BIOS-584 Python Programming (Non-Bios Student)

Week 04

Instructor: Tianwen Ma, Ph.D.

Department of Biostatistics and Bioinformatics,
Rollins School of Public Health,
Emory University

Lecture Overview

- Manipulation of lists
  - Create empty lists
  - Assignment or replacement values
  - Revisit functions in Quiz1
  - Length of a list
- For loop
  - Basic syntax
  - Three ways to iterate for loops
- While loop
- week-04-additional-list-and-for-loop.ipynb

Empty list

- You can create an empty list using the None object
- None is a special object in Python that represents the absence of a value.
- The type of None is NoneType. It is the only instance of this type.

```
list_answers = [None, None, None]
print(list_answers)
```

```
[None, None, None]
```

```
print(type(None))
```

```
<class 'NoneType'>
```

Empty list

- None is often used to represent missing values, or in our case today, placeholders.
- None is not the same as 0, False, or an empty string ‘ ’
- No quotations mark with None.

```
None == 'None'
```

```
False
```

```
None == False
```

```
False
```

```
None == ' '
```

```
False
```

More information: https://realpython.com/null-in-python/

Assign or replace values to lists

- You can assign or replace values in a list using the index of the element
- Again, the index always starts from 0.

```
# What's the name of your hometown?
list_answers[0] = 'Atanta'
print(list_answers)
```

```
['Atanta', None, None]
```

```
list_answers[1] = 'Ann Arbor'
list_answers[2] = 'Seattle'
print(list_answers)
```

```
['Atanta', 'Ann Arbor', 'Seattle']
```

Append values to lists

- If you do not know the length of your list, you can use list.append() command.
- It adds the element to the end of the list
- Add only **one** element at a time

```
# We initialize an empty list with []
new_list = []
new_list.append('Atlanta')
new_list.append('Ann Arbor')
new_list.append('Seattle')
```

```
print(new_list)
```

```
['Atlanta', 'Ann Arbor', 'Seattle']
```

Extend lists

- You can add multiple elements to a list using the list.extend() command

```
new_list.extend(['Athens', 'Augusta', 'Savannah'])
print(new_list)
```

```
['Atlanta', 'Ann Arbor', 'Seattle', 'Athens', 'Augusta', 'Savannah']
```

Lists with repeated values

- You can create a list with repeated values using * operator
- The syntax is
  - List = [value] * n
- Three examples:
  - Repeat a single value 10 times
  - Repeat a list 4 times
  - Repeat 8 null values

```
# repeat a single value 10 times
list_rep_10 = [10] * 10
print(list_rep_10)
```

```
[10, 10, 10, 10, 10, 10, 10, 10, 10, 10]
```

```
[27]:
```

```
# repeat an existing list 4 times
list_rep_4 = list_answers * 4
print(list_rep_4)
```

```
['Atanta', 'Ann Arbor', 'Seattle', 'Atanta', 'Ann Arbor', 'Seattle',
'Atanta', 'Ann Arbor', 'Seattle', 'Atanta', 'Ann Arbor', 'Seattle']
```

```
[26]:
```

```
# repeat 8 null values
list_rep_8 = [None] * 8
print(list_rep_8)
```

```
[None, None, None, None, None, None, None, None]
```

Counting the length a list

- You can count the length of a list using the len() function.

```
print(list_answers)
print(len(list_answers))

print(new_list)
print(len(new_list))
```

```
['Atanta', 'Ann Arbor', 'Seattle']
3
['Atlanta', 'Ann Arbor', 'Seattle', 'Athens', 'Augusta', 'Savannah']
6
```

Common Mistakes

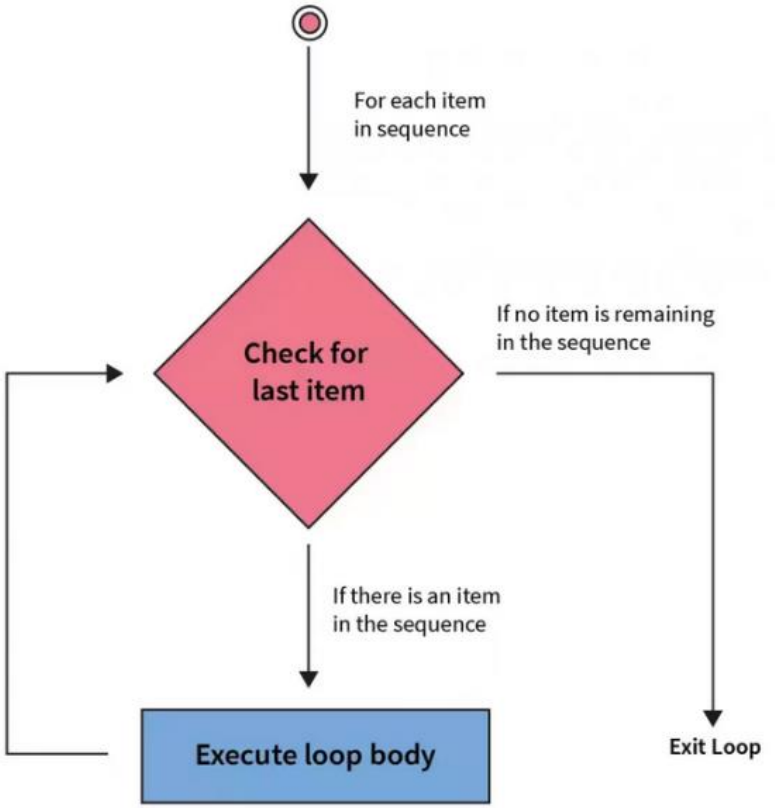
- Lists are not arrays, so you cannot perform mathematical operations using lists.
- You can only concatenate lists using the + operator.

```
list_a = [1, 2, 3]
print(list_a * 4)

list_b = [4, 5, 6]
print(list_a + list_b)

vec_a = np.array(list_a)
print(vec_a * 4)

list_a_from_vec = vec_a.tolist()
print(list_a_from_vec)
print(type(list_a_from_vec))
print(isinstance(list_a_from_vec, list))
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
[1, 2, 3, 4, 5, 6]
[ 4  8 12]
[1, 2, 3]
<class 'list'>
True
```

For loop

- A for loop is a way to iterate over a sequence of elements
- A useful tool to automate repetitive tasks



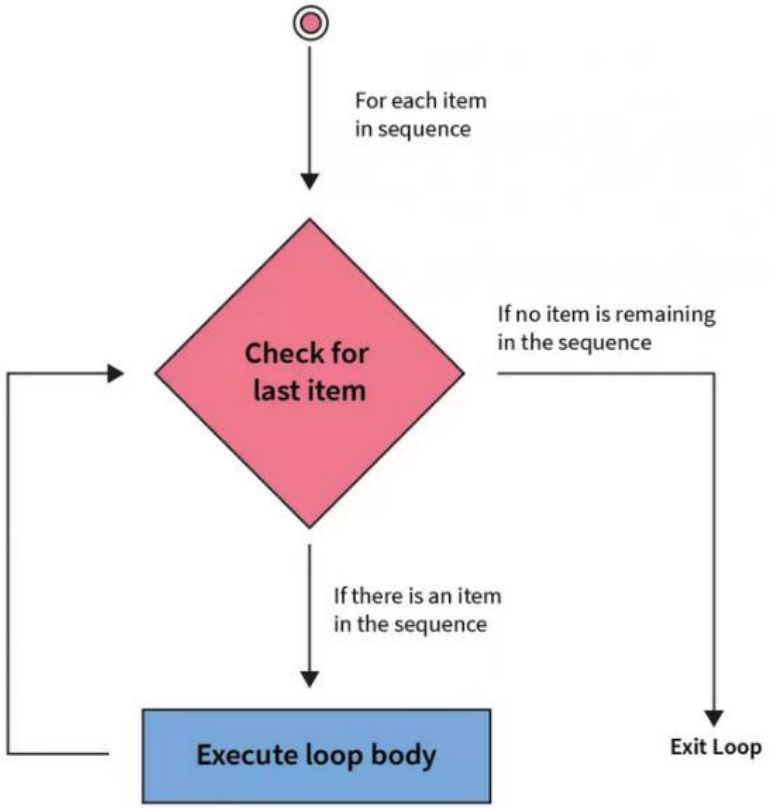
For loop

- The syntax is as follows:

```
for element in sequence:
    do something
```

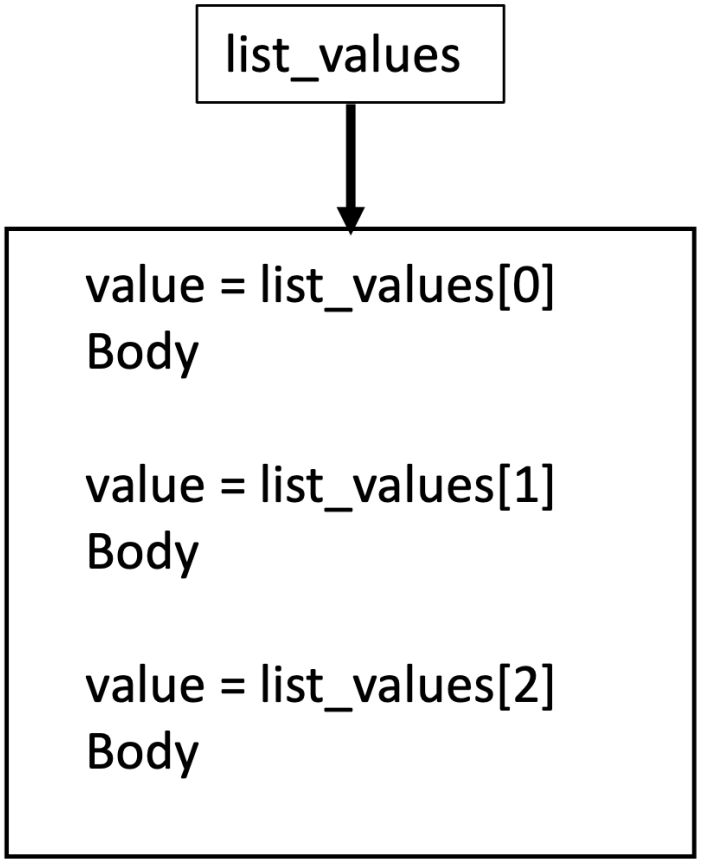
```
var_ls = [1, 2, 3, 4]
for i in var_ls:
    print(i)
```

```
1
2
3
4
```



What can we do with a for loop?

- You can iterate over a list of elements
  - Numbers, strings, or any other type of object
- You can iterate over a range of numbers
- You can iterate over a list of lists (nested lists)
- You can write nested for loops.



Examples

```
icd_10_cm_codes_ls = [
    ['alchol related disorders', 'opioid related disorders', 'cannabis related disorders'], # 'F10-F19',
    ['schizophrenia', 'schizotypal disorder', 'delutional disorders', 'brief psychotic disorder'], # 'F20-F29',
    ['manic episode', 'bipolar disorder', 'depressive episode', 'major depressive disorder, recurrent'] # 'F30-F39',
]

for code_iter_ls in icd_10_cm_codes_ls:
    print(code_iter_ls)
```

```
['alchol related disorders', 'opioid related disorders', 'cannabis related disorders']
['schizophrenia', 'schizotypal disorder', 'delutional disorders', 'brief psychotic disorder']
['manic episode', 'bipolar disorder', 'depressive episode', 'major depressive disorder, recurrent']
```

Nested for loops

```
for code_iter_ls in icd_10_cm_codes_ls:
    for disease_iter in code_iter_ls:
        print(disease_iter)
```

```
alchol related disorders
opioid related disorders
cannabis related disorders
schizophrenia
schizotypal disorder
delutional disorders
brief psychotic disorder
manic episode
bipolar disorder
depressive episode
major depressive disorder, recurrent
```

Additional examples

```
# If you want to introduce each disease type. You can write
icd_10_code_F20_29_ls = ['schizophrenia', 'schizotypal disorder', 'delutional disorders', 'brief psychotic disorder']
print('The F20-29 codes include ' + icd_10_code_F20_29_ls[0] + ' disease type.')
print('The F20-29 codes include ' + icd_10_code_F20_29_ls[1] + ' disease type.')
print('The F20-29 codes include ' + icd_10_code_F20_29_ls[2] + ' disease type.')
print('The F20-29 codes include ' + icd_10_code_F20_29_ls[3] + ' disease type.')
```

```
The F20-29 codes include schizophrenia disease type.
The F20-29 codes include schizotypal disorder disease type.
The F20-29 codes include delutional disorders disease type.
The F20-29 codes include brief psychotic disorder disease type.
The F20-29 codes include schizophrenia disease type.
```

```
# You can also use .format()
print('The F20-29 codes include {} disease type.'.format(icd_10_code_F20_29_ls[0]))
print('The F20-29 codes include {} disease type.'.format(icd_10_code_F20_29_ls[1]))
print('The F20-29 codes include {} disease type.'.format(icd_10_code_F20_29_ls[2]))
print('The F20-29 codes include {} disease type.'.format(icd_10_code_F20_29_ls[3]))
```

```
The F20-29 codes include schizophrenia disease type.
The F20-29 codes include schizotypal disorder disease type.
The F20-29 codes include delutional disorders disease type.
The F20-29 codes include brief psychotic disorder disease type.
```

Time-consuming and inefficient. We can use for loop to save time!

Additional examples

```
# But this is very time-consuming. What if you have 10, 50, or 100 disease types to introduce?
for disease_type_iter in icd_10_code_F20_29_ls:
    print('The F20-29 codes include ' + disease_type_iter + ' disease type.')
```

```
The F20-29 codes include schizophrenia disease type.
The F20-29 codes include schizotypal disorder disease type.
The F20-29 codes include delutional disorders disease type.
The F20-29 codes include brief psychotic disorder disease type.
```

```
# or use .format()
for disease_type_iter in icd_10_code_F20_29_ls:
    print('The F20-29 codes include {} disease type.'.format(disease_type_iter))
```

```
The F20-29 codes include schizophrenia disease type.
The F20-29 codes include schizotypal disorder disease type.
The F20-29 codes include delutional disorders disease type.
The F20-29 codes include brief psychotic disorder disease type.
```

Iterate over additional elements

- We can use + to concatenate two lists and write for loops.

```
icd_10_code_F30_39_ls = ['manic episode', 'bipolar disorder', 'depressive episode',
                         'major depressive disorder, recurrent']
for disease_iter in icd_10_code_F20_29_ls + icd_10_code_F30_39_ls:
    print(disease_iter)
```

```
schizophrenia
schizotypal disorder
delutional disorders
brief psychotic disorder
manic episode
bipolar disorder
depressive episode
major depressive disorder, recurrent
```

Three ways to iterate

- We simply iterate each element.
- We can use range(len(list_name)) and access the index of each element to iterate.

```
for disease_type_iter in icd_10_code_F20_29_ls:
    print('The F20-29 codes include {} disease type.'.format(disease_type_iter))
```

```
The F20-29 codes include schizophrenia disease type.
The F20-29 codes include schizotypal disorder disease type.
The F20-29 codes include delutional disorders disease type.
The F20-29 codes include brief psychotic disorder disease type.
```

```
for num_iter in range(len(icd_10_code_F20_29_ls)):
    print('The F20-29 codes include {} disease type.'.format(icd_10_code_F20_29_ls[num_iter]))
```

```
The F20-29 codes include schizophrenia disease type.
The F20-29 codes include schizotypal disorder disease type.
The F20-29 codes include delutional disorders disease type.
The F20-29 codes include brief psychotic disorder disease type.
```

Three ways to iterate

- We can also use enumerate() to display both relative index and its corresponding element.
  - Relative index comes first, and separated from iterative element by a “,”
  - Relative index starts from 0.

```
# we could also use enumerate() to show both relative index and its corresponding element.
for id_iter, disease_iter in enumerate(icd_10_code_F20_29_ls):
    print('The F20-29 codes include {} disease type in the location {}.'.format(
        disease_iter, id_iter+1)
    )
```

```
The F20-29 codes include schizophrenia disease type in the location 1.
The F20-29 codes include schizotypal disorder disease type in the location 2.
The F20-29 codes include delutional disorders disease type in the location 3.
The F20-29 codes include brief psychotic disorder disease type in the location 4.
```

Use of .format() to print

- Syntax:
  - print("{} … {} ".format(variable1, variable2))
- Support multiple inputs
  - The number of {} should be followed with the same size of variables to fill inside format().
- No restriction on data type, automatically convert it to string in this case.

```
for num_iter in range(len(icd_10_code_F20_29_ls)):
    print('The F20-29 codes include {} disease type in the location {}.'.format(
        icd_10_code_F20_29_ls[num_iter], num_iter+1)
    )
```

```
The F20-29 codes include schizophrenia disease type in the location 1.
The F20-29 codes include schizotypal disorder disease type in the location 2.
The F20-29 codes include delutional disorders disease type in the location 3.
The F20-29 codes include brief psychotic disorder disease type in the location 4.
```

Save time while coding

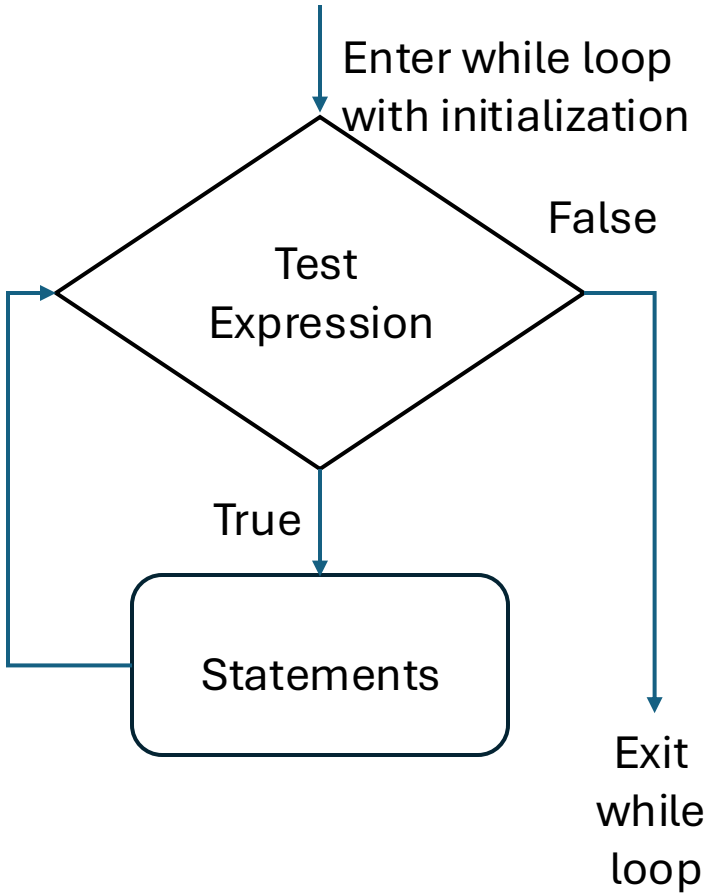
- You want to plot multiple scatterplots with different variable names.
- Instead of writing multiple codes, you can use the for loop and change the variable names accordingly.

Practice problem

- Finish the practice problem

While loop

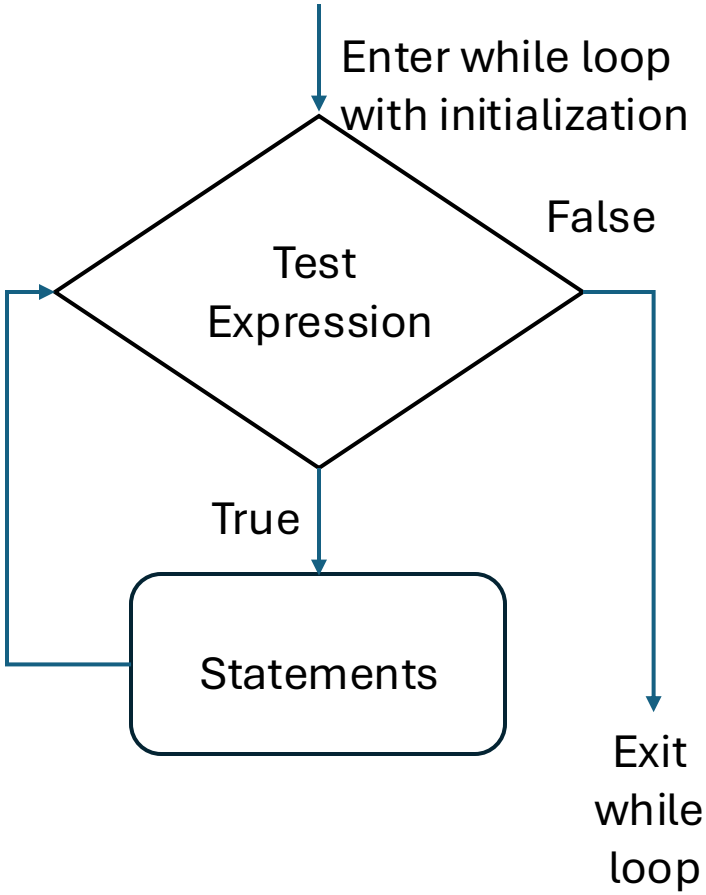
- A while loop is to execute a block of codes repeatedly until a given condition is satisfied.
- When the condition is False, the line immediately after the loop is executed.



While loop

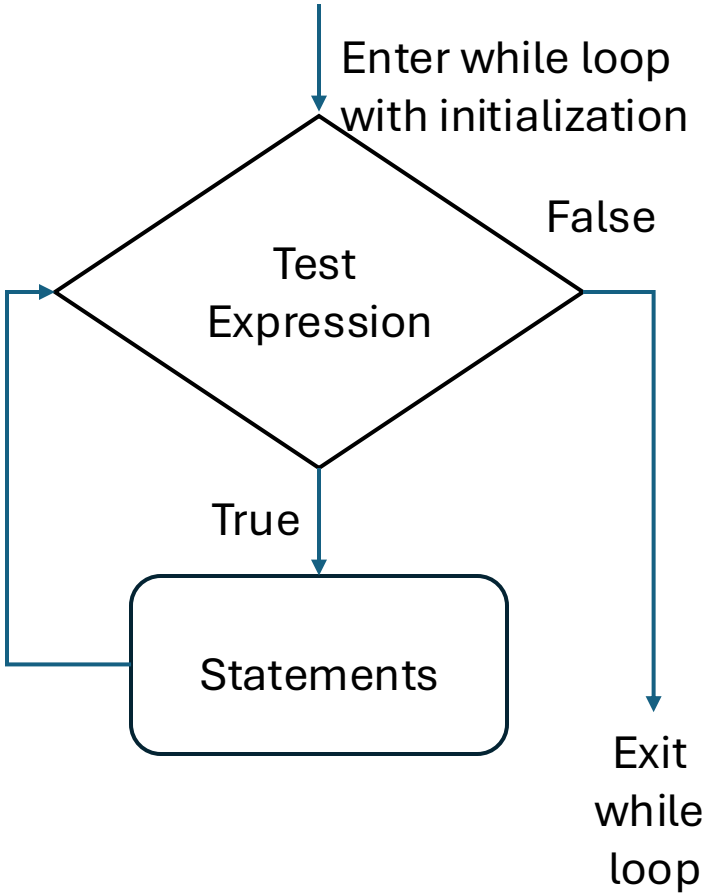
- The syntax looks like:
  - Make sure 4 indentations for statements!

```
initialization
related to expression
while expression:
    statements
```



What can we do with a while loop?

- For loop are used with a known times of iteration
- While loop are used to iterate with an unknown number of times.
  - The number of iterations depends on a given condition.



A simple example

```
# simple example
i = 1
while i < 6:
    print(i)
    i += 1
```

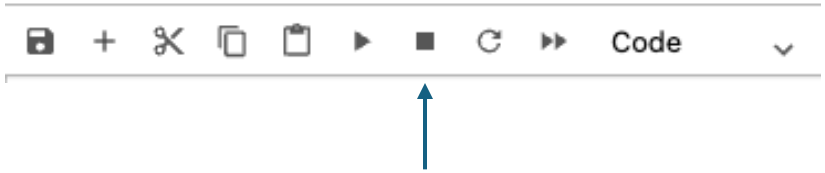
+=: i = i + 1

```
1
2
3
4
5
```

While loop

- The while loop will continue running the code block as long as the condition evaluates to True.
- Avoid the following infinite while loop
  - “Cmd+C” (Mac) or “Ctrl+C” (Windows) Terminal to stop a running process if you run the python file on Terminal.

```
# Avoid writing this while loop in python.
age = 28

# the test condition is always True
while age > 19:
    print('Infinite Loop')
# How do you stop it in Jupyter notebook?
# Press the "Interrupt the kernel" button.
```



Advanced features with while loop

- break: Immediately terminate a loop.
  - The program execution then proceed with the first statement following the loop body
- continue: Ends with the current iteration.
  - The execution jumps back to the loop header, and the loop condition is evaluated to determine whether the loop will execute again.
- else: Runs when the loop terminates naturally because the inital condition becomes False.

The break statement: Exiting a loop early

- Terminate the execution of a while loop and make your program continue with the first statement immediately after the loop body.

- Let's delve into the details.

```
# while with break
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    if number_iter == 2:
        break
    print(number_iter)
print('loop ended')
```

```
5
4
3
loop ended
```

The break statement: Exiting a loop early (Step-by-step)

- number_iter=6 (initial)
- First iteration:
  - Check while condition (pass)
  - number_iter=5
  - Check if condition (fail)
  - Print number_iter (5)
  - Go to next iteration

```
# while with break
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    if number_iter == 2:
        break
    print(number_iter)
print('loop ended')
```

```
5
```

The break statement: Exiting a loop early (Step-by-step)

- number_iter=5
- Second iteration:
  - Check while condition (pass)
  - number_iter=4
  - Check if condition (fail)
  - Print number_iter (4)
  - Go to next iteration

```
# while with break
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    if number_iter == 2:
        break
    print(number_iter)
print('loop ended')
```

```
5
4
```

The break statement: Exiting a loop early (Step-by-step)

- number_iter=4
- Third iteration:
  - Check while condition (pass)
  - number_iter=3
  - Check if condition (fail)
  - Print number_iter (3)
  - Go to next iteration

```
# while with break
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    if number_iter == 2:
        break
    print(number_iter)
print('loop ended')
```

```
5
4
3
```

The break statement: Exiting a loop early (Step-by-step)

- number_iter=3
- Fourth iteration:
  - Check while condition (pass)
  - number_iter=2
  - Check if condition (pass)
  - Run “break”
  - Quit the entire while loop, including the print function.

```
# while with break
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    if number_iter == 2:
        break
    print(number_iter)
print('loop ended')
```

```
5
4
3
```

The break statement: Exiting a loop early (Step-by-step)

- Print final string.

```
# while with break
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    if number_iter == 2:
        break
    print(number_iter)
print('loop ended')
```

```
5
4
3
loop ended
```

The break statement: Exiting a loop early

- What if we move the print function early?
- Why do we see the change in the output?

- Can you write down the logic flow by yourself?

```
# while with break
# What if we move the print function early?
number_iter = 6
while number_iter > 0:
    number_iter -= 1
    print(number_iter)
    if number_iter == 2:
        break
print('loop ended')
```

```
5
4
3
2
loop ended
```

The continue Statement: Skipping tasks in an iteration

- Skip some tasks in the current iteration when a given condition is met.
- Let's delve into the details.

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
3
0
Loop ended
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=6 (initial)
- First iteration:
  - Check while condition (pass)
  - number_iter=5
  - Check if condition (fail)
  - Print number_iter (5)
  - Go to next iteration

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=5
- Second iteration:
  - Check while condition (pass)
  - number_iter=4
  - Check if condition (fail)
  - Print number_iter (4)
  - Go to next iteration

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=4
- Third iteration:
  - Check while condition (pass)
  - number_iter=3
  - Check if condition (fail)
  - Print number_iter (3)
  - Go to next iteration

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
3
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=3
- Fourth iteration:
  - Check while condition (pass)
  - number_iter=2
  - Check if condition (pass)
  - Run continue; Skip functions thereafter (not the entire loop), i.e., do not print
  - Go to next iteration

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
3
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=2
- Fifth iteration:
  - Check while condition (pass)
  - number_iter=1
  - Check if condition (pass)
  - Run continue; Skip functions thereafter (not the entire loop), i.e., do not print
  - Go to next iteration

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
3
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=1
- Sixth iteration:
  - Check while condition (pass)
  - number_iter=0
  - Check if condition (fail)
  - Print number_iter (0)
  - Go to next iteration

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
3
0
```

The continue Statement: Skipping tasks in an iteration (Step-by-Step)

- number_iter=0
- Seventh iteration:
  - Check while condition (fail)
- Exit the while loop
- Print string
- End

```
# while with continue,
# skip tasks within an iteration
number_iter = 6

while number_iter > 0:
    number_iter -= 1
    if number_iter in [1,2]:
        continue
    print(number_iter)

print("Loop ended")
```

```
5
4
3
0
Loop ended
```

The else statement: running tasks at natural loop termination

- Python allows an optional else clause at the end of while loops.
- Run else clause only if the while loop terminates naturally without encountering a break statement.

```
while condition:
    <body>
else:
    <body>
```

```
# while with else
# only run else in presence of natural termination
i = 0
while i < 4:
    i += 1
    print(i)
else:  # Executed because no break in for
    print("No Break")
```

```
1
2
3
4
No Break
```

```
i = 0
while i < 4:
    i += 1
    print(i)
    break
else:  # Not executed as there is a break
    print("No Break")
```

```
1
```

The else statement: running tasks at natural loop termination

- Can you write down the logic flow by yourself?

```
# while with else
# only run else in presence of natural termination
i = 0
while i < 4:
    i += 1
    print(i)
else:  # Executed because no break in for
    print("No Break")
```

```
1
2
3
4
No Break
```

The else statement: running tasks at natural loop termination

- Can you write down the logic flow by yourself?

```
i = 0
while i < 4:
    i += 1
    print(i)
    break
else:  # Not executed as there is a break
    print("No Break")
```

```
1
```

More examples

- Finish the final practice.