

```
a = [1, 5, 7]


# apply an operation to each item in a list using the
  built-in map function
b = list(map(lambda x: (x+1)**2, a))


# filter a list using the built-in filter function
c = list(filter(lambda x: x > 3, a))
```

Libraries/Packages – Import

- Collections of classes and methods
- Check existing libraries before writing a new function or class
- To use, simply import
  - Hey, ***always read the documentations***

```
import math
a = math.log(10)
b = math.log(10, 10)

import math as mt
a = mt.log(10)

from math import *
# bad practice!
# It overwrites variables & functions in workspace
```

Libraries – Install

- pip: the python package manager
  - It installs the specified package and the required dependent packages.
  - It installs the package in the location associated with the python executable.
  - When “import” fails, check the above items.

```
pip install httpie

pip install httpie --upgrade

pip uninstall httpie
```

Libraries – Conda

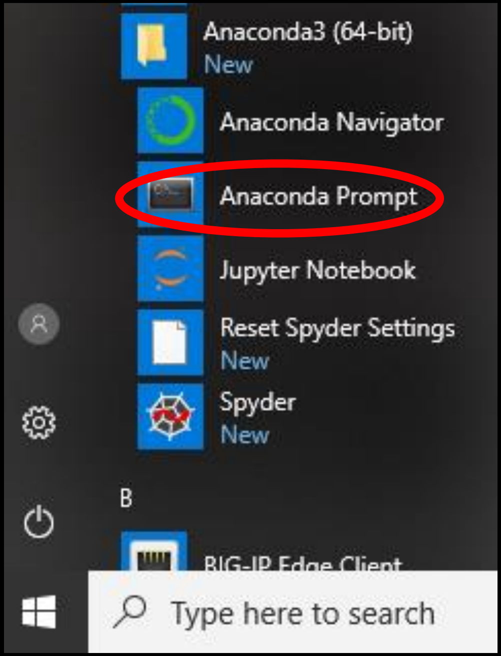
- conda - package and environment manager
  - pip can install any package (PyPI), conda has a more vetted selective repository
  - Conda has better control over compatibility
  - Conda can also do environment management

```
conda install numpy

conda update numpy

conda remove numpy
```

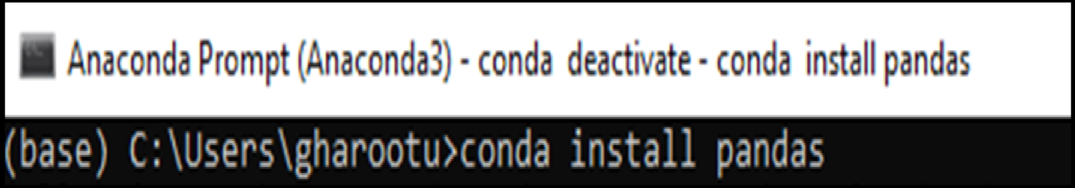
Dependent Python Libraries

❖ Install Python packages using “pip”, the python package manager.

1.



2.



```
Anaconda Prompt (Anaconda3) - conda deactivate - conda install pandas
(base) C:\Users\gharootu>conda install pandas
```

Python Libraries/Packages

Install ***vs.*** ***Import***

- Copy a package to the hard drive
- Install only once

- Load codes in memory
- Import in each session

Error Handling

- ❖ Two types of errors, as in all programming languages
  - Syntax errors
    - Errors during compilation.
  - Exceptions
    - Error during execution, a.k.a, run-time errors.
  - You’ll soon do much better.
  - And your dear friend of Google can help you get there faster.

- ❖ Logic errors, as in everyday life.
  - The output is wrong.
  - The output is different from your expectations.
  - You’ll do better eventually after some significant time and effort.

Try … Except

- To handle exceptions

```
try:
  file=open('test.txt');
except:
  print('file not found');


import math;
try:
     math.log(0);
except:
  print('check equation');
```

```
import math;

try:
  math.log(0);
  file=open('test.txt');
except IOError:
  print('file not found');
except ValueError:
  print('check equation');
except:
  print('something is wrong');
```

Debugging

- A fancy debugger does NO magic.
- You are the magician.

Your Arsenals

- ***Stack Traceback***
- ***Breakpoint***
- ***Printing***

Debugging In Jupyter

- Use "print"
- Use the built-in Python debugger "*pdb*".

```
import math;

a = [6, 11, 20, 9, -5];
b = [];
for i in a:
    x = math.log(i);
    y = i/x;
    b.append(y);
```

```
import math;

a = [6, 11, 20, 9, -5];
b = [];
for i in a:
    print('i=', i, '\n');
    x = math.log(i);
    print('x=', x, '\n');
    y = i/x;
    print('y=', y, '\n');

    b.append(y);
```

```
import math;
import pdb;

a = [6, 11, 20, 9, -5];
b = [];
for i in a:
    pdb.set_trace();
    x = math.log(i);
    y = i/x;
    b.append(y);
```

A Note on Versions

Python 2 vs. Python 3
NOT Compatible!

- Many academic libraries work only on python 2.7, the last update of Python 2.
- Most platforms maintain both Python 2.7 and Python 3. Be careful which version you are executing.

QUESTIONS!

KAHOOT!

COMPUTATIONAL LAB

Next time:

Data Structure with Python

HOMEWORK

- Finish reading chapters 3 and 15 in the textbook.