while loop

INTERMEDIATE PYTHON

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

if-elif-else

`control.py`

- Goes through construct only once!

```
z = 6
if z % 2 == 0 :  # True
    print("z is divisible by 2")  # Executed
elif z % 3 == 0 :
    print("z is divisible by 3")
else :
    print("z is neither divisible by 2 nor by 3")

...   # Moving on
```

- While loop = repeated if statement

While

```
while condition :
    expression
```

- Numerically calculating model
- "repeating action until condition is met"
- Example
  - Error starts at 50
  - Divide error by 4 on every run
  - Continue until error no longer > 1

While

```
while condition :
    expression
```

`while_loop.py`

```
error = 50.0

while error > 1:
    error = error / 4
    print(error)
```

- Error starts at 50
- Divide error by 4 on every run
- Continue until error no longer > 1

While

```
while condition :
    expression
```

`while_loop.py`

```
error = 50.0
#     50
while error > 1:    # True
      error = error / 4
      print(error)
```

```
12.5
```

While

```
while condition :
    expression
```

while_loop.py

```
error = 50.0
#     12.5
while error > 1:    # True
      error = error / 4
      print(error)
```

```
12.5
3.125
```

While

```
while condition :
    expression
```

`while_loop.py`

```
error = 50.0
#     3.125
while error > 1:    # True
      error = error / 4
      print(error)
```

```
12.5
3.125
0.78125
```

While

```
while condition :
    expression
```

`while_loop.py`

```
error = 50.0
#     0.78125
while error > 1:    # False
    error = error / 4
    print(error)
```

```
12.5
3.125
0.78125
```

While

```
while condition :
    expression
```

`while_loop.py`

```
error = 50.0
while error > 1 :    # always True
    # error = error / 4
    print(error)
```

```
50
50
50
50
50
50
50
50
...
```

- DataCamp: session disconnected
- Local system: Control + C

Let's practice!

INTERMEDIATE PYTHON

for loop

INTERMEDIATE PYTHON

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

for loop

```
for var in seq :
    expression
```

- "for each var in seq, execute expression"

fam

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
print(fam)
```

```
[1.73, 1.68, 1.71, 1.89]
```

fam

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
print(fam[0])
print(fam[1])
print(fam[2])
print(fam[3])
```

```
1.73
1.68
1.71
1.89
```

for loop

```
for var in seq :
    expression
```

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
for height in fam :
    print(height)
```

for loop

```
for var in seq :
    expression
```

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
for height in fam :
    print(height)
    # first iteration
    # height = 1.73
```

```
1.73
```

for loop

```
for var in seq :
    expression
```

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
for height in fam :
    print(height)
    # second iteration
    # height = 1.68
```

```
1.73
1.68
```

for loop

```
for var in seq :
    expression
```

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
for height in fam :
    print(height)
```

```
1.73
1.68
1.71
1.89
```

- No access to indexes

for loop

```
for var in seq :
    expression
```

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
```

- ???

```
index 0: 1.73
index 1: 1.68
index 2: 1.71
index 3: 1.89
```

enumerate

```
for var in seq :
    expression
```

`family.py`

```
fam = [1.73, 1.68, 1.71, 1.89]
for index, height in enumerate(fam) :
    print("index " + str(index) + ": " + str(height))
```

```
index 0: 1.73
index 1: 1.68
index 2: 1.71
index 3: 1.89
```

Loop over string

```
for var in seq :
    expression
```

`strloop.py`

```
for c in "family" :
    print(c.capitalize())
```

```
F
A
M
I
L
Y
```

Let's practice!

INTERMEDIATE PYTHON

Loop Data Structures Part 1

INTERMEDIATE PYTHON

Hugo Bowne-Anderson
Data Scientist at DataCamp

datacamp

Dictionary

```
for var in seq :
    expression
```

`dictloop.py`

```
world = { "afghanistan":30.55,
          "albania":2.77,
          "algeria":39.21 }
for key, value in world :
    print(key + " -- " + str(value))
```

1. Loop Data Structures Part 1
So you already saw how looping over lists and strings works, but what about those other data structures, such as dictionaries and Numpy arrays? Well, in both cases, you can use a similar for loop construct, but the way you define the "sequence" over which you're iterating will differ depending on the data structure.

```
ValueError: too many values to
            unpack (expected 2)
```

Python sees that you expect two values in every iteration, like enumerate did before when you wanted the index and value from a list element, **but in this case, Python has no idea how to go about this.**

Dictionary

```
for var in seq :
    expression
```

`dictloop.py`

```
world = { "afghanistan":30.55,
          "albania":2.77,
          "algeria":39.21 }
for key, value in world.items() :
    print(key + " -- " + str(value))
```

```
algeria -- 39.21
afghanistan -- 30.55
albania -- 2.77
```

3. Dictionary

We can fix this by calling the method items() on world. This will generate a key and value in each iteration. If you have a look at the printout, there's something strange: afghanistan comes first in world, but not in the printout. That's because dictionaries are inherently unordered: the order in which they're iterated over is not fixed, at least in Python 3.5.

Dictionary

```
for var in seq :
    expression
```

`dictloop.py`

```
world = { "afghanistan":30.55,
          "albania":2.77,
          "algeria":39.21 }
for k, v in world.items() :
    print(k + " -- " + str(v))
```

```
algeria -- 39.21
afghanistan -- 30.55
albania -- 2.77
```

Numpy Arrays

```
for var in seq :
    expression
```

`nploop.py`

```
import numpy as np
np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
bmi = np_weight / np_height ** 2
for val in bmi :
    print(val)
```

```
21.852
20.975
21.750
24.747
21.441
```

2D Numpy Arrays

`nploop.py`

```
import numpy as np
np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
meas = np.array([np_height, np_weight])
for val in meas :
    print(val)
```

```
[ 1.73  1.68  1.71  1.89  1.79]
[ 65.4  59.2  63.6  88.4  68.7]
```

6. 2D Numpy Arrays
Let's see if this also works with a 2D Numpy array. Here, I created meas, by combining the np_height and np_weight arrays. If we want to print out each element in this 2D array separately, **the same basic for loop won't do the trick though**. The 2D array is actually built up from an array of 1D arrays. The for loop simply prints out an entire array on each iteration.

2D Numpy Arrays

`nploop.py`

```
import numpy as np
np_height = np.array([1.73, 1.68, 1.71, 1.89, 1.79])
np_weight = np.array([65.4, 59.2, 63.6, 88.4, 68.7])
meas = np.array([np_height, np_weight])
for val in np.nditer(meas) :
    print(val)
```

```
1.73
1.68
1.71
1.89
1.79
65.4
...
```

7. 2D Numpy Arrays
To get every element of an array, you can use a Numpy function called nditer(). The input is the array you want to iterate over, meas in our case. This time, we get 10 printouts, first all the heights, then all the weights. Nice!

Recap

- Dictionary
  - for `key, val in my_dict.items() :`
- Numpy array
  - `for val in np.nditer(my_array) :`

8. Recap
To recap: if you want to iterate over key-value pairs in a dictionary, use the items() method on the dictionary to define the sequence in the for loop. If you want to iterate over all elements in a Numpy array, you should use the nditer() function to specify the sequence. **Pay attention here: dictionaries require a method, Numpy arrays use a function**.

Let's practice!

INTERMEDIATE PYTHON

Loop Data Structures Part 2

INTERMEDIATE PYTHON

Hugo Bowne-Anderson
Data Scientist at DataCamp

brics

```
         country    capital    area  population
BR        Brazil   Brasilia   8.516      200.40
RU        Russia     Moscow  17.100      143.50
IN         India  New Delhi   3.286     1252.00
CH         China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98
```

`dfloop.py`

```
import pandas as pd
brics = pd.read_csv("brics.csv", index_col = 0)
```

for, first try

`dfloop.py`

```
import pandas as pd
brics = pd.read_csv("brics.csv", index_col = 0)
for val in brics :
    print(val)
```

```
country
capital
area
population
```

3. for, first try
If a Pandas DataFrame were to function the same way as a 2D Numpy array, then maybe a basic for loop like this, to print out each row, could work. Let's see what the output is. Well, this was rather unexpected. We simply got the column names. Also interesting, but not exactly what we want. In Pandas, you have to mention explicitly that you want to iterate over the rows.

iterrows

dfloop.py

```
import pandas as pd
brics = pd.read_csv("brics.csv", index_col = 0)
for lab, row in brics.iterrows():
    print(lab)
    print(row)
```

```
BR
country          Brazil
capital        Brasilia
area              8.516
population        200.4
Name: BR, dtype: object
...
RU
country         Russia
capital         Moscow
area              17.1
population       143.5
Name: RU, dtype: object
IN ...
```

4. iterrows

You do this by calling the iterrows method on the brics country, thus specifying another "sequence": The iterrows method looks at the data frame, and on each iteration generates two pieces of data: the label of the row and then the actual data in the row as a Pandas Series. Let's change the rest of the for loop to reflect this change: we store the row label as lab, and the row data as row. To understand what's happening, let's print lab and row seperately. In the first iteration, lab is BR, and row is this entire Pandas Series. Because this row variable on each iteration is a Series, you can easily select additional information from it using the subsetting techniques you learned about earlier.

Selective print

`dfloop.py`

```
import pandas as pd
brics = pd.read_csv("brics.csv", index_col = 0)
for lab, row in brics.iterrows():
    print(lab + ": " + row["capital"])
```

```
BR: Brasilia
RU: Moscow
IN: New Delhi
CH: Beijing
SA: Pretoria
```

The row data that's generated by iterrows() on every run is a Pandas Series. This format is not very convenient to print out. Luckily, you can easily select variables from the Pandas Series using square brackets:

Add column

`dfloop.py`

```
import pandas as pd
brics = pd.read_csv("brics.csv", index_col = 0)
for lab, row in brics.iterrows() :
    # - Creating Series on every iteration
    brics.loc[lab, "name_length"] = len(row["country"])
print(brics)
```

Running this scripts shows that it worked: there's a new column in there with the length of the country names. Nice, but not especially efficient, because you're creating a Series object on every iteration. For this small DataFrame that doesn't matter, but if you're doing funky stuff on a ginormous dataset, this loss in efficiency can become problematic.

```
         country    capital    area  population  name_length
BR        Brazil   Brasilia   8.516      200.40            6
RU        Russia     Moscow  17.100      143.50            6
IN         India  New Delhi   3.286     1252.00            5
CH         China    Beijing   9.597     1357.00            5
SA  South Africa   Pretoria   1.221       52.98           12
```

Using iterrows() to iterate over every observation of a Pandas DataFrame is easy to understand, **but not very efficient.** On every iteration, you're creating a new Pandas Series.
If you want to add a column to a DataFrame by calling a function on another column, the iterrows() method in combination with a for loop is not the preferred way to go. Instead, you'll want to use apply().

apply

dfloop.py

```
import pandas as pd
brics = pd.read_csv("brics.csv", index_col = 0)
brics["name_length"] = brics["country"].apply(len)
print(brics)
```

```
        country    capital    area  population
BR       Brazil   Brasilia   8.516      200.40
RU       Russia     Moscow  17.100      143.50
IN        India  New Delhi   3.286     1252.00
CH        China    Beijing   9.597     1357.00
SA  South Africa   Pretoria   1.221       52.98
```

7. apply
A way better approach if you want to calculate an entire DataFrame column by applying a function on a particular column in an element-wise fashion, is apply(). In this case, you don't even need a for loop. This is how it's done. Basically, you're selecting the country column from the brics DataFrame, and then, on this column, you apply the len function. Apply calls the len function with each country name as input and produces a new array, that you can easily store as a new column, "name_length". This is way more efficient, and also easier to read, if you ask me.

Let's practice!

INTERMEDIATE PYTHON