Python for Data Analysis: Methods & Tools

Python for everyday people

Written by Felipe Dominguez - Professor Adjunct
Hult International Business School

Chapter 04 - For & While Loops

1. Iterating in Python

You are now able to create more interesting and complex programs compared to where you started. You can use conditional statements to control the flow of the program based on the boolean outcome of multiple conditions. In addition to conditions, programs often need to perform iterative tasks. Let's consider the following example:

```
You are trying to unlock your phone, which requires you to
enter your password. If you enter the correct password, you'll
access your phone. If you don't, you will be prompted to input
your password again. This loop can continue until you either
enter the correct password or run out of tries.
```

As you can see, the access to your phone is based on a loop sequence controlled by a condition and the user's input! You can think this situation as a loop that will stop once a certain condition is met: either the user input the correct password or reaches the maximum number of tries. As you can imagine, the loop is running a certain code that is controlled by the outcome of a condition.

The dictionary defines loops as *a structure, series, or process, the end of which is connected to the beginning*. In the context of programming, a loop is a set of code where the end is connected to the beginning in some way. Loops are powerful coding structures that allows you to iterate over multiple items and run code without the need to explicitly write it multiple times.

There two loop functions in Python:

```
i. for: Iterates until the last item of an object (E.g., List,
set, among others) [Code 1.1.]
ii. while: Iterate while a condition is true. [Code 1.2.]
```

Both types of loops will be covered in this chapter. Let's go for it!

In [2]:
```
# Code 1.1.
# Fibonacci series:
# the sum of two elements defines the next

# Declare a & b variables
a, b = 0, 1

# for statement, iterate from rang 0 to 9
for i in range(0,10):
    print(f"""iteration: {i}, Fibonacci number: {a}""")
    a, b = b, a + b
```

```
iteration: 0, Fibonacci number: 0
iteration: 1, Fibonacci number: 1
iteration: 2, Fibonacci number: 1
iteration: 3, Fibonacci number: 2
iteration: 4, Fibonacci number: 3
iteration: 5, Fibonacci number: 5
iteration: 6, Fibonacci number: 8
iteration: 7, Fibonacci number: 13
iteration: 8, Fibonacci number: 21
iteration: 9, Fibonacci number: 34
```

In [2]:
```
# Code 1.2.
# Fibonacci series:
# the sum of two elements defines the next

# Declare a & b variables
a, b = 0, 1

# while statement (condition: a < 10)
while a < 35:
    print(a)
    a, b = b, a + b
```

```
0
1
1
2
3
5
8
13
21
34
```

2. For loops

2.1. For fundamentals

For loops are used to **iterate** over an **iterable** object (e.g., lists, sets, arrays, etc.). The loop will continue until it has iterated through all the elements in the object. In the example in code 1.1, the range(0,10) function creates a list of elements from 0 to 9. The loop starts with the first element, i = 0, and runs the code in the loop's body. The loop starts with the first element, i = 0, and runs the code in the loop's body. After the code in the body is completed, the loop moves on to the next element, with i taking on the value of the next element in the list. In other programming languages, you might need to manually increment the iterable item or use a "next" statement to move to the next element. However, in Python, the loop automatically moves to the next element in the iterable object. The loop will terminate when it has iterated through all the elements in the object, in this case when i = 9. Illustration 2.1.1 shows a visual representation of this process.

The structure of a **for loop statement** is similar to that of a conditional statement, as shown in illustration 2.1.2.

The first line declares the for loop, which is finished with a semicolon (:). Between the for keyword and the semicolon, you must specify the **iterator item** and the **object to be iterated over**. Any code indented below the first line will be executed as part of the for loop.

Note: *The item iterator will always keep the value of the current item in the object being iterated over.*

The iterator item

The iterator item is shown as **element** in figure 2.1.2 and it is used to stream the data one by one from the iterable object. You can choose any name for the iterator variable, but it is recommended to use a name that explains what you are iterating over. For example, code 2.1.4 demonstrates how to iterate through the elements of a list of countries. This code produces the same result as code 2.1.3, but in a more compact and efficient way.

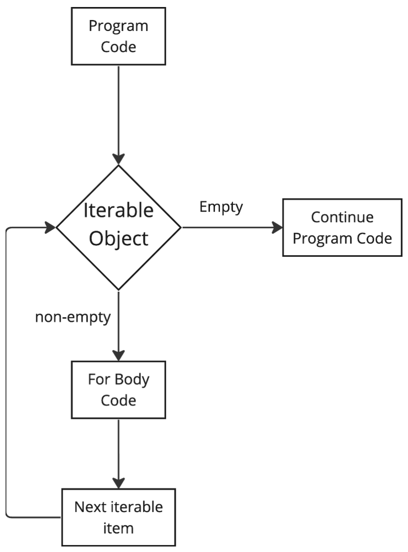


Figure 2.1.2. for Statement

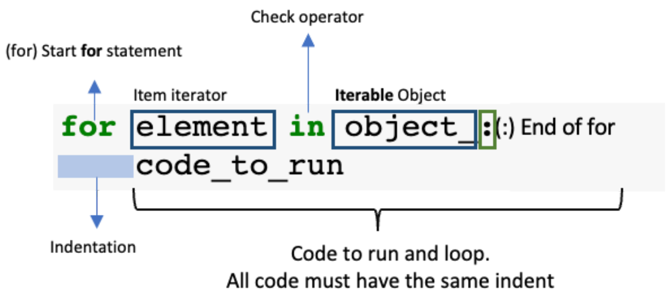


Figure 2.1.2. for Statement

In [3]:
```
# Code 2.1.1.

# Declare list variable
fav_countries = ['Germany', "Spain", "USA"]

# print all countries
print(fav_countries[0])
print(fav_countries[1])
print(fav_countries[2])
```

```
Germany
Spain
USA
```

In [4]:
```
# Code 2.1.2.

# print all countries
for country in fav_countries:
    print(country)
```

```
Germany
Spain
USA
```

2.2. Let's continue looping

You can create more complex **for loops** by combining for loops with conditional statements. This can be useful for changing values or applying specific programs for specific situations within a loop. For example, you might use this technique to update the prices of certain items in a list of clothing or to replace missing values in a list. You could also use it to categorize variables into ranges, such as low, medium, or high income households.

For instance, you might use a for loop to analyze household income by state in 2021.

In [1]:
```
# Code 2.2.1.

# Declare a list of lists with the mean household income per state. AL, MA, TX,
# US are missing (US: us average)
state_income = [["AL",    ''], ["AK", 65813], ["AZ", 55487], ["AR", 50625], \
                ["CA", 76614], ["CO", 70706], ["CT", 83294], ["DE", 59931], \
                ["DC", 96477], ["FL", 62270], ["GA", 55786], ["HI", 60947], \
                ["ID", 52369], ["IL", 67244], ["IN", 56497], ["IA", 57163], \
                ["KS", 58924], ["KY", 51266], ["LA", 54217], ["ME", 58484], \
                ["MD", 69817], ["MA",    ''], ["MI", 56494], ["MN", 66280], \
                ["MS", 45881], ["MO", 55325], ["MT", 56949], ["NE", 61205], \
                ["NV", 60213], ["NH", 73200], ["NJ", 77016], ["NM", 50311], \
                ["NY", 76837], ["NC", 56173], ["ND", 64524], ["OH", 56879], \
                ["OK", 53870], ["OR", 61596], ["PA", 64279], ["RI", 64376], \
                ["SC", 52467], ["SD", 64462], ["TN", 56560], ["TX",    ''], \
                ["UT", 56019], ["VT", 61882], ["VA", 66305], ["WA", 73775], \
                ["WV", 48488], ["WI", 59626], ["WY", 69666], ["US",   '']]

# Only print the first 4 elements.
for state in state_income[0:4]:
    print(state)
```

```
['AL', '']
['AK', 65813]
['AZ', 55487]
['AR', 50625]
```

2.3. Iterating through objects

As you can observed, every line of the code 2.2.1. outcome is in the format "['state', income]". Well, this is because you **are iterating in the first level of the iterable object (list of lists)**.

Let's analyze the for statement.

Code 2.2.1. process.

```
for state in state_income:
    print(state)

The 1st item of the iterator item ("state")  in state_income is
a list: ['AL',    '']
The 2nd item of the iterator item ("state")  in state_income is
a list: ['AK', 65813]
                    ...
                    ...
                    ...
The last item of the iterator item ("state")  in state_income
is a list: ['US',    '']
```

What happen if you want to get just the state value or just the income value? Well, you will need **to iterate one level deeper**. To do that, you have to add a new item iterator, so Python can understand that for each iteration you want to pull the values present in each list.

Let's analyze code 2.2.2.

For within fors!

Code 2.2.2 works correctly when all the lists within the main list have the same length. When you pass two iterator variables (state and h_income), Python understands that it needs to "unpack" each list and store the particular value in each variable. However, this is not the case when the length of the inner lists is different.

As you can see in code 2.2.3, if you try to use the same logic on a list of lists with different lengths, Python will return an error when it reaches the first list with a different length.

The for loop in this code expects to find three iterator variables (state, h_income, and region) in the elements of the state_income_2 list. However, the third element only contains two iterator variables. Therefore, Python is unable to continue processing it.

So, to iterate over a list with different lengths you will need to **nest a loop** into another loop. Code 2.2.4. presents a solutions for this problem.

There is a lot happening in this code, so let's break it down and analyze it line by line

</table>

In [6]:
```
# Code 2.2.2.

# Iterate one level deeper. state and h_income
# store one particular value each
for state, h_income in state_income[0:4]:
    print(f"""{state}: {h_income}""")
```

```
AL: 
AK: 65813
AZ: 55487
AR: 50625
```

In [32]:

```
# Code 2.2.3.

# Declare new variable with region
state_income_2 = \
                [["AL",   '', "Southeast"], \
                ["AK", 65813,  "Far West"], \
                ["AZ", 55487],              \
                ["AR", 50625, "Southeast"]]

# This code will output an error
print("------- The code below will throw an error -------")
for state, h_income, region in state_income_2:
    print(f"""{state}; {h_income}; {region}""")
```

```
------- The code below will throw an error -------
AL; ; Southeast
AK; 65813; Far West
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most r
ecent call last)
Input In [32], in <cell line: 12>()
     10 # This code will output an error
     11 print("------- The code below will throw an error -
------")
---> 12 for state, h_income, region in state_income_2:
     13     print(f"""{state}; {h_income}; {region}""")

ValueError: not enough values to unpack (expected 3, got 2)
```

In [8]:

```
# Code 2.2.4.

# Declare new variable with region
state_income_2 = \
                [["AL", 49769, "Southeast"], \
                ["AK", 65813,  "Far West"], \
                ["AZ", 55487],              \
                ["AR", 50625, "Southeast"]]

# Loop within state_income_2 list
for state in state_income_2:

    # Declare a string variable
    text = ''

    # Nested loop
    for i in range(len(state)):

        # Concat text
        text = text + " - "+ str(state[i])

    # print each state
    print(text)
```

```
 - AL - 49769 - Southeast
 - AK - 65813 - Far West
 - AZ - 55487
 - AR - 50625 - Southeast
```

2.4. Combine for & if

The data for household income by state is **incomplete**. As you can see, there is no information for Arizona, Massachusetts, Texas, or the US average.

When working in the fields of data science, statistics, or analytics, you will often encounter messy, unordered, misspelled data. That's why it's important to always start by understanding and cleaning your data when working with a new dataset. There are various methods for dealing with missing data, but the first step is always to determine if you can retrieve the data from other sources. You will learn different techniques for working with **null (missing)** values in future courses. For now, let's start by replacing the missing values with known values.

2.4.1. Replacing known missing values

Let's say you have found the missing household income data for Alabama and Massachusetts. How can you add this data to the original list? One option would be to create a new list and write the new values there. However, this approach becomes impractical if the data changes frequently or if you need to add 3000 values instead of just 2. It doesn't look like an easy task now, right?

To update a value within a **list of lists**, you need to iterate over each list element and check for missing values. As a first step, let's try inputting a fixed value for all the missing values. To do this, you will need to use a conditional statement to check for missing values and save the result to a new list. Take a look at code 2.4.1 for an example.

Note: Hard-code *is the process of writing code containing static syntax (e.g. using a number instead of a variable that store a number)*

In [9]:
```
# Code 2.4.1.

# Store a value for all missing values.
user_input = int(input(prompt = "Please enter a value for household income: "))

# Create a new list
new_state_income = []

# loop over each state_income list element
for state, income in state_income:
    # Checking if the item is empty
    if income == "":

        # update income value from user input
        income = user_input

    # append new list element into the new list of incomes
    new_state_income.append([state, income])

# Check every income missing (AL, MA, TX, US) was replaced with user input.
print(new_state_income)
```

```
Please enter a value for household income: 200
[['AL', 200], ['AK', 65813], ['AZ', 55487], ['AR', 50625],
['CA', 76614], ['CO', 70706], ['CT', 83294], ['DE', 59931],
['DC', 96477], ['FL', 62270], ['GA', 55786], ['HI', 60947],
['ID', 52369], ['IL', 67244], ['IN', 56497], ['IA', 57163],
['KS', 58924], ['KY', 51266], ['LA', 54217], ['ME', 58484],
['MD', 69817], ['MA', 200], ['MI', 56494], ['MN', 66280],
['MS', 45881], ['MO', 55325], ['MT', 56949], ['NE', 61205],
['NV', 60213], ['NH', 73200], ['NJ', 77016], ['NM', 50311],
['NY', 76837], ['NC', 56173], ['ND', 64524], ['OH', 56879],
['OK', 53870], ['OR', 61596], ['PA', 64279], ['RI', 64376],
['SC', 52467], ['SD', 64462], ['TN', 56560], ['TX', 200],
['UT', 56019], ['VT', 61882], ['VA', 66305], ['WA', 73775],
['WV', 48488], ['WI', 59626], ['WY', 69666], ['US', 200]]
```

More specific!

In the previous example, you asked the user to input the data. Even if the user input a correct value, such as the US average or the household income in Texas, this value may not be appropriate for all of the missing values in the dataset. Therefore, it's important to be more specific about how you update the values. Code 2.4.1.2 presents a way to update the household income for any state using a dictionary, while code 2.4.1.3

allows the user to input a value to update the household income for a random state.

In [2]:
```
# Code 2.4.1.2.


# Declare a dictionary with the values we don't have
null_values = {"AL": 49769,
               "MA": 83653,
               "TX": 59865}

# Declare new list of lists
new_state_income = []

# loop over each state_income list element
for state, income in state_income:

    # Condition statement
    if state in null_values.keys():

        # Loop statement 2
        for key in null_values.keys():

            # Condition statement 2
            if key == state:

                # append new list value
                new_state_income.append([key,
null_values[key]])

    # if the state is not in the dictionary
    # keep same value as original.
    else:
        new_state_income.append([state, income])

print(new_state_income)
```

```
[['AL', 49769], ['AK', 65813], ['AZ', 55487], ['AR', 5062
5], ['CA', 76614], ['CO', 70706], ['CT', 83294], ['DE', 599
31], ['DC', 96477], ['FL', 62270], ['GA', 55786], ['HI', 60
947], ['ID', 52369], ['IL', 67244], ['IN', 56497], ['IA', 5
7163], ['KS', 58924], ['KY', 51266], ['LA', 54217], ['ME',
58484], ['MD', 69817], ['MA', 83653], ['MI', 56494], ['MN',
66280], ['MS', 45881], ['MO', 55325], ['MT', 56949], ['NE',
61205], ['NV', 60213], ['NH', 73200], ['NJ', 77016], ['NM',
50311], ['NY', 76837], ['NC', 56173], ['ND', 64524], ['OH',
56879], ['OK', 53870], ['OR', 61596], ['PA', 64279], ['RI',
64376], ['SC', 52467], ['SD', 64462], ['TN', 56560], ['TX',
59865], ['UT', 56019], ['VT', 61882], ['VA', 66305], ['WA',
73775], ['WV', 48488], ['WI', 59626], ['WY', 69666], ['US',
'']]
```

In [5]:
```
# Code 2.4.1.2.
# Import
import random
```

```
# Select a random state to update
user_state = state_income[random.randint(0,52)][0]

# Ask user value of the household income
user_input = input(f"""Input household income for:
{user_state} - """)
null_values[user_state] = user_input

# Declare new list of lists
input_new_state_income = []

# loop over each state_income list element
for state, income in state_income:

    # Condition statement
    if state in null_values.keys():

        # Loop statement 2
        for key in null_values.keys():

            # Condition statement 2
            if key == state:

                # append new list value
                input_new_state_income.append([key,
null_values[key]])

    # if the state is not in the dictionary
    # keep same value as original.
    else:
        input_new_state_income.append([state, income])

print(input_new_state_income)
```

```
Input household income for:
NC - 2
[['AL', 49769], ['AK', 65813], ['AZ', 55487], ['AR', 5062
5], ['CA', 76614], ['CO', 70706], ['CT', 83294], ['DE', 599
31], ['DC', 96477], ['FL', 62270], ['GA', 55786], ['HI', 60
947], ['ID', 52369], ['IL', 67244], ['IN', 56497], ['IA', 5
7163], ['KS', 58924], ['KY', 51266], ['LA', 54217], ['ME',
58484], ['MD', 69817], ['MA', 83653], ['MI', 56494], ['MN',
66280], ['MS', 45881], ['MO', 55325], ['MT', 56949], ['NE',
61205], ['NV', 60213], ['NH', 73200], ['NJ', 77016], ['NM',
50311], ['NY', 76837], ['NC', '2'], ['ND', 64524], ['OH', 5
6879], ['OK', 53870], ['OR', 61596], ['PA', 64279], ['RI',
64376], ['SC', 52467], ['SD', 64462], ['TN', 56560], ['TX',
59865], ['UT', 56019], ['VT', 61882], ['VA', 66305], ['WA',
73775], ['WV', 48488], ['WI', 59626], ['WY', 69666], ['US',
'']]
```

2.4.2. Replacing unknown missing values

As mentioned earlier, you will often encounter missing values in your dataset. One common method for handling missing values is to input new values based on statistical metrics such as the **mean or median**.

To calculate the average of a list you can use the **NumPy** package. NumPy is the fundamental package for scientific computing in Python, that allows you to perform mathematical, logical, linear algebra, and statistical operations. In order to access numpy you will need to import it. Conventionally, numpy is imported as **np**.

```
import numpy as np
```

To calculate the mean value, you need to iterate over the **new_state_income** list, which contains all the values for household income by state (as seen in code 2.4.1.2). You need to save the values of all household incomes that do not have the value "US" (the last item in the new_income_state list). Finally, you can apply the **np.mean()** method and save the mean value in a variable..

Take a look at code 2.4.2.1 and 2.4.2.2 to see how to save all the values to the **final_state_income** list.

In [12]:
```
# Code 2.4.2.1.
import numpy as np

# Declare new list
income_per_state = []

# Loop statement
for state, income in new_state_income:

    # Conditional statement
    if state != "US":

        # append income
        income_per_state.append(float(income))

# Calculate mean (arithmetic)
us_average = round(np.mean(income_per_state), 0)

# print result
print(us_average)
```

```
62115.0
```

In [13]:
```
# Code 2.4.2.2.

# Input new value into US item
final_state_income = []

for state, income in new_state_income:
    if state == "US":
        final_state_income.append([state, us_average])
    else:
        final_state_income.append([state, income])

# print last value (us average)
print(final_state_income[-1])
```

```
['US', 62115.0]
```

Bonus: Plotting household income by state (2021)

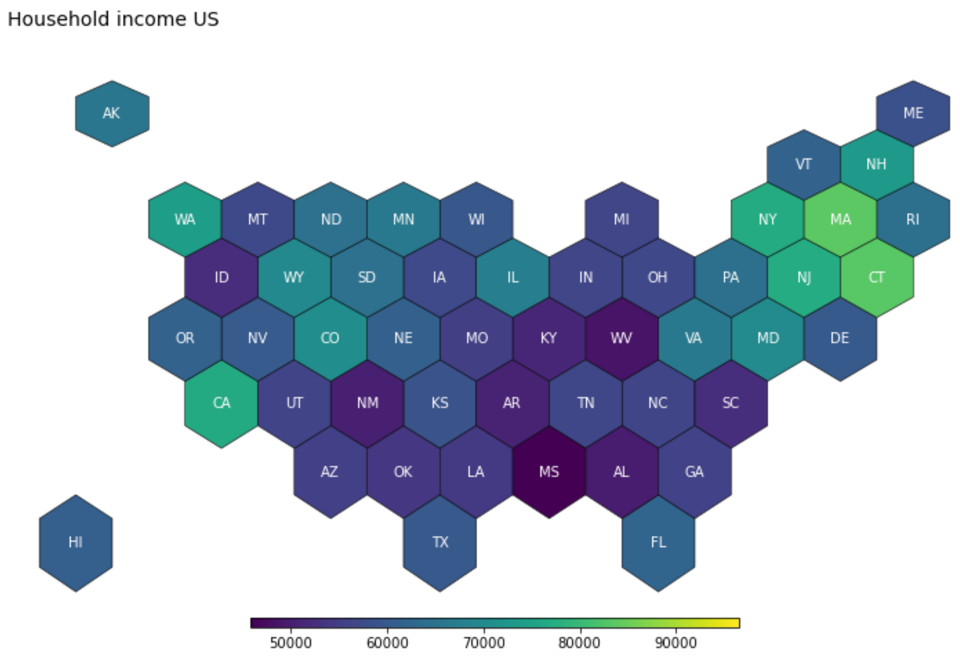
Let's plot the data of household income into a US hexagon map. Look at **Bonus Code**.

Note: *You will need to install the following packages: geopandas, matplotlib, and pandas. Create a new cell block a run: %pip install **package_name**.*

In [14]:
```
# Bonus Code
# %pip install geopandas
import geopandas as gpd
import matplotlib.pyplot as plt
import pandas as pd

# Read household income file
df_household_i = pd.read_csv("./bea_income.csv")

# download and store geodata
url = "https://raw.githubusercontent.com/holtzy/The-Python-Graph-
Gallery/master/static/data/us_states_hexgrid.geojson.json"
geoData = gpd.read_file(url)

# add a "centroid" column with the centroid position of each
county
geoData['centroid'] = geoData['geometry'].apply(lambda x:
x.centroid)

# Add a new column to the geo dataframe that will be used for
joining:
geoData['state'] = geoData['google_name'].str.replace(' \
(United States\)','')

# Merge the mariage dataset with the geospatial information
geoData =
geoData.set_index('state').join(df_household_i.set_index('GeoNam
# Initialize the figure
fig, ax = plt.subplots(1, figsize=(13, 13))

# map counties with the right color:
geoData.plot(
    ax=ax,
    column="y_2021",
    cmap="viridis",
    norm=plt.Normalize(vmin=45881, vmax=96477),
    edgecolor='black',
    linewidth=.5
)

# Remove useless axis
ax.axis('off')

# Add title, subtitle and author
ax.annotate('Household income US', xy=(10,440),  xycoords='axes
pixels',      \
            horizontalalignment='left',
verticalalignment='top', fontsize=14, color='black')

# for each county, annotate with the county name located at the
centroid coordinates
for idx, row in geoData.iterrows():
    ax.annotate(
        text=row['iso3166_2'],
        xy=row['centroid'].coords[0],
        horizontalalignment='center',
        va='center',
        color="white"
    )

# Add a color bar
sm = plt.cm.ScalarMappable(cmap='viridis',
norm=plt.Normalize(vmin=45881, vmax=96477))
fig.colorbar(sm, orientation="horizontal", aspect=50,
fraction=0.01, pad=0 )
```

```
/var/folders/1k/brqb_p7s34s2ycd6vzfrbxf80000gn/T/ipykernel_
59555/744655871.py:18: FutureWarning: The default value of
regex will change from True to False in a future version.
  geoData['state'] = geoData['google_name'].str.replace(' \
(United States\)','')
```

```
Out[14]: <matplotlib.colorbar.Colorbar at 0x7fb44999c190>
```



3. While loops

For loops run as long as there are still iterator items to iterate over. On the other hand, ***while loops*** *run as long as a specified condition is True. Once the condition is False, Python will move on with the code out of the while loop. Figure 3.1. presents a conceptual drawing of while loops flow.*

While loops are a powerful coding structure that can be used for a variety of tasks, such as:

```
Control user input (access your phone)
Breaking lists into groups
```

Let's dive into while statements fundamentals.

Note: *For loops have a* ***finite lifespan*** *and will run until the iterable object has no more items. In contrast, while loops can run indefinitely if the condition is always true. Therefore, you need to be careful when defining* *while loops. You can see the differences between for loops and while loops in codes 3.1 and 3.2. You can see the differences between an finite and infinite while loops in codes 3.1 and 3.2.*

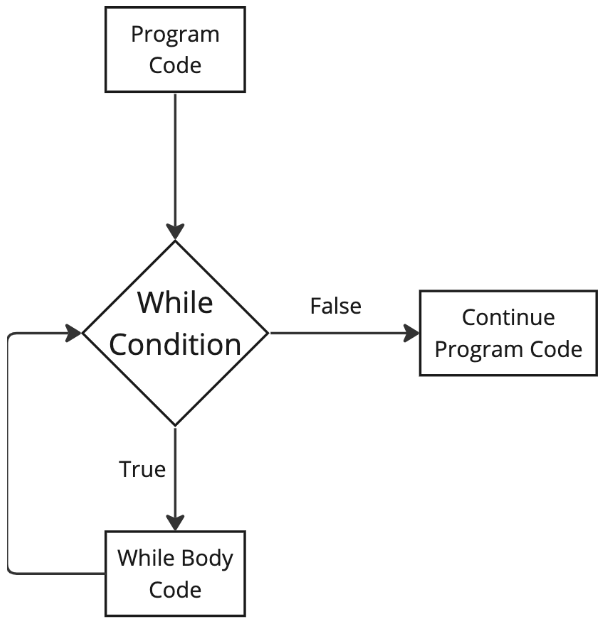
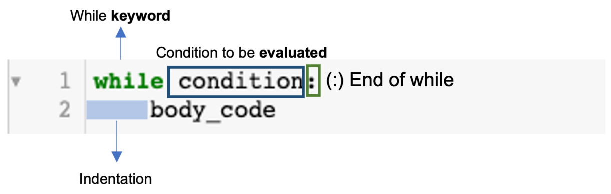


Figure 3.1. Conceptual drawing of while loop



```
while condition:
    body_code
```

Figure 3.2. While statement structure

In [10]:
```
# Code 3.1.

# declare variable
i = 1
```

```
n = 5

# while loop from i = 1 to 5
while i <= n:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

In [ ]:
```
# Code 3.2.
# Warning: This is an infinite loop.
# If you run, click the stop button

# declare variable
i = 1
n = 5

# while loop from i = 1 to 5
while i <= n:
    print(i)
    #i +=1
```

3.1. While fundamentals

While statements follow a clear and easy syntax:

```
while condition:
    body_code
```

After declaring the while loop, you need to specify the condition and end the line with a semicolon (:). As mentioned before, the while loop will run the body code as long as the condition is True. To illustrate this, let's create a code to test a password. To do this, you will need to::

```
1. Define a password and ask user to input a
password.
2. Define a condition to loop every time the user
input the wrong password
3. Give the user a certain number of attempts to
enter the correct password.
4. Allow the user to access the phone if the
password is correct.
```

*Code 3.1.1. defines the while condition based on two expressions: * If the user input a password different than the correct one (**user_input != password**) * If the user exhausted the number of attempts (**attempt > 0**)*

*Once **one of these** conditions is False, Python will run any code below and out of the while statement. In this case, the user will either be able to access the phone or not.*

In [16]:
```
# Code 3.1.1.

# Define correct password
password = "1234"

# Ask user to input a password
user_input = input(prompt = "Please introduce your 4 digists and numeric passowrd: ")

# Define number of attempts
attempt = 3

# while statement
while user_input != password and attempt > 0:

    # Ask user to input a new password. Tell how many attempts are left.
    user_input = input(prompt = f"""\nThat's not the correct password. Please try again
You have {attempt} attempts left. Please introduce your 4 digists and numeric passowrd: \n""")

    # reduce the number of attempts in 1.
    attempt -= 1

# Conditional statement.
if  user_input == password:
    print("\nYou have acces now!")

else:
    print("Try again in 5 minutes")
```

```
Please introduce your 4 digists and numeric passowrd: 12

That's not the correct password. Please try again
You have 3 attempts left. Please introduce your 4 digists
and numeric passowrd:
32

That's not the correct password. Please try again
You have 2 attempts left. Please introduce your 4 digists
and numeric passowrd:
4

That's not the correct password. Please try again
You have 1 attempts left. Please introduce your 4 digists
and numeric passowrd:
1234

You have acces now!
```

3.2. Truly, give me a break!

*There is a way to end a while loop before the condition is False. The **break** statement is used to exit (i.e., break) a loop. It is typically triggered by a conditional statement and is written inside the loop statement. When encountered, it terminates **the current loop** (i.e., it will always end the innermost loop) and immediately resumes execution at the next statement after the loop. **Note:** Be careful! Any code at the same indentation level and below a **break** statement will **never** run.*

In [17]:
```
# Code 3.2.1.

# while statement
while True:
    print("This while loop will run only 1 time")

    # Breaking the while!
    break

    print("This code will not run!")
```

```
This while loop will run only 1 time
```

In [18]:
```
# Code 3.2.2.

# for statement
for i in range(0, 10):
    print("This for loop will run only 1 time")

    # Breaking the while!
    break

    print("This code will not run!")
```

```
This for loop will run only 1 time
```

*Let's use the break statement to **exit** the while when the user has reached the last attempt or it has input the correct answer.*

In [30]:
```
# Code 3.1.1.

# Define correct password
password = "1234"

# Ask user to input a password
user_input = input(prompt = "Please introduce your 4 digists and numeric passowrd: ")

# Define number of attempts
attempt = 3
```

```
# while statement
while True:

    if user_input != password and attempt > 0:

        # Ask user to input a new password. Tell how many 
attempts are left.
        user_input = input(prompt = f"""\nThat's not the 
correct password. Please try again
You have {attempt} attempts left. Please introduce your 4 
digists and numeric passowrd: \n""")

    elif user_input == password:
        print("\nYou have acces now!")
        break

    elif attempt == 0:
        print("Try again in 5 minutes")
        break
    attempt -= 1
```

```
Please introduce your 4 digists and numeric passowrd: 1234

You have acces now!
```

4. When to while, when to for?

*A good rule of thumb to decide whether use a for or a while loop is to use **for loops when the number of iterations is known** (i.e., fixed number of iterables) and **while loops when it is unknown** (i.e., loop based on conditions). Furthermore, sometime it will make sense to nest a for into a while or vice versa.*

5. Summary

Hey there! You are almost ready with the fundamentals of Python!

Keep up your good progress!

*In this class you learned how to work with loops, specifically **for and while loops**. * **for loops** -> iterates over an iterable object. * **while loops** -> iterates as long as a condition is True * The syntax **break** exits a current loop*

Line	Interpretation
3-7	Declaring a multi-level list.
10	First loop statement is defined. Python will go through each element of the first-level list.
12	As each value of the list will be stored individually, you will need to concatenate them after. The text variable will store this values in one object.
14	Nested loop statement. Python decides how many times to run based on the length of each state (item iterator) list.
16	**concatenate** the items of each state object
18	print the text value outside of the nested for statement. *Check the print syntax is outdented*