Control Flow Python v2

1. Logical Vector Method:

One method for executing decision making tasks is by using logic. Instead of if-else statements, they are replaced with logical multipliers, which will result in True (1) or False (0).

There are several advantages to this:

- Logical approach can be faster to compute than if-else statements.
- Logical approach can make the code more readable.

For example, in the code example6 and example7, if-else statements and logical approach are used respectively. For the latter, only one condition will evaluate to True (1). So, the addition in the final line will produce the output of the formula corresponding to the True condition, and ignore the others that doesn't match the condition.

Logical vector method is used in example7.

$$tax = \begin{cases} 0.1inc & for\ inc \le 10000 \\ 1000 + 0.2(inc - 10000) & for\ 10000 < inc \le 20000 \\ 3000 + 0.5(inc - 20000) & for\ inc > 20000 \end{cases}$$

Variable	Calculations	Condition
tax1	0.1 * inc	inc <= 10000
tax2	1000 + 0.2 * (inc - 10000)	inc > 10000 and inc <= 20000
tax3	3000 + 0.5 * (inc - 20000)	inc > 20000
tax = tax1 + tax2 + tax3		
Example: inc = 15000		
tax1	0.1 * 15000	False (0)
tax2	1000 + 0.2 * (15000 - 10000)	True (1)
tax3	3000 + 0.5 * (15000 - 20000)	False (0)
tax = tax1 + tax2 + tax3 = 0 + 2000 + 0 = 2000		

2. Working of [round(val, 4) for val in Q] :

   In code of example10.py, there is a line of code where this line appears. It can be separated to 3 parts: for val in Q, round(val, 4) and [].

   In this case, the for loop was completed first, then the round() function.

   - First, there is for val in Q. This is a for loop that takes every element in Q and performs some function on it. In previous lines of code, few elements of calculation results have been inserted into Q, this means that Q is a list of numbers. In this case, for every element in Q, we call it *val*. This code will take every *val* in Q and perform round() function to it.
   - The round() function takes a number and returns a floating point number that is rounded, up to a specific number of decimals that the user can decide. The syntax of the function is round(Number, Decimals). Where Number is the number that user wants to round up, and Decimals is how much decimals that the user wants. Decimal parameter is optional.

     For example:

     result = round(4.6545671219, 3)

     print(result)   #Result is 4.655, which is round up result of x with 3 decimals.

   - Then, there is the []. This will take all the result of the code (for every *val* in Q, round up *val* to 4 decimals) and put them in an empty list.

   Combining all together, the breakdown workings of [round(val, 4) for val in Q] is take *val* in Q, round them up to 4 decimals, and then put them in an empty list. Repeat this for every *val* in Q.

   The final result is **[0.0, 1.0575, 1.9908, 2.8144]**.

3. Nested loop:

In Python programming language, there are two types of loops: for loop and while loop. Using these, we can create nested loops, which means one or more “inner loops” inside an “outer loop”. The "inner loop" will be executed one time for each iteration of the "outer loop".

For example, while loop inside the for loop, for loop inside the for loop, etc.

The syntax for nested loops:

```
“outer loop” Expression:
  “inner loop” Expression:
    Statement inside “inner loop”
  Statement inside “outer loop”
```

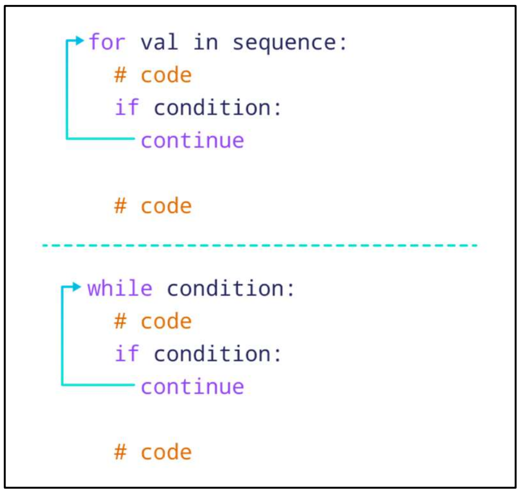
Example code:

Code	Output

for i in range(3):
    print(f"Index: {i}")
    for j in range(1, 3):
        print(f"   statement_{j}")
	Index: 0
	   statement_1
	   statement_2
	Index: 1
	   statement_1
	   statement_2
	Index: 2
	   statement_1
	   statement_2

4. Continue statement:

   continue statement skips all the remaining code/operation in the current iteration of the loop, and progresses to the next iteration of the loop. It will "continue" operation in the next iteration. In nested loops, the continue statement only works in the body the loop it is stated, not affecting the outer loops.

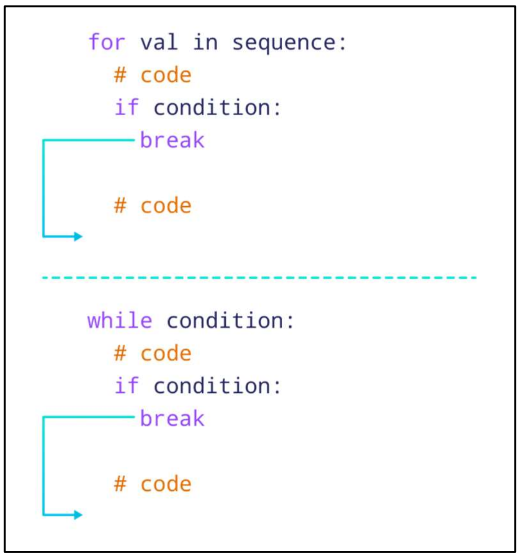


```
for val in sequence:
  # code
  if condition:
    continue

  # code
-----------------------------------
while condition:
  # code
  if condition:
    continue

  # code
```

Flow of continue statement

5. Break statement:

   break statement immediately terminates the loop body when encountered. Code/operation after the break loops is not executed. It will only "break" a loop that contains it. For example, in nested loops, break statement will make the flow exit out of the loop that contains it, not the outer loops.



```
for val in sequence:
  # code
  if condition:
    break

  # code
-----------------------------------
while condition:
  # code
  if condition:
    break

  # code
```

Flow of break statement

6. Vectorizing for loop (meshgrid):

In Python applications, oftentimes large amounts of data need to be processed. Non-optimised functions/loops can slow down operations. This is because the increased computing time and reduce efficiency. Vectorization is a technique that utilizes these standard functions to improve the performance of an algorithm. Vectorization is included in NumPy.

Vectorization is a technique used to improve the performance of Python code by eliminating the use of loops. This feature can significantly reduce the execution time of code. Vectorization stores and manipulates data in an array or vector format rather than as individual units.

When using vectorizing in a for loop, it can speed up the calculations and remove the need for inner loops. In example11, there are nested loops. If we use vectorization, we can remove the inner loop, like in example14.

In example14, vectorization occurs in this line:

`B[:, idx] = a * (1 + r) ** n`

We can break it down to 2 parts: `B[:, idx]` (Left part) & `a * (1 + r) ** n` (Right part)

At right part: the statement calculates the future value of an array of principal amounts. The result is stored in an empty array. The right side uses the formula:

$$B = a \times (1 + r)^n$$

At left part: the code `[:, idx]` stores the calculated value into a matrix. The row and column to store the value is respective to the index called `idx`. For every calculated value, it is store in the column `idx`.

For example:

```
a = np.array([100, 500, 800])
r = 0.09
B = np.zeros((3, 5))
for idx, n in enumerate(range(2, 12, 2)):
    B[:, idx] = a * (1 + r) ** n
```

When idx = 0, n = 2. Vectorization will compute all values in parallel instead of calculating each value one by one like in a loop. This calculates the future values for all elements in a in one step. Which means:

```
B[:, 0] = [100 * (1 + 0.09) ** 2, 500 * (1 + 0.09) ** 2, 800 * (1 + 0.09) ** 2]
```

Resulting in:

```
B =
[[118.81 0. 0. 0. 0. ]
[594.05 0. 0. 0. 0. ]
[950.48 0. 0. 0. 0. ]]
```

The process will continue for every iteration. And with each iteration, vectorization will compute values in parallel, and the matrix B will fill up with values.

When calculating large amounts of data, for loops can be slow. Opting for arrays and vectorized operations can save time and increase efficiency.

The example above can be calculated using NumPy's np.meshgrid function, vectors a and n are expanded into two-dimensional arrays A and N. These arrays are then directly used in calculations to be stored in 2D array B.

meshgrid function:

It generates a matrix or "grid" from 1D arrays to 2D arrays. If there are two lists, one for x values and another for y values, meshgrid combines them to make a grid of all possible (x, y) points.

This makes it easier for future calculations it allows user to perform vectorized operations across a matrix/grid of points efficiently.

For example:

Code	Output

`a = np.array([100, 500, 800])`
`n = np.array([2, 4, 6, 8, 10])`
`r = 0.09`
 	
`N, A = np.meshgrid(n, a)`
`B = A * (1 + r) ** N`
	`Array A:`
	`[[100 100 100 100 100]`
	` [500 500 500 500 500]`
	` [800 800 800 800 800]]`
	
	`Array N:`
	`[[ 2  4  6  8 10]`
	` [ 2  4  6  8 10]`
	` [ 2  4  6  8 10]]`
	
	`Array B:`
	`[[ 118.81    141.15   167.71   199.26   236.74]`
	` [ 594.05    705.79   838.55   996.28   1183.68]`
	` [ 950.48    1129.27  1341.68  1594.05  1893.89]]`

7. Working of for idx, n in enumerate(range(2, 12, 2)) :

   Breakdown of the code helps understanding of the workings.

   - Firstly, (range(2, 12, 2): This will generate a series of numbers. The numbers will start from 2, until it reaches 12 (exclusive) with addition of 2 each time. Each number have an index, starting from 0.

Output	2, 4, 6, 8, 10
Index	0, 1, 2, 3, 4

   - Then, enumerate() function: This will add a counter (index) to the range. It will generate a pair of index and value (index, value) for each item in the range. In this case, the index is named idx. And value is named n.
   - idx will start at 0, increment by 1. n is the current element inside the range.
   - This code will generate a series of numbers [2,4,6,8,10] and create the index, idx with value 0. For every n in [2,4,6,8,10], n will be used in the loop.

8. Advantages of using array operation method compared to loop:

Advantages of array	
Array operation	Loop operation
Faster execution time	Slow execution time
Parallel execution	Cannot compute in parallel
Optimized for mathematical functions	Not optimised for mathematical and large amount of data

References:

Python round() Function

https://www.w3schools.com/python/ref_func_round.asp

Nested loop

https://www.geeksforgeeks.org/python-nested-loops/

Python break and continue statement

https://www.programiz.com/python-programming/break-continue

Vectorization

https://medium.com/pythoneers/vectorization-in-python-an-alternative-to-python-loops-2728d6d7cd3e

https://www.youtube.com/watch?v=YzFT3C8PsC4

Enumerate function

https://www.geeksforgeeks.org/enumerate-in-python/

meshgrid function

https://www.geeksforgeeks.org/numpy-meshgrid-function/