Python for Biophysics

Ignacio Calderon de la Barca

Coding bootcamp
with examples in Biophysics

BIOPHYSICS @ SFU
FORDE LAB

September 13, 2020

I. PYTHON BASICS

A. Data types

Most programming languages require you to define your variables to store a designed memory for storage and computation, so for example if you want to define three variables `a,b,c` in C you would write:

```
\\ ## Example in C
int a;
float b;
double c;
char d

\\ ## assign values
a = 55;
b = 60.8325;
c = 12.000000000000001;
d = "Python";

\\ ## print values
print(a, b, c, d);
```

where before assigning any values to our variables we have to specify what type they are. So either an integer, float, character or double precision variable (up to 15 decimal points) as `int, float, char, double`, respectively. However, in Python, the variable type is automatically understood by the program upon value assignment or operation.

(a) Try it yourself! Print the same values in an iPython cell. Remember that no `;` is needed at the end of the line. A convenient notation for powers in Python is `1e-5`, which means 1×10^{-5}, use this and print an extra value `e = 12+1e-15`.

(b) With the use of `type()` check the type of your variables. In a new cell play with converting the type of your variables. For example convert `a` to a float by typing `float(a)`; convert `c` to a string type with `str(c)`.

(c) A very convenient aspect of dealing with python structures is that operations update the state of your variables. For instance, if you divide `a/1.0`, the result has been converted to a float number.

(d) Python also provides the option of working with complex numbers, for example try converting `a` into a float `complex(a)`.

B. Data structures

Python is very powerful and versatile working with different data structures simultaneously. In the previous section I A, we briefly got some familiarity with the type of single-valued variables. Now to Extend and work with more complex data, we will learn about the different types of structures in Python that enables data manipulation with higher complexity.

1. Lists

Lists in Python are a collection of elements, that can be from different types. The dimensions do not necessarily have to match and one can manipulate and change the size of the elements very easily.

(a) Construct a list, name it `L1`, that contains all multiples of 3 up to 21. You can do this manually first `L1 = [3, 6, 9, 12, 15, 21]`. Note that lists in Python are contained in square brackets.

(b) Another way of creating lists is, for example creating a range object and then converting it to a list. Create a list of values from 44 to 88 in steps of 4 using `list(range(44,89,4))`

(c) A list is a data structure that can have mixed types of data, for example it can contain `float, int, str` in the same object. Create a three mixed lists, and each list should have 3, 7 and 5 elements, respectively. Name them `a, b, c`.

(d) A powerful advantage of working with lists is that they are easy to combine to create nested and sub-nested lists or lists of lists. Lets create an empty list `All`, now use the append command repeatedly to create a sub-nested structure: `All.append(...)`, where `...` represent the three lists.

C. Arrays

(a) Arrays are ordered structures that have an straightforward interpretation, they can represent vectors, matrices and tensors.

(b) Lets say we have a matrix $A \epsilon \mathcal{R}^{n \times m}$. Which means that it has n rows and m columns.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$$

The conventional notation is that the firs subindex refers to the rows and the second subindex refers to the columns. In Python we can index entire columns and rows using ':' notation. For example if we would like to index the entire second row we would call 'A[1,:]' where '1' refers to the second column and ':' indicates all columns. Likewise if we would like to get the entire last column and all rows then we will type 'A[:,-1]' where '-1' refers to the last element of the columns. So

$$A[1,:] = \begin{bmatrix} a_{21} & a_{22} & \cdots & a_{2m} \end{bmatrix}$$

and

$$A[:,m] = A[:,-1] = \begin{bmatrix} a_{1m} \\ a_{2m} \\ \vdots \\ a_{nm} \end{bmatrix}$$

we will code an example below

D. Magic functions

Python has many "magic" functions which are functions that provide a shortcut access to information about our code. Here we present few of them.

(a) Define a function that outputs the squared value of an input number, say 22. Save your file as `_22_squared_.ipynb`. Now in a new scrip, run your file using the magic function `%run _22_squared_.ipynb`.

(b) Another very useful function is `%timeit`, whcich will print the time that it took to execute the routine.

E. Practice problems

(a) Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included). The numbers obtained should be printed in a comma-separated sequence on a single line.

(b) Write a Python program to accept the user's first and last name and then getting them printed in the the reverse order with a space between first name and last name.

(c) Write a Python program to find the volume of a sphere with diameter 12 cm. Formula: $V = \frac{4}{3}\pi r^3$

(d) Create the below pattern using nested for loop in Python.

```
*
**
* * *
* * **
* * * * *
* * **
* * *
**
*
```

(e) Write a Python program to reverse a word after accepting the input from the user. Sample Output: Input word: ineuron Output: norueni

II. THE FUNDAMENT OF PROBABILITIES: THE FLIPPING COIN

A. Random numbers

1. Notion of probabilities

(a) Use `help()` to read the arguments that `np.random.rand` takes. Be sure to understand how use this function.
(b) Generate many random numbers (floats between 0 and 1), collect them in an array variable call it `r`. Print on screen their maximum and minimum, using `print()`.
(c) Use a conditional statement to identify and answer: How many numbers are bigger than 0.5, 0.75 and 0.9?
(d) Based on your previous answer what is the likelyhood of getting a number bigger than 0.75 from `r`. Check what the bluebinomial distribution is.
(f) To what number would you compare the random numbers in `r` so that 60% of the time you get a `True` boolean outcome.
(g) Define a function that identifies and counts the number of heads (`k`) and tails (`t`) given an array of random numbers as input.

2. Biased coin

(a) Simulate $N = 10$ tosses of a biased coin with $p = P(\text{Heads}) = 0.6$.
(b) Count the number of Heads, test that your function in part II A 1(g) is working correctly.
(c) Now lets build up statistics. Repeat the experiment $M = 100$ times, saving the number of heads every run in an array, you can call it `Y`.

3. Visually explore the data created

(a) What are the maximal and minimal number of heads observed?
(b) How many unique values are in your dataset? Check the documentation for `np.unique`.
(c) Import the graphic library `import matplotlib.pyplot as plt` what type of plot would be more informative to see the frequency of 0 heads, 1 head, 2 heads, etc...
(d) Create a figure that shows the **frequency distribution**, label your axis, and provide a title and a legend. You can check the bluehandbook.
(e) Add vertical lines, use `vlines` to display the mean μ, and standard deviation $\pm\sigma$.
(f) Adjust your axis limits using `xlim` and `ylim`.

B. Statistics and Distributions

1. *Mean, variance, standard deviation and quartiles*

Consider x as the outcome of some observable attribute X, and $P(x)$ is the probability of that outcome occurring,

$$\mu = E[X] = \sum_{x\in X} xP(x) \qquad \text{mean} \qquad (1)$$

$$\sigma^2 = \mathrm{Var}[X] = \sum_{x\in X} (x-\mu)^2 P(x) \qquad \text{variance} \qquad (2)$$

$$\sigma = \mathrm{std}[X] = \sum_{x\in X} (x-\mu) P(x) \qquad \text{standard deviation.} \qquad (3)$$

(a) Using your code from II A simulate three datasets of $N = 10, 100, 100000$ runs of $M = 50$ coin tosses, with a probability $P(\text{Heads}) = p = 0.5$. Record the number of heads, `k` of each set.

(b) In a markdown cell, type your prediction for what the Probability Distribution Function **(PDF)** of probability will look like. Make sure to think about where the peak will lie, what determines the location of the peak?

(c) Calculate the **mean**, **variance** and **standard deviation** of the PDF, for all three experiments. As a coding exercise, do not use NumPy nor other libraries, create your own functions.

(d) Explore the relationship of how μ depends on M and p. Write down your predictions. Try to come up with a mathematical relationship.

(e) To make this notion more visual run the experiment for different biased coins, resulting in an outcome of $p = P(\text{Heads})$ that ranges from 0 to 1, in steps of 0.1. Plot the expectation value and variance as a function of p. Comment your results.

(f) Now run the experiment again for 15 values for N ranging from 10 to 10^6, tossing $M = 50$ coins. Check the documentation for `np.logspace`. Then, plot the mean or expected value of the outcome (# of heads) as a function of N. Also, plot the variance as a function of N. Comment your result.

(g) Lastly, plot the ratio of the mean to the standard deviation as a function of the number of coins tossed $\frac{\sigma(M)}{\mu(M)}$. Play by changing your axis scale using `plt.yscale('log')` and likewise for x.

You should see that the following relationship holds

$$\frac{\sigma}{\mu} \sim \frac{1}{\sqrt{N}} \qquad (4)$$

which is a consequence of the **Central Limit Theorem (CLT)**. It provides a basis to the assertion that the more observation are made the better the approximations get.

C. Cumulative distribution and inter-quartile range

(a) Compute and plot the cumulative distribution and find the number of heads for percentiles: 0.1, 0.25, 0.5, 0.75, 0.9.

(b) Make a figure showing how the median and Inter-Quartile-Range (IQR) depend on M and p.

III. RANDOM WALKS

1. *Mean Squared Displacement*

(a). Simulate a 1D random walk of length N steps. Generate $M = 10000$ such walks for several N (say $N = 10$, 20, 30, 40, and 50). For each walk, record the **end-to-end distance**, $r(N)$. As you simulate, think about how to store your results as an array, where you have as many rows as values of N and as many columns as the number of walks you simulated. So, at the end of your simulation you'll have a single array of data.

(b). Look up a python function that can calculate a mean of an array. Figure out how to calculate the mean $\langle r(N)\rangle$, for your different choices of steps with a single call to the function using your array of data. Print the resulting values.

What do your simulations suggest that the mean of the end-to-end distance of a random walk is? Write your answer in a text cell.

(c). Calculate the **mean squared end-to-end distance (MSD)**, $\langle r(N)^2 \rangle$. Plot the mean squared end-to-end distance (MSD) vs N as a scatter plot. What sort of curve does the squared end-to-end distance appear to be as a function of N? If you had taken N steps all in the same direction (i.e. not a random walk, but a directed walk), what sort of curve would $r(N)^2$ be as a function of N?

(d). From your data above, choose 3 different values of increasing N and make histograms of $r(N)$ as 3 subplots. Use the same range for each and figure out how many bins you need to use so that the binning is the same for each. As above, normalize your histograms so that they are densities. Do these distributions look like any distribution that you know? How do the distributions change as you increase the number of steps? Can you think about how the quantities you calculated in 2 and 3 relate to the parameters of these distributions? Write your answer in a text cell.

(e) Creating fast code is really important. You may have noticed or questioned that in the previous code we generated a new experiment only to pull out the information of the last position for each walkers. Import the timer function `from timeit import default_timer` and use it as follows

```
## time your code
from timeit import default_timer}
t0 = default_timer()

##################
## write code.. ##
##################

tf = default_timer()
print(f"time taken: {tf-t0:.4f} seconds")
```

IV. DIFFUSION COEFFICIENT AND CURVE FITTING

1. Linear fit and MSD

Consider the linear relation $f(x) = \beta_0 + \beta_1 x$
Example:
$y(x) = 3 + 10x + \epsilon$
Using the same idea with the MSD

$$\begin{aligned} MSD(t) &= b + 2Dt \\ MSD(t) &= \beta_0 + \beta_1 t \\ \text{where} \quad \beta_0 &= b = 0 \\ \beta_1 &= 2D \end{aligned}$$

2. Find diffusion from position distributions

$$\begin{aligned} P(\Delta x_i, t) &= \frac{1}{\sqrt{4\pi D t}} e^{-\frac{\Delta x_i^2}{4Dt}} \\ P(x, \sigma^2) &= \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\langle x\rangle)^2}{2\sigma^2}} \end{aligned}$$

From the equations above we find that the probability distribution for one random walker x_i is a Gaussian distribution with mean $\langle x \rangle = 0$ and standard deviation $\sigma^2 = 2Dt$. The probability for all random walkers to be in position Δx is the product of all random probablity distributions for all random walkers.

$$\begin{aligned} P(\vec{x}, t) &= P(\Delta x_1, t) \times P(\Delta x_2, t) \times \ldots \times P(\Delta x_M, t) \\ P(\vec{x}, t) &= (4\pi D t)^{-M/2} e^{-\frac{1}{4Dt}\sum_i^M x_i^2} \end{aligned}$$

Linear relantionship between σ^2 and t

$$\sigma^2(t) = 2Dt$$

For a single data point t_i then we can find D

$$\sigma^2(t_i) = 2Dt_i$$
$$\rightarrow D = \frac{\sigma^2(t_i)}{2t_i}$$

If we have all time-steps information then D can be found using a linear fit

$$\sigma^2(t) = 2Dt$$
$$y(x) = mx + b$$

where $\quad y(x) = \sigma^2(t), \quad b = 0, \quad m = 2D$, so the diffusion term is

$$\boxed{D = \frac{m}{2}}$$