Lab 1: Introduction to Python, data types, and logic

Data Science for Biologists • University of Washington • BIOL 419/519 • Winter 2019

Course design and lecture material by Bingni Brunton (https://github.com/bwbrunton) and Kameron Harris (https://github.com/kharris/). Lab design and materials by Eleanor Lutz (https://github.com/eleanorlutz/), with helpful comments and suggestions from Bing and Kam.

Table of Contents

Helpful Resources

- Python Data Science Handbook (http://shop.oreilly.com/product/0636920034919.do) by Jake VanderPlas
- Python Basics Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/e30fbcd9-f595-4a9f-803d-05ca5bf84612) by Python for Data Science
- Jupyter Notebook Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/48093c40-5303-45f4-bbf9-0c96c0133c40) by Python for Data Science
- Matplotlib Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/28b8210c-60cc-4f13-b0b4-5b4f2ad4790b) by Python for Data Science
- Numpy Cheat Sheet (https://datacamp-community-prod.s3.amazonaws.com/e9f83f72-a81b-42c7-af44-4e35b48b20b7) by Python for Data Science
- Jupyter Notebook documentation (https://jupyter-notebook.readthedocs.io/en/stable/)
- A gallery of interesting Jupyter Notebooks (https://github.com/jupyter/jupyter/wiki/A-gallery-of-interesting-Jupyter-Notebooks)
- Markdown syntax (https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet) if you are interested in incorporating Markdown blocks (such as this one) into your own Jupyter Notebooks.

Data

- The data in this lab is from Hendry et al 1999 (https://www.jstor.org/stable/pdf/3546699.pdf?casa_token=ap-R7uPGOaAAAAAA:HFkWyeP0MmkcSrjYFP3Xtz-C5JHAFicrXA7atRBAck3AhiMVWFwKWuj-bhyfL3PEDTBGKitQpy1R_Ym4NKZjFYSbFgICllBwwaJBMHze_q3GQ_poOg) and was edited for teaching purposes.

Lab 1 Part 1: Review of Python data types

There are several basic data types we can work with in Python. This week we'll look at some of the most common types of data - comments, numbers, strings, variables, and lists. Each of these types of data have specific rules for how Python can understand and manipulate the data. For example, strings and lists can be evaluated using the command `len()` to find the length. However, Python cannot use `len()` to find the number of digits in a number.

Comments

Comments are like notes in the margins of your wet lab notebook. Use comments to describe what a piece of code is doing and why, so that other people (or future you!) can more easily understand your code. Although comments are ignored by Python, they are crucial for making sure that your code is legible and useful for research.

Exercise 1: Add a comment with your name and today's date to the bottom line of the block below:

In [1]:
```
# Any line of code preceded by a pound sign "#" is a comment.
# Comments can span
# multiple lines.

# ELEANOR LUTZ, March 21 2019
```

Numbers

Most data you work with will be numbers. Python supports whole and decimal numbers, and numbers can be added, subtracted, multiplied, etc.

In [2]:
```
# Numbers can be evaluated in Python using standard order of operations.
(2 + 3) * 4.14
```

Out[2]:
```
20.7
```

Exercise 2: Evaluate the sum of 15.756, 12.445, and 90.265.

In [3]:
```
15.756 + 12.445 + 90.265
```

Out[3]:
```
118.46600000000001
```

Strings

If biology data are not numbers, they may be strings, or characters used to represent text. Strings can represent sample names, locales, DNA sequences, etc. Python strings must be enclosed by single or double quotes.

Using Python, we can look at some interesting characteristics of a piece of string data. For example, we can see whether or not something we are interested in is contained within the string. Here we would like to evaluate whether or not "AUG", a start codon sequence marking the beginning of a protein, is present in this long string of RNA text data.

Note: Python will print either `True` or `False` to answer this question. `True` and `False` are another type of data called booleans.

In [4]:
```
"AUG" in "AUGUGCACGUCGAUCCCGACCGGCGCGCGAUCGCAUCGCUUUUCGGAAAUCGAUCGAUUA"
```

Out[4]:
```
True
```

In [5]:
```
# Strings have lengths, and we can use the command len() to return the l
len("AUGUGCACGUCGAUCCCGACCGGCGCGCGAUCGCAUCGCUUUUCGGAAAUCGAUCGAUUA")
```

Out[5]:
```
60
```

Exercise 3: A RNA codon is a sequence of 3 RNA nucleotides. Find out how many 3-letter codons are in this particular piece of RNA.

In [6]:
```
len("AUGUGCACGUCGAUCCCGACCGGCGCGCGAUCGCAUCGCUUUUCGGAAAUCGAUCGAUUA") / 3
```

Out[6]:
```
20.0
```

Variables

A variable is a nickname for a value, list, or dataset. In Python, we can assign a value to a variable name using the `=` equals sign. For example, to assign the value 25.2394857182 to a variable `weight_mg`:

In [7]:
```
weight_mg = 25.2394857182
```

From now on, whenever we use the name `weight_mg`, Python will substitute the assigned value, 25.2394857182.

In [8]:
```
weight_mg + 5
```

Out[8]:
```
30.2394857182
```

Variables are useful for reducing typing errors. In the previous example, it would be tedious to retype or paste `25.2394857182` every time we wanted to run a different calculation. Instead, we can use the variable nickname `weight_mg`, which is easy to type and conveys something useful about what the data means.

In Python, variable names:

- can include letters, digits, and underscores
- cannot start with a digit
- are case sensitive.

This means that, for example:

- `location_0` is a valid variable name, but `0_location` is not
- `DNA` and `dna` are different variables

It is best practice to pick a name that describes something useful about the variable. For example, if I wanted to create a variable to store the value of room temperature in Kelvin, I might name it something like: `room_temp_K`.

In [9]:
```
# In the previous "string" lab section, we had to copy and paste our RNA
# to ask two different questions.
# Let's use a variable so we only need to input our data once:

test_RNA = "AUGUGCACGUCGAUCCCGACCGGCGCGCGAUCGCAUCGCUUUUCGGAAAUCGAUCGAUUA
```

In [10]:
```
"AUG" in test_RNA
```

Out[10]:
```
True
```

In [11]:
```
len(test_RNA)
```

Out[11]:
```
60
```

Exercise 4: Using the variable `test_RNA`, find out again how many 3-letter codons are in this RNA sequence.

In [12]:
```
len(test_RNA)/3
```

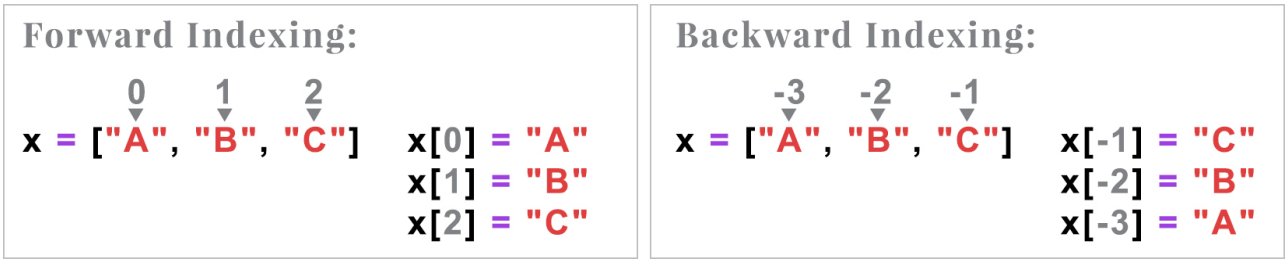
Out[12]:
```
20.0
```

Lists

A python list is a collection of multiple objects, such as numbers or strings, gathered into one entity. Lists are enclosed by brackets `[]`, and each item in the list is separated by a `,`. Items in a list do not all have to be of the same type - strings, booleans, floats, and many others can coexist in the same list.

In [13]:
```
example_list = [123.456, "hello", True, 2]
```

To retrieve items we want from a list, we use a process called list indexing. In python, each item in a list is assigned an *index value*, which describes where to find that specific item. Python lists have two different indexing systems: *forward indexing* and *backward indexing*, to allow us to count from both the front and back of the list easily.



In [14]:
```
# Print the first item in the list by referencing its index value, 0
print(example_list[0])
```

```
123.456
```

The data below is adapted from *Hendry et al 1999* (https://www.jstor.org/stable/pdf/3546699.pdf?casa_token=ap-R7uPGOaAAAAAA:HFkWyeP0MmkcSrjYFP3Xtz-C5JHAFicrXA7atRBAck3AhiMVWFwKWuj-bhyfL3PEDTBGKitQpy1R_Ym4NKZjFYSbFgICllBwwaJBMHze_q3GQ_poOg), and lists the body masses of a subset of female sockeye salmon sampled in Pick Creek, Alaska.

Exercise 5: Print the mass of the 1st, 12th, and the last salmon to be weighed in this research study.

In [15]:
```
masses = [3.09, 2.91, 3.06, 2.69, 2.88, 2.98, 1.61, 2.16, 1.56, 1.76, 1.

# Your code goes here
print("Mass of 1st salmon:", masses[0])
print("Mass of 12th salmon:", masses[11])
print("Mass of last salmon:", masses[-1])
```

```
Mass of 1st salmon: 3.09
Mass of 12th salmon: 3.3
Mass of last salmon: 1.69
```

Using Python, we can quickly find useful attributes of a dataset, such as the number of salmon in the study, or the total weight of all salmon.

In [16]:
```
# len() returns the length of a list

number_of_salmon = len(masses)
print("Number of salmon is:", number_of_salmon)
```

```
Number of salmon is: 15
```

In [17]:
```
# sum() returns the sum of values in the list

sum_of_salmon_weights = sum(masses)
print("Sum of salmon weights is:", sum_of_salmon_weights)
```

```
Sum of salmon weights is: 35.379999999999995
```

In [18]:
```
# Lists can also be modified to include new values.
# Here we are adding another salmon to the dataset:
masses.append(1.44)

# When we inspect the list, we now see that the
# new salmon has been added to the end of the list.
print(masses)
```

```
[3.09, 2.91, 3.06, 2.69, 2.88, 2.98, 1.61, 2.16, 1.56, 1.76, 1.79, 3.3
, 1.91, 1.99, 1.69, 1.44]
```

In [19]:
```
# Like strings, you can also search for the existence of individual elem
2.91 in masses
```

Out[19]:
```
True
```

In [20]:
```
# Lists can also be sorted by value...
masses.sort()
print(masses)
```

```
[1.44, 1.56, 1.61, 1.69, 1.76, 1.79, 1.91, 1.99, 2.16, 2.69, 2.88, 2.9
1, 2.98, 3.06, 3.09, 3.3]
```

In [21]:
```
# Or reversed
masses.reverse()
print(masses)
```

```
[3.3, 3.09, 3.06, 2.98, 2.91, 2.88, 2.69, 2.16, 1.99, 1.91, 1.79, 1.76
, 1.69, 1.61, 1.56, 1.44]
```

Lab 1 Part 2: Conditional logic in Python

To analyze data, we need to test individual data values. For instance: Is this number greater than a biologically important threshold? Does this value match our predictions? Logical operators are used to see if these statements are `True` or `False`.

Logical operators

Logic tests depend on several "operators" which are mostly the same across all programming languages.

- Equal to: `==`
- Not equal to: `!=`
- Greater than: `>`
- Less than: `<`
- Less than or equal to: `<=`
- Greater than or equal to: `>=`

Logical tests return the boolean values `True` or `False` to answer the logical test.

In [22]:
```
# Test for equality between an arithmetic statement and a value.
2 + 2 == 4
```

Out[22]:
```
True
```

Exercise 6: Run a logical test to see if 5.01 is greater than or equal to 6.

In [23]:
```
5.01 >= 6
```

Out[23]:
```
False
```

If statements

We can ask Python to make different choices based on the results of a logical test. This is called an `if` statement. In a Python `if` statement:

- The first line begins with the command `if` , followed by a logical test, followed by a `:`
- The subsequent lines describes what should happen if the statement is true.
- The subsequent lines are always indented one tab more from the original if statement. This formatting style indicates to Python that the indented code should be run only when the if statement is true.

In [24]:
```
temperature_F = 125

if temperature_F > 115:
    print("Excessive heat warning!")
```

```
Excessive heat warning!
```

If the logical test in an `if` statement is false, Python will ignore the indented block of code following the `if` statement. However, everything else in the code will still run as normal:

In [25]:
```
print("We are printing something before the if statement.")

temperature_F = 101

if temperature_F > 115:
    print("Excessive heat warning!")

# This code is no longer indented, so it is not part of the if statement
print("We are printing something after the if statement.")
```

```
We are printing something before the if statement.
We are printing something after the if statement.
```

elif and else

We can also chain several tests together using the commands `else` and `elif` (short for else if). For example:

In [26]:
```
temperature_F = 101

if temperature_F > 115:
    print("Excessive heat warning!")
elif temperature_F < 32:
    print("You need a jacket to go outside today!")
else:
    print("It's not too hot or cold today.")
```

```
It's not too hot or cold today.
```

Exercise 7: Write an `if` statement that fulfils the following requirements:

- if the dam water level is above 50, print a flood warning
- if the dam water level is below 5, print a drought warning
- otherwise, print that nothing is wrong

Make sure to run the code several times with different `dam_level` values (e.g. 60, 0, and 20) to make sure your code works as expected.

In [27]:
```
dam_level = 20

# Your code goes here
if dam_level > 50:
    print("Flood warning!")
elif dam_level < 5:
    print("Drought warning!")
else:
    print("Nothing is wrong!")
```

```
Nothing is wrong!
```

Logical tests can be combined

We can use `and` to require multiple logical tests to be true. We can also use `or` to accept one or more tests as true.

An example of a test with an `and` statement:

We predict that an aurora will probably not be visible unless the sky is clear *and* solar activity is high. This code runs a logical test for these two questions, and depending on the results, makes different choices in what to print.

In [28]:
```
cloudy = False
solar_activity = 172.0

if cloudy == False and solar_activity > 150:
    print("Conditions may be good for seeing an aurora tonight")
else:
    print("It seems unlikely that you will see an aurora tonight.")
```

```
Conditions may be good for seeing an aurora tonight
```

An example of a test with an `or` statement:

There are 3 possible RNA stop codons that mark the end of a protein sequence: UAG, UAA, *or* UGA. We want to see if any one of the three are present in an RNA sequence, since all 3 stop codons serve the same function in biology.

In [29]:
```
RNA = "AUGUUGAGGGGUAGUAGUAGUGGGUGUAGUGAUCCCCCCCCCCCGUGUCGCUCGAUCGUAACCUA"

if "UAG" in RNA or "UAA" in RNA or "UGA" in RNA:
    print("There is at least one stop codon present in this RNA data.")
else:
    print("There is no stop codon in this RNA data.")
```

```
There is at least one stop codon present in this RNA data.
```

Exercise 8: Both excessively high or low blood pressure can be dangerous. Edit the code below to print a warning if `patient_blood_pressure` is either above 140, *or* below 50. Make sure to change the value of `patient_blood_pressure` to several different values to make sure your code works as expected.

In [30]:
```
patient_blood_pressure = 100

# Your code goes here
if patient_blood_pressure > 140 or patient_blood_pressure < 50:
    print("Dangerous blood pressure levels!")
else:
    print("Normal blood pressure levels")
```

```
Normal blood pressure levels
```

An example of a test with both an `and` and `or` statement:

Tests can combine `and` and `or` statements together. For example, some animals have distinct fur colors adapted to the animal's environment. Here we are studying a group of rabbits living on the edge of the forest and the beach. We would like to test our rabbits to see if the fur color matches what we expect for its environment:

In [31]:
```
location = "beach"
color = "black"

if (location == "beach" and color == "white") or (location == "forest" a
    print("This rabbit has the expected color for its habitat location."
else:
    print("There is something unexpected about this rabbit.")
```

```
There is something unexpected about this rabbit.
```

Exercise 9: Write an `if` / `else` statement to check if a plant sample belongs to "Species X." A plant is a member of "Species X" if both of these statements are true:

- Its leaf length is under 1mm *or* over 3mm
- Its leaf color is either green *or* red

In [32]:
```
leaf_color = "green"
leaf_length_mm = 1.56

# Your code goes here
if (leaf_color == "green" or leaf_color == "red") and (leaf_length_mm <
    print("This plant belongs to Species X")
else:
    print("This plant does not belong to Species X")
```

```
This plant does not belong to Species X
```

Lab 1 Part 3: Looping over data in Python

Computers are extremely useful for repeating tasks over and over. In Python, a `for` loop asks Python to *do the exact same thing many times*. You might ask Python to run the same probability test 1000 times, or you might want to create the same graph for 100 different data samples.

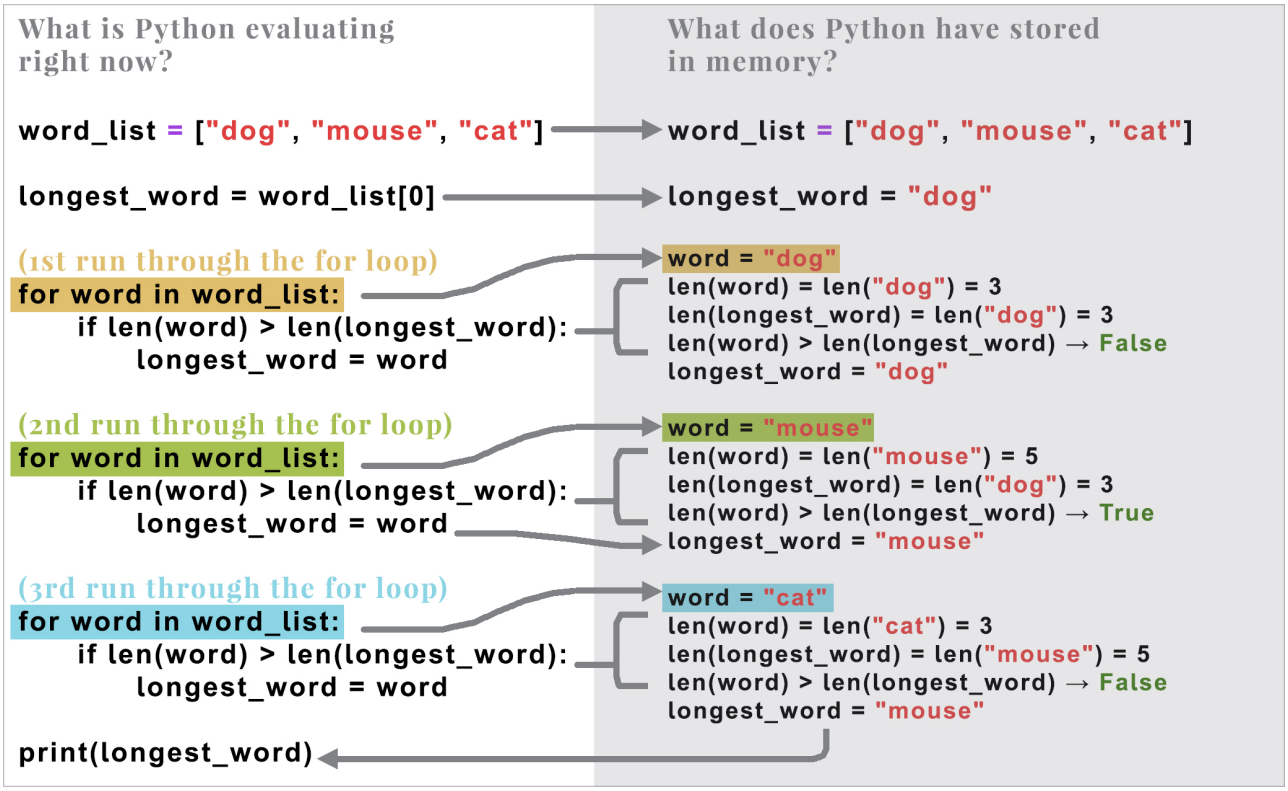
Similar to an `if` statement, a `for` loop begins with the word `for`, followed by a statement describing what we should evaluate, followed by a `:`. The subsequent indented lines describe what should happen during each repeated evaluation of the code.

In class we worked on using a `for` loop to find the maximum value in a dataset. Let's walk through an example of similar code that finds the longest word in a list of words:

In [33]:
```
word_list = ["dog", "mouse", "cat"]

longest_word = word_list[0]

for word in word_list:
    if len(word) > len(longest_word):
        longest_word = word

print(longest_word)
```

```
mouse
```

The command `for word in word_list:` asks Python to run all of the following indented lines of code once, **for each word in the variable word_list**. As Python loops through a for loop, it can make calculations, update variables, print statements, and do other actions. In this example, here is what Python is doing during each iteration of the loop:



Exercise 10: Using a `for` loop, print every item in `vocabulary` that is longer than 5 letters long.

In [34]:
```
vocabulary = ["mosquito", "bee", "spider", "albatross"]

# Your code goes here
for word in vocabulary:
    if len(word) > 5:
        print(word)
```

```
mosquito
spider
albatross
```

Lab 1 Part 4: Introduction to libraries

In class this week we used the command `import numpy as np` to create an **array** - a special kind of data structure that can only store values of the same data type (remember that a Python list can store many different kinds of data types, such as strings and numbers).

Libraries like `numpy` are a collection of code written by other people. By importing a library, you can use all of *their* code in addition to the code you write yourself. There are thousands of useful Python libraries you can use, and you can even write your own libraries to share with other scientists. In this class we'll learn to use several of the default libraries including `numpy` (which contains code to efficiently work with matrices, linear algebra, and random numbers).

For this lab, we will use another library called `matplotlib` as an example. Matplotlib is a plotting library - it helps you make figures and plots quickly. Matplotlib is a very large library, so it has some subsections that we can reference. For this lab we want to import a specific subsection of the `matplotlib` library called `pyplot`. To do this we can import the library subsection using the command `import matplotlib.pyplot`. Then we can use code kept within the library subsection by using the keyword `matplotlib.pyplot`.

However, `matplotlib.pyplot` is a very long phrase to have to type multiple times. Similar to variables, we can give libraries nicknames as well, by using the phrase `as` right after importing the library. For example, if we type `import matplotlib.pyplot as plt`, we can use the nickname `plt` to reference `matplotlib.pyplot` instead of typing out the full name of the library.

You'll notice in the code block below, where we import the `matplotlib.pyplot` library, there is also another line of code: `% matplotlib inline`. This is a snippet of code called a **magic command**. This particular magic command tells your Jupyter Notebook to show your plots and figures inside of the Jupyter Notebook itself, instead of saving them elsewhere. We will be using this code snippet throughout this class to quickly see what we are plotting within our Jupyter Notebook.

In [35]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

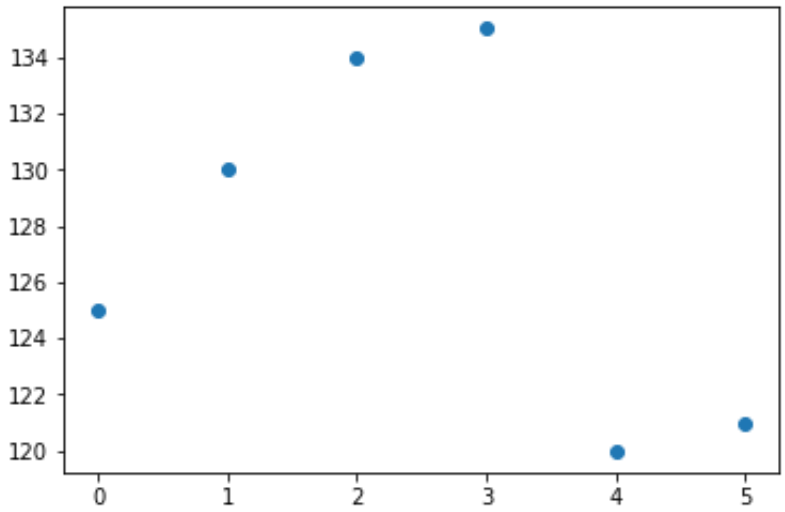
Now that we have imported this matplotlib pyplot library, we can use any of the code commands contained within it. To reference these commands, we use the name (or nickname) of the library - in this case, `plt` - followed by a `.`, followed by the command we want to use. Each library comes with its own set of useful commands, and you will need to read the documentation of each specific library to understand what is possible.

For this lab, we'll first use the matplotlib command `scatter`, which is a command that creates a scatterplot of a given set of data.

Exercise 11: Look at the output of the following code, and try to understand how the `plt.scatter()` command works. Which variable goes on the x axis - the first variable given to the `plt.scatter()` command, or the second?

In [36]:
```
time = [0, 1, 2, 3, 4, 5]
weight = [125, 130, 134, 135, 120, 121]

plt.scatter(time, weight)
```

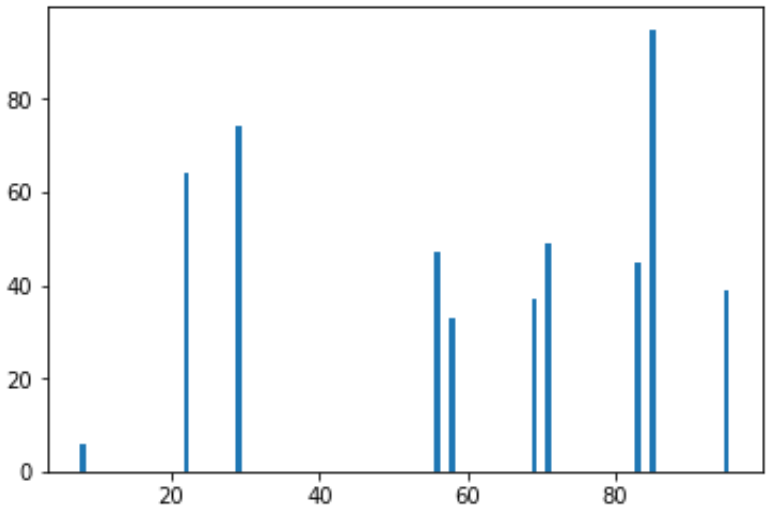
Out[36]:
```
<matplotlib.collections.PathCollection at 0x7ff3b20361d0>
```



Matplotlib can also create bar plots using the command bar()...

In [37]:
```
x = [58, 56, 71, 85, 83, 95, 29, 8, 69, 22]
y = [33, 47, 49, 95, 45, 39, 74, 6, 37, 64]

plt.bar(x, y)
```

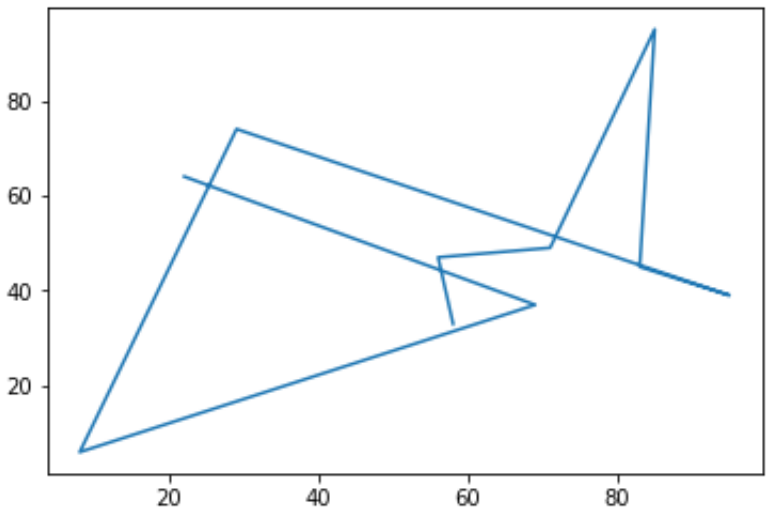
Out[37]:
```
<Container object of 10 artists>
```



Or a line plot using the command plot():

In [38]:
```
x = [58, 56, 71, 85, 83, 95, 29, 8, 69, 22]
y = [33, 47, 49, 95, 45, 39, 74, 6, 37, 64]

plt.plot(x, y)
```

Out[38]:
```
[<matplotlib.lines.Line2D at 0x7ff3b1ee5208>]
```



Lab 1 Bonus exercises

Bonus Exercise 1: To create a string variable, we enclosed the text in quotes, like this: `x = "hello"` . But what if you want to create a variable that includes quote characters? Declare this sentence as one variable: `"Begin at the beginning," the King said, very gravely, 'and go on till you come to the end: then stop.' - Lewis Carroll`

In [39]:
```
text = "\"Begin at the beginning,\" the King said, very gravely, 'and go

print(text)
```

```
"Begin at the beginning," the King said, very gravely, 'and go on till
you come to the end: then stop.' - Lewis Carroll
```

Bonus Exercise 2A: The symbol `%` , also called modulo, is a logical operator that returns the remainder in a division operation. Use the internet to learn more about using modulo in python, then use `%` and an `if` statement to print whether or not x is an even number.

In [40]:
```
x = 127

# Your code goes here
if x % 2 == 0:
    print(x, "is an even number")
else:
    print(x, "is not an even number")
```

```
127 is not an even number
```

Bonus Exercise 2B: We are interested in analyzing only the RNA that encodes a protein sequence. This means that the number of letters, or nucleotide bases, in the data should be divisible by three (the number of nucleotides per amino acid codon). Does our data contain errors? Use `%` to see how many extraneous letters are present in the data (if any).

In [41]:
```
RNA = "AUGUUGAGGGGUUAGUAGUGAGUGGGUGUAGUGAUCCCCCAUCCCCCGUGUCGCUCGAUCGUAAC

# Your code goes here
print("Number of extra nucleotides is:", len(RNA) % 3)
```

```
Number of extra nucleotides is: 2
```

Bonus Exercise 3: In Part 2 of this lab, we looked at a scenario in which we updated one variable and looked at the results on another variable that used the first:

In [42]:
```
x = "AUGUGCA"
y = len(x)
x = "A"

print("The length of x is:", len(x))
print("The value of y is:", y)
```

```
The length of x is: 1
The value of y is: 7
```

Here is a different example of this scenario, where we are assigning `y` to equal `x` without making any changes. Do you notice anything unexpected when comparing these outputs? Use the internet to try and understand what may be causing these differences.

In [43]:
```
x = [1, 2, 3]
y = x
x[0] = "A"

print("The value of x is:", x)
print("The value of y is:", y)
```

```
The value of x is: ['A', 2, 3]
The value of y is: ['A', 2, 3]
```

Answer to 3:

When we typed the command `y = x`, we were intending to make a *copy* of the original variable `x`. But in Python, when we set a variable to `=` another variable, it does not actually make a copy of the original variable. Instead, it creates a new reference (or shortcut) that remembers the association between the two variables.

In [ ]:
```
```