Recitation02

September 2, 2020

0.1 Python 3

Python 3 is a highly expressive programming language. You can use Python to write state-of-the-art machine learning code, run a large network of web servers, or use it as a calculator. One of the aims of the inventors of Python was to write a programming language with an intuitive **syntax**—a set of words, operators, and structure considered correct.

Computers don't speak the language as humans. They speak in machine code, long sequences of Binary code (think of the movie The Matrix). How programming languages translate between human input, what we type into our computer, and machine code, what the computer understands, is called the **Programming Process**. The majority of languages approach this process by either being **interpreted** or **compiled**.

An **interpreted** language translates human code into machine code line by line, sometimes called "on the fly". A major advantage to interpreted languages, vs compiled, is how much easier it is to run these programs. The user does need to complete any additional steps beyond writing their code. On the downside, interpreted languages tend to be a bit slower than their compiled counterparts. Python 3 is an interpreted language.

Some programming languages require you to predefine resources for all the variables you plan to create. This is called **static typing**. Python 3 is not one of these languages. Instead, Python 3 figures out the resources you'll need from your code—called **Dynamic typing**—and makes it much faster to write code.

The goal of an interpreted, Dynamically typed language is to trade the time it takes to run a program (longer with Python 3) with the time it takes to write the program itself (slower with a compiled, statically typed language).

Our goal will be to learn how to
* use fundamental objects in Python 3 * write loops * build functions * import modules * write a simple algorithm for taking a simple random sample.

0.2 Fundamental objects in Python 3

0.2.1 Variables

Variables are how we reserve space in the computer for numbers or strings. When you create a variable, the computer allocates a unique space in it's memory to store this variable's data.

To create a variable, to assign space to a number or string, we write `<the variable name> = <the data to store>`. For example, I'd like to create a variable called `x` and assign it the value 5. All I need to do is type

[106]:
```
x=5
```

You can store to types of numerical information in Python 3: floats or integers. By default Python 3 considers numbers integers—numbers with no fractional part. A **floating-point number**, called a **float** for short, is a format for specifying positive and negative decimal numbers in a computer. If the variable you've created has the potential to be positive, negative, or have a decimal, make sure it is a float. To assign a variable to a float include a period in the number. Here are some examples.

[107]:
```
x=5.            # float
x=5             # integer
y = 32.45       # float
PHDSI = -0.321 # float
```

To assign a set of characters to a variable, enclose them in either single or double quotes. Like this `<the variable name> = "the string"` or like this `<the variable name> = 'the string but with single quotes'`. Here are a few examples

[108]:
```
hereIsAString = "PHDS-I Rulez"
anotherString = 'Banjo the Science Dog!'
```

0.2.2 Lists

Lists are ordered sequences of numbers or strings that can be accessed and changed. To create variable that stores a list, enclose the numbers or strings you want in the list with square brackets and separate them with commas. For example, suppose I want a list called `randomList`of the numbers: 1, 2, 3 and then the string "Kazoo the Science Dog". Then I would write

[109]:
```
randomList = [1,2,3,"Kazoo the Science Dog"]
```

Lists are a ways to store multiple pieces of information and associate them with a single location in your computer, a variable. You can access the items in your list using **indexing**. To access the first item in your list, you type `<the variable name>[the ordered number of what piece of information you want]`. For example, lets assume I want the 2nd item in my `randomList`. I would type

[110]:
```
randomList[1]
```

[110]: 2

UMMMMMMM, WHAT! I wanted the second item in my list (the value 2) but I didn't type `randomList[2]`. Instead I typed `randomList[1]`. You did not forget how to count. Python does not start counting at 1 like us humans like to do. Python starts counting at 0 (my favorite number, thanks Python).

To access data you've stored in a list, you do not need to use positive numbers as your index, you can use negative numbers to. If I wanted the last item in my list I could type either

[111]:
```
randomList[3]
```

[111]:
```
'Kazoo the Science Dog'
```

Or I could have typed

[112]:
```
randomList[-1]
```

[112]:
```
'Kazoo the Science Dog'
```

You can also access a range of information by typing the first index, a colon, and the last index. But beware! Python will only include the information at indices up to, **but not including**, the last index.

[113]:
```
randomList[2:4]
```

[113]:
```
[3, 'Kazoo the Science Dog']
```

The range of indices written above were: 2, 3, and 4. But Python only returned the info at indices 2 and 3. If you want to include that last item it is common practice to write

[114]:
```
randomList[2:4+1]
```

[114]:
```
[3, 'Kazoo the Science Dog']
```

Another important property of lists is that they can be modified. You can add to the end of a list by appending the value you'd like to add to the end of the list

[115]:
```
randomList.append( "A string I want to include at the end of my list" )
print(randomList)
```

```
[1, 2, 3, 'Kazoo the Science Dog', 'A string I want to include at the end of my
list']
```

Or you can change the information in any of the current indices that store information. If I wanted to change the information in the 1st slot of the list (the value 1) with the value 77 then I can write

[116]:
```
randomList[0] = 77 # remeber we start counting at zero!
print(randomList)
```

```
[77, 2, 3, 'Kazoo the Science Dog', 'A string I want to include at the end of my
list']
```

0.2.3 Tuples

A tuple acts a lot like a list except for one major difference: once you create a tuple, you cannot modify it. To create a tuple enclose information separated by commas by a left and right parenthesis. Like this,

[117]:
```
aTuple = (0.4,1.2,4.5)
```

You can still access particular items just like you can with lists.

[118]:
```
print(aTuple[1])
```

```
1.2
```

but watch what happens when I try to change an item in a tuple

[119]:
```
aTuple[1] = 7
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-119-8ac4ce701461> in <module>
----> 1 aTuple[1] = 7


TypeError: 'tuple' object does not support item assignment
```

0.2.4 Tuples vs lists

Though you can use either a tuple or list, it is a convention to use lists for homogenous data and tuple for inhomogeneous data. For example, suppose you wanted to track the last x, y, and z coordinates of a wild bird population. Because each coordinate refers to a different dimension in space a tuple would be a better fit.

[ ]:
```
birdLocation00 = (  2.3, 4.5 , 0.9)
birdLocation01 = ( -2.3, 44.5, -0.9)
birdLocation02 = (  4.3, 14.5, 0.99)
```

If instead you wanted to track the amount of worms they eat for distinct meals (measured in grams) it may be best to use a list. This is because the data all refers to the same type of information: grams of food gobbled up.

[ ]:
```
birdMeals00 = [430, 222, 450, 233]
birdMeals01 = [430, 100]
birdMeals02 = [130, 22, 330]
```

0.2.5 Sets

Like a list and tuple, a set is a collection of information. But unlike a list or tuple a set is unordered and only unique items can be stored in a set.

You can create a set by enclosing data separated by commas inside a left and right curly bracket.

[ ]:
```
aSet = {1,1,2,3,5,8,13,21}
print(aSet)
```

But notice what happened with the set created above. Even though two "ones" were included in the set, the set that was created only included a single one—items in a set are unique.

The advantage in Python of a set are set operations. With two sets A and B you can perform

[ ]:
```
A = {'a','b','c'}
B = {'a','c','d'}
```

Set Union

[ ]:
```
print(A|B)
```

Set Intersection

[ ]:
```
A & B
```

Set Difference

[ ]:
```
A-B
```

0.2.6 Dictionaries

Dictionaries create a list of pairs, or associations, between two pieces of information. The first piece of information is called a **key** and the associated piece is called a **value**.

A dictionary is created by placing a colon (:) between each key, value pair, separating these pairs by commas, and enclosing these pairs by curly brackets.

Suppose I wanted to link each bird's number from the example above with their coordinates and their meals. I could create a dictionary.

[ ]:
```
birdData = { 1: [(2.3, 4.5 , 0.9)   , [430, 222, 450, 233] ]
           , 2: [( -2.3, 44.5, -0.9), [430, 100] ]
           , 3: [(  4.3, 14.5, 0.99), [130, 22, 330]  ]
           }
```

You can access elements in dictionaries by referring to their keys. For example, if we want data from Bird number 2 we can write

[ ]:
```
print(birdData[2])
```

and if we only wanted the meals they have had, we could further index our dictionary

[ ]:
```
print(birdData[2][1])
# Why can I do this? Hint: What type of Python object did I assign to the number␣
 ↪2 in this dictionary?
```

0.3 Loops

Loops allow us to "copy and paste" lines of code with a similar structure, replacing certain aspects of those lines with values from a list, tuple, or dictionary. The two most common types of lists are "For loops" and "While loops"

0.3.1 For loops

You can use a For loop to run through values in a list:

[ ]:
```
for x in [1,2,10,'Fiddle']:
    print(x)
```

The For loop above ran through values in the list `[1,2,10,'Fiddle']`, assigned each value in order to the variable `x`, and then executed the code "inside" the For Loop. In this case the only code to execute was `print(x)`.

Here is a second example where we do a bit more with each item in the list.

[ ]:
```
for x in [10,4,3,23]:
    y = x-10
    print(y)
```

We see that each item in the list `[10,4,3,23]` is assigned the variable `x`. Then we create a variable `y` that is `x-10` and print the value of `y`.

We can also loop through tuples

[ ]:
```
for foodAmountsInGrams in (100,23,343,10):
    foodAmountsInLbs = foodAmountsInGrams*0.00220462
    print(foodAmountsInLbs)
```

Looping through dictionaries is a bit different. To use a For loop with a dictionary, you need to provide two variables in the loop: one variable for the key and a second variable for the value. You also need to add `.items` to the end of the dictionary.

[ ]:
```
for (birdNumber,birdCoordsAndFeed) in birdData.items():
    print(birdNumber)
    print(birdCoordsAndFeed[0])
```

But what is `birdData.items()` doing? When you append the code `.items()` to any dictionary, a list is created that contains two pieces of information: a key and the corresponding value.

[ ]:
```
print(birdData.items()) # notice how the below code encloses everything in [], a␣
↪list!
```

This is not the only way to loop through a dictionary. You can also loop through just the keys

[ ]:
```
for birdNumber in birdData:
    print(birdNumber)
```

As a recap of For loops, before we move to While loops, a For loop has the following structure.

```
for <variable> in <Python Object>:
    Multiple lines of code
    to run
    for each value in <Python Object>
```

0.3.2 While Loops

While loops have a similar structure to For loops. The key difference is that a While loop is finished after a specific condition is met whereas a For loop is finished after it runs through the last value in your list, tuple, or dictionary.

Let's look at an example of a While Loop.

[ ]:
```
x = 0
y = 0
while x < 4:
    y = y+10 # create a variable y and assign to it the value of y in memory␣
 ↪plus an additioanl "10"
    x = x+1  # create a variable x and assign to it the value of x in memory␣
 ↪plus an additional "1"
print(y)
```

A while loop has the following structure

```
while <condition>:
    Multiple lines of code
    to run
    for each value in <Python Object>
```

and once the `<condition>` is met the loop finishes.

0.3.3 If else

One very common conditional structure in any programming language, including Python, is the if-else statement. The if-else statement executes lines of code dependent on whether a list of conditions are met or not.

The structure looks like this

```
    if <some condition>:
        indented lines of code
        to
        run if the condition is True
    elif <some other condition>:
        indented lines of code
        to
        run if the first condition was False and the above condition was True
    elif <some other condition>:
        indented lines of code
```

```
        to
        run if the above two condition were False and this one is True
    else:
        indented lines
        of code
        to run if none
        of the above statements were true
```

If-else statements are a way to introduce structure and logic into your program. For example, if I wanted to pick out birds whose first meal was greater than 400 grams I could write

[124]:
```
birdNumberWhoAteMoreThan400Grams = []                    # This is an EMPTY list
for (birdNum, coordinatesAndMeals) in birdData.items():  # look at all the code␣
 ↪we're writing already!
    meals = coordinatesAndMeals[1]                       # extract the second␣
 ↪entry in our list, the meals.
    if meals[0] > 400:                                   # is the first meal␣
 ↪they ate more than 400?
        birdNumberWhoAteMoreThan400Grams.append(birdNum) # if yes then append␣
 ↪their number to a list
print(birdNumberWhoAteMoreThan400Grams)
```

```
[1, 2]
```

0.3.4 Functions

Functions are pieces of code that take inputs and return outputs, much like a mathematical function. In Python functions are created with the keyword `def` and include the keyword `return`. Functions are essential components of any program in any language and allow you to run the same piece of code for different inputs.

The structure of a function is

```
def <name of your function>(<inputs separated by commas>):
    lines of code
    that
    are indented
    return <the Python object, or information, you want to return>
```

An example of a function could be to count the number of observations that fall inbetween two numbers. The input would be the observations, and the two numbers to fall between, and we would return the number of observations.

[127]:
```
def countObsBtw2Numbers(observations, firstNum,seconNum):
    count = 0
    for obs in observations:
        if firstNum <= obs < seconNum: # an example of a condition to be met.
            count = count+1
    return count
```

We can then generate some observations and apply our function to them.

[130]:
```
import numpy as np
OneHundredRandomObservations = np.random.normal(0,1,100)  # dont worry, this is␣
 ↪the next part of recitation.

obsBtwNeg1AndPos1 = countObsBtw2Numbers( OneHundredRandomObservations, -1, 1)
print(obsBtwNeg1AndPos1)
```

```
65
```

In the code above, we first generated 100 random observations. Next we applied our function `countObsBtw2Numbers` which took three inputs: * The observations we want to count * The first number observations need to be greater than or equal to * The second number observations need to be strictly less than

Our function returns an integer called `count` and in our code we assigned count to a variable called `obsBtwNeg1AndPos1`.

0.4 Modules

In the code there was a line we didn't talk about yet. It looked like this `import numpy as np`. This is an example of a module—a computer program that is a (usually large) list of functions.

Modules can be downloaded from the web and installed in Python, and many modules we'll use are pre-installed. To access a module you use the `import` keyword. The structure for importing a module looks like this

```
import <name of module> as <an optional way to refer to the module>
```

In the code above we imported a module named **NumPy** an enormous list of functions for numerical computing in Python. Inside Numpy (this is the same for other modules) are several different programs that contains lists of functions. You can access a list of functions by placing a dot between the module and list of functions you want to access.

In our example we wanted to choose a subset of functions in the `random` list under the NumPy module so we wrote `np.random.<the function inside random which is inside Numpy>`.

0.5 One attempt at taking a SRS of a population.

[ ]:
```

```