Day3_Loops_and_if_Statements

July 22, 2021

Day 3: Introduction to loops and if statements in Python

Yesterday, went over variables, different data types and strings. We learned about indexing with lists and using keys to call information from a dictionary. These are the important basics for coding languages, and we will be building off of them today.

Today, we will be going over these tools that make automating your research easier. These tools are similar across coding languages as well. The first thing we will go through is for loops, which allow you to perform a process many times very quickly. The next thing we will be going through is if statements. These allow you to check conditions of interest, and then perform actions depending on whether they are true or false.

Goals for the day:

- Go over an introduction to for loops
- Begin using if/else statements for multiple conditions
- Learn elif statements
- Use dictionaries with if/else statements
- Check multiple conditions using and and or
- Learn how to nest if statements in a for loop
- Review floating point numbers

Functions Learned:

- Create a for loop:

`For i in range      print (output)`

- If/else statement:

`if (input):     print(output) else:     print(output)` - Elif statement:

`if (input):     print (output) elif (input):     print(output) else: print(output)`

- Test if two conditions are both true:

`if (input 1) and (input 2):     print(output)  else:     print(output)`

- Test if one condition out of two is true:

`if (input) or (input):     print(output)  else:     print(output)`

1 For loops in python

You'll often want to run through all entries in a list and perform the same action on all of them. Useful examples are if you want to perform the same math operation or print them all out.

The mechanism of a for loop is that you give it a list to run through, and then it will step through each element of that list once until the actions are finished.

The indentation on the second line is key, and tells Python what you want to do for each iteration. In this example, we create a list of fruits labelled `fruits`. We can then step through each of these elements in the list, starting with the first element of the list.

[1]:
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  print(fruit)
```

```
apple
banana
cherry
```

The mechanics of what is happening here is that each sequential element in `fruits` is being assigned to the indexer `fruit` and that is being printed out. The colon at the end of the for line tells python where to start the loop, and includes anything that is indented after that line. Now, something a little more complex:

[2]:
```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
  print("My favorite fruit is", fruit)
print('I am not in the for loop')
```

```
My favorite fruit is apple
My favorite fruit is banana
My favorite fruit is cherry
I am not in the for loop
```

[3]:
```
fruit
```

[3]:
```
'cherry'
```

One of the classic ways that scientists write for loops is to use the index `i` and then a range of values. This range is important, because it tells the `for` loop where to start and when to stop. It starts at 1 and then it goes by 1 to 10 and **does not included 10**.

[4]:
```
for i in range(1,10,1): # from 1 to 10 (not including 10), step by one
  print(i)
```

```
1
2
3
4
5
6
```

```
7
8
9
```

To visualize how `range` works, we can change to go from 1 to 100, instead with a step size of 10. So, it starts at 1, then goes to 11, and so on. It doesn't include 101 because it is outside of the range:

[5]:
```python
for i in range(1,100,10): # for 1 to 100 (not including 100), step by TEN
  print(i)
```

```
1
11
21
31
41
51
61
71
81
91
```

Now that we can see how the `i` and `range` work `for` loop works, we can use it to build a more complicated for loop. We set up an empty list `squares`, and add each new value using the function `append`:

[6]:
```python
squares = [] # this sets up an empty list

for i in range(1,10,1):
  squares.append(i**2)

print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

What is happening here, is that i is momentarily being assinged a value within range. We can then perform an operation on that value (here an exponent) and then append the value to the empty list we created.

2 Coding check in

The post doc in Katie's lab refuses to use Fahrenheit when discussing the temperatures outside in Chicago! She needs help calculating the temperature in Fahrenheit to Celsius for a range of values from -10 to 80! You can do this for every 10 degrees.

$C = (F - 32) * (5/9)$

[p.s. this is a real story].

Note: It also helps to write this out with words first, since Python pretty much works that way.

Hint: you will momentarily assign each fahrenheit value to `i`

[7]:
```
## This is where you can write your answer!

# set up empty list


# set up for loop with range (the range will be your fahrenheit values)


# print out your new array
```

2.1 Answer

[8]:
```
# set up empty array
celcius = []



# set up for loop with range
for i in range(-10,81,10):
  celcius.append((i -32) * (5/9))



# print out your new array
print(celcius)
```

```
[-23.333333333333336, -17.77777777777778, -12.222222222222223,
-6.666666666666667, -1.1111111111111112, 4.444444444444445, 10.0,
15.555555555555557, 21.11111111111111, 26.666666666666668]
```

3 If/else statements

If statements are conditional tests that allow you to interrogate values and then perform actions if they are true or false. These are handy tools for cleaning up data and performing analysis. We will start small and work our way up.

First, we will recall back to yesterdays exercise when we checked if something was true or not using the ==,>, or < sign. There are several different tests we can do. These are including

- == if something is equal to
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to
- != not equal to

[9]:
```
my_age = 27 # defining my age

my_age >= 18 # asking if this is greater than or equal to 18
```

[9]:
```
True
```

So we set our age to be some value, and then we asked Python if that value was greater than or equal to 18, and it returned either `True` or `False`. We utilize an if statement to then perform some action.

So now, we can write a if statement that will print out a statement that we can vote if we are older than 18, and we cannot vote if we are younger than 18.

[10]:
```
my_age = 10

if my_age >= 18: # if my_age is greater than equal to 18
  print("You can vote!")
else:
  print("Maybe next year!")
```

```
Maybe next year!
```

3.1 Using `and` and `or`

You may also want to check multiple conditions, which can be done using the `and` and `or` conditions. In the case of `and`, both conditions must be true to pass. In the case of `or`, only one of them needs to be true to pass.

[11]:
```
my_animal = 'shark'
your_animal = 'shark'

if (my_animal == 'shark') and (your_animal == 'shark'):
  print("We both like sharks")
else:
  print("At least one of us does not like sharks")
```

```
We both like sharks
```

3.2 Coding Check in

It's the end of the fourth quarter the Tampa Bay Buccaneers are down 28-30 against the Cleveland Browns. They are too far away for a field goal, so Tom Brady (quarterback) is going to attemp a Hail Mary to Rob Gronkowski (Tight End).

In order to get the touch down, Tom Brady needs to throw the ball at least 60 yards `and` Rob Gronkowski needs to be open. Write an `if` statement that tests whether Gronk can make the catch and prints out who wins.

First, define your variables `brady_throw` to be some number, and `gronk_open` to be `True` or `False`. Then, test those conditions using your `if` statement.

Note: It helps to write this out in words first!

[12]:
```
### Write your code for the coding check in here
```

3.2 Answer (don't check until you've tried)

[13]:
```
brady_throw = 55
gronk_open = True

if (brady_throw >= 60) and (gronk_open == True):
  print("Brady passes to Gronk for the Touch Down, Buccs win!!")
else:
  print("Brady and Gronk fail to make the pass, the Browns win the game!!")

# GO BROWNS
# there are several ways this could be written and be more complicated
```

```
Brady and Gronk fail to make the pass, the Browns win the game!!
```

4 `elif` statements

Sometimes, there will be multiple situations you want to test at the same time. In this case, we use an `elif` statement, which tests another condition.

A key note here is that Python will only go on to the next `elif` statement if it fails the test, so the ordering of the statments is very important. This is similar to making a decisions tree.

This situation might arise if you are an Illinois resident and you want to visit the Field Museum and get the all access pass so you can go to a 3D movie. There are different prices based on your age, so we need to have more than one test. We start at the youngest age and then work our way up. The last statement we have is the `else` statement, which gives all the people who didn't pass the other requirements their ticket price:

The order of events * Ask if age is less than 3, admission is **free** if it is * Ask if age is less than 11 (but older than 3 because it failed that test), admission is $27 if it is * Ask if age is older than 65 (but older than 11 because it failed that test), admission is $35 if it is * Everyone not fitting these categories, admission is $38 for the rest

[14]:
```
age = 27

if age < 3:
  print("Your admission is free") # take that baby to the Field!
elif age <= 11:
  print("Your admission is $27")
elif age >= 65:
  print("Your admission is $35")
else:
  print("Your Admission is $38")
```

```
Your Admission is $38
```

4.1 Dictionaries and `f strings` in the `elif` statement.

This code is a little clunky and you have to hard code all the prices in the `if` statement. Recalling from our dictionaries from yesterday, we can instead use a dictionary and something called an `f` string to print it out specific prices in our if statement. You can think of f as for format when thinking about the mechanics of this, it is trying to format the new string with a variable.

So we will first set up our dictionary named field_dictionary:

[15]:
```
life_stages = ["baby", "child", "adult", "senior"]
admission = [0, 27, 38, 35]

field_dictionary = dict(zip(life_stages, admission))
print(field_dictionary)
```

```
{'baby': 0, 'child': 27, 'adult': 38, 'senior': 35}
```

We can use the function get and the key child to call up the price of admission for a child:

[16]:
```
field_dictionary.get('child')
```

[16]:
```
27
```

We can now use an f string to print these values together by setting the admission_price using the dictionary and then calling that inside curly brackets in the f string.

[17]:
```
admission_price = field_dictionary.get('adult') # use the key to get the␣
 ↪admission price value

print(f"The cost of admission is ${admission_price}") # print out the correct␣
 ↪admission price for each life stage
```

```
The cost of admission is $38
```

You can even save this to a variable called new_string that will print out the string with the correct admission price.

[18]:
```
admission_price = field_dictionary.get('adult') # use the key to get the␣
 ↪admission price value

new_string= f"The cost of admission is ${admission_price}"

print(new_string)
```

```
The cost of admission is $38
```

So we can include this back in our original if statement:

[19]:
```
# create a dictionary for life stages and admission price
life_stages = ["baby", "child", "adult", "senior"]
admission = [0, 27, 38, 35]
field_dictionary = dict(zip(life_stages, admission))

# set our age we want to test
age = 10
```

```
# use our elif statements to tell us our price of admission
if age < 3:
  admission_price = field_dictionary.get('baby')
  print(f"The cost of admission is ${admission_price}")
elif age < 11:
  admission_price = field_dictionary.get('child')
  print(f"The cost of admission is ${admission_price}")
elif age >= 65:
  admission_price = field_dictionary.get('senior')
  print(f"The cost of admission is ${admission_price}")
else:
  admission_price = field_dictionary.get('adult')
  print(f"The cost of admission is ${admission_price}")
```

```
The cost of admission is $27
```

4.2 Coding check in!

You're running a pet grooming salon and you take in cats, dogs, and elephants but nothing else. Set a fair price for each of these animals using a dictionary. Set up a series of `if` and `elif` statements that tell the customer how much it will be to wash their pet. Make sure to tell them that if it's not a cat, dog, or elephant, you can't groom their animal!

[20]:
```
### space to put your code here!

## dictionary of animals and prices

# define your pet

# elif statements
```

4.2 Answer to coding check in!

[21]:
```
## dictionary of animals and prices
animals = ["cat","dog", "elephant"]
prices = [30,50,1000]

pet_dict = dict(zip(animals,prices))

# define your pet

my_pet = "zebra"

# elif statements

if (my_pet == "cat"):
  my_price = pet_dict.get('cat')
  print(f"Your price is ${my_price}")
```

```
elif (my_pet == "dog"):
  my_price = pet_dict.get('dog')
  print(f"Your price is ${my_price}")
elif (my_pet == "elephant"):
  my_price = pet_dict.get('elephant')
  print(f"Your price is ${my_price}")
else:
  print("We don't groom that pet here")
```

```
We don't groom that pet here
```

5 Using `in` to test if a value is in a list

We can also use the test `in` to see if a value or string is in a list. We must go through the list by using a `for` loop, and then test the conditions of that using an if statements. This is an example of nesting an `if` statement within a `for` loop.

Imagine you are at an ice cream sundae shop and you're requesting flavors. We want to loop through out requested toppings to see if they are available at the shop.

We first define the toppings available, then the toppings requested. Then we use a for loop to go through each of the toppings requested. If the topping in `toppings_requested` is in `toppings_available` it will print that it's adding the topping. If it is not available, it says we don't have that topping.

[22]:
```
toppings_available = ['sprinkles', 'fudge', 'whipped cream', 'bananas'] # the␣
↪toppings available at the shop
toppings_requested = ['sprinkles', 'fudge', 'pie'] # the toppings someone orders

for topping in toppings_requested: # for loop going through each of the␣
↪requested toppings
  if topping in toppings_available: # checking to see if each topping is␣
↪available
    print(f"Adding {topping}!")
  else: # if it is not available
    print(f"We don't have {topping}!")
```

```
Adding sprinkles!
Adding fudge!
We don't have pie!
```

6 Extra Coding check in

Imagine you're at Target and you want to buy a vase for a friend. Your friend is very picky about their colors, and you need to ask the store manager if she has any specific colors available.

Create a list of three colors that the store manager has (`colors_available`), and another list of three colors your friend might like (`colors_want`). Have the store manager respond with whether or not they have that color available for all the colors your friend might like.

[23]:
```python
### coding check in here

# lists of colors available and colors you want


# for loop and if statement
```

6.1 Coding check in answer

[24]:
```python
colors_available = ['red', 'blue', 'violet'] # the toppings available at the shop
colors_want = ['violet', 'pink', 'blue'] # the toppings someone orders

for color in colors_want: # for loop going through each of the requested toppings
  if color in colors_available: # checking to see if each topping is available
    print(f"We have a vase in {color}!")
  else: # if it is not available
    print(f"We don't have a vase in {color}!")
```

```
We have a vase in violet!
We don't have a vase in pink!
We have a vase in blue!
```

Mini-lesson: Floating Point

1.0 Different "types" of numbers

There are two main python types for numbers: `int` and `float`

Type `int` are integers, such as 0, 1, 2, . . . , 10, . . .

Type `float` are numbers in their decimal form, such as 0.1, 1.1, 55.000000001, . . .

`int` can be cast in `float` and vice versa.

Examples of both types are below:

[25]:
```python
x = 1

# As a sanity check, we print the type for the variable x:
print(type(x))
print("x = {}".format(x))
```

```
<class 'int'>
x = 1
```

[26]:
```python
# We cast the integer stored in x to a float.
x = float(x) # After running this line, the variable x now represents a floating␣
↪point value and not an integer.

# Sanity check:
```

```
print(type(x))
print("x = {}".format(x))
```

```
<class 'float'>
x = 1.0
```

[27]:
```
# When casting from float to int, the number gets truncated such that the␣
↪decimal part is removed
print(int(3.2))
print(int(3.5))
print(int(3.8))
```

```
3
3
3
```

1.1 Floating point issues and limitations

There are times when the result of a calculation with floating points seems to defy what we've traditionally learned…

For example, what is the sum of 1.1 and 2.2?

Let's see what Python thinks…

[28]:
```
print("Does 1.1 + 2.2 = 3.3?")
print(1.1 + 2.2 == 3.3)
```

```
Does 1.1 + 2.2 = 3.3?
False
```

[29]:
```
print(1.1+2.2)
```

```
3.3000000000000003
```

The above is a very classic floating point error, similar to the one we saw yesterday.

What happened?

Floating-point numbers are implemented in computer hardware as binary fractions. Thus, most decimal fractions cannot be accurately stored.

As a default, python uses double-precision floating point. This means that 53 bits are used, so the otherwise infinite representation is rounded to 53 significant bits.

In your computer, `0.1` is stored as the following rounded **binary value**: `0.0001100110011001100110011001100110011001100110011001101`.

This binary value is then converted to the following **decimal value** when printed to the console: `0.1000000000000000055511151231257827021181583404541015625`, which is slightly greater than 0.1.

[30]:
```python
print("0.1 to the 60th decimal place: {:.60f}".format(0.1))
print("0.1 to the 19th decimal place: {:.19f}".format(0.1))
print("0.1 to the 17th decimal place: {:.17f}".format(0.1))
```

```
0.1 to the 60th decimal place:
0.100000000000000005551115123125782702118158340454101562500000
0.1 to the 19th decimal place: 0.1000000000000000056
0.1 to the 17th decimal place: 0.10000000000000001
```

The following is an example from yesterday:

[31]:
```python
weight_kg = 25
conversion = 2.2

weight_lbs = weight_kg * conversion
print(weight_lbs)
```

```
55.00000000000001
```

[32]:
```python
"55 to the 60th decimal place: {:.60f}".format(2.2*25)
```

[32]:
```
'55 to the 60th decimal place:
55.000000000000007105427357601001858711242675781250000000000000'
```