Looping

Control flow aims to provide a structure/order to programming assists with keeping two key principles

1. DRY - Don't Repeat Yourself

   Aims at reducing the repetition of patterns and code duplication

2. KISS - Keep It Simple Stupid

   It states that most systems work best if they are kept simple rather than made overly complicated.

There are 2 ways we can do looping in Python

1. While loop (aka `indefinite iteration`
2. For loop (aka `definite iteration`)

General structure of loop

- Condition-controlled loop : The loop continues until the given condition is evaluated as `False`
- Collection-controlled loop: The loop continues to execute against each element in the collection (sequence).

While loop

Structure of while loop

```
counter = 0    # required only when it is "condition-controlled loop"

while <condition>:   # start of while loop
    <statement_1>    # body of while loop
    <statement_1>
    ...
    ...
    <statement_1>    # end of while loop (indentation ends here)
else:
    <statement :: whenever condition in while is evaluated as `False`>
```

Example : Fibonacci series

```
# method 1


"""
 0,     1,     1,     2,      3,      5,      8
 prev   curr                                        # first iteration
        prev   curr                                 # second iteration
               prev   curr                          # third iteration
                       prev   curr                  # forth iteration

NOTE ::
1. The order of statements in the while "body" is VERY important.
2. We calculate "future" and assign it to "curr" for next iteration.

"""

prev = 0
curr = 1

while curr < 100:
    future = curr + prev
    print(future)
    prev = curr
    curr = future
```

Above code snippet can also be written as -

```
prev = 0
curr = 0

while curr < 100:
    print(curr)
    prev, curr = curr, prev + curr
```

Rules of else block in looping (applies both while and for loop)

- `else` block executes only when loop exits normally
- `else` block executes only when there is no break statement in loop

Example -

```
secret_password = "aeiou@123"
valid_ = ""

while valid_ != secret_password:
    valid_ = input("What is the secret password?")
else:
    print("You have logged in successfully.")
```

Break and continue statement

- `break` statement breaks the loop
- `continue` statement skips current iteration

Game of Sticks

This is a very simple game to demonstrate use of `break` and `continue` statement.

```
"""
There are 21 sticks.
There are 2 kinds of users -
    1. human user
    2. computer

When it's human's turn, human user picks sticks between (1-4).
In next turn, computer picks sticks between (1-4).
Whoever picks last stick looses the game.

"""

sticks = 21

while True:
    print(f"sticks available - {sticks}")
    sticks_taken = int(input("please pick sticks (1-4) - "))

    if sticks == 1:
        print("[ LOST ] You have taken last stick, you are looser!!!")
        break

    elif sticks_taken >= 5 or sticks_taken <= 0:
        print("[ ERROR ] wrong choice. Please pick in the range (1-4). Please choose again -")
        continue

    print(f"computer picks - {5 - sticks_taken}\n")
    sticks -= 5
```

Nested while loop

Example to print first 10 arithmetic tables -

```
i = 1
while i < 11:

    n = 1
    while n < 11:
        print(f"\t{i * n}", end = "")
        n += 1

    print()
    i += 1
```

For loop

- This type of loop is used for “collection-controlled loop”.
- list, string, tuple are some examples of sequences (collections of elements) on which for loop can be executed.
- Any “iterable” object can be used in for loop. In fact, definition of `iterable` goes like following -

> *An `iterable` is any Python object capable of returning its members one at a time, permitting it to be iterated over in a for-loop.*

Syntax

```
for <itarator variable> in <sequence>:
    <statement1>
    <statement2>
    ...
    ...
    <statementn>
else:
    print("whatever you want to execute when sequence is fully iterated")
```

Let’s see 2 types of examples -

- **Using sequence**
- **Using iterator object**

Example using a sequence,

```
languages = ["python", "java", "ruby", "perl", "cpp", "c#"]

for lang in languages:
    print(lang)
else:
    print("\nDONE")
```

Example using iterator object,

```
for i in range(1, 11):
    print(i)
else:
    print("Final message!")
```

In above example, `range()` is an in-built function

- Function can take 3 input arguments: `start`, `end`, `step` (similar concept as is in slicing)

- `start` is inclusive
- `end` is exclusive
- `step` is number of hops
- `range()` function by default returns `range` object which is an `iterable`
- `iterable` is a type of object that can be iterated using `for` loop

`enumerate()` function

- The enumerate() function in Python takes in a data collection as a parameter and returns an enumerate object.
- The enumerate object is returned in a key-value pair format. The key is the corresponding index of each item and the value is the items.
- We can loop over sequence of items with the index using `enumerate()`

```
groceries = ["bananas","butter","cheese","toothpaste"]

for index, grocery in enumerate(groceries):
    print(index, grocery)
```

Here is the syntax of the `enumerate()` function and its parameters:

```
>>> enumerate(iterable, start)

# The enumerate() function takes in two parameters: iterable and start.

# `iterable` is the data collection passed in to be returned as an enumerate object.
# start is the starting index for the enumerate object. The default value is 0.
```

Example of `enumerate()` function with non-default value for parameter `start`

```
>>> names = ["Rahul", "Hardik", "Virat"]
>>> enum_names = enumerate(names, 10)

>>> print(list(enum_names))

[(10, 'Rahul'), (11, 'Hardik'), (12, 'Virat')]
```

Nested for loop

```
india = ["virat", "chahal", "rohit", "hardik"]
aus = ["zampa", "maxwell", "finch"]

for indian_player in india:

    print(f"** {indian_player} goes for handshake **\n")

    for aus_player in aus:
        print(f"{indian_player} handshakes {aus_player}")

    index_ = india.index(indian_player)
    print(f"** remaining players - {india[index_+1:]} **\n")
```

`break` and `continue` in for loop

- `break` and `continue` statements work exactly similar way in `for` loop as they do in `while` loop.

`break` example -

```
foo = ["apple", "banana", "mango"]

for i in foo:
    print(i)
    if i == "banana":
        break
else:
    print("Final message")  # This message will NEVER be printed as we use `break`
```

`continue` example -

```
foo = ["apple", "banana", "mango"]

for i in foo:
    if i == "banana":
        continue        # "banana" will NEVER be printed as we skip iteration here
    print(i)
```

Fibonacci series using `for` loop

```
limit = int(input("How many elements do you want? : "))
prev = 0  # first element of series
curr = 1  # second element of series

if limit <= 0:
    print("The requested series is", curr)
else:
    print(prev, curr, end=" ")
    for i in range(2, limit):
        future = prev + curr
        print(future, end=" ")
        prev = curr
        curr = future
```

List comprehension

- `List comprehension` is an elegant way of converting multi-line for loop into a single line code.
- Result of `list comprehension` is always a new list object.
- To replace a for loop with `list comprehension` the body of for loop needs to be either a single expression or should be bunched together in a `function`.

Syntax

```
[<expression> for <iterator_variable> in <sequence>]

# Optionally, you can also add a "condition" (clause) at the end

[<expression> for <iterator_variable> in <sequence> if <condition>]

# The "expression" can also be a "ternary operator"
```

Example,

```
foo = [1, 2, 3, 4, 5, 6, 7, 8, 9]
for i in foo:
    print(f"{i*i}")

# Above code can be changed to following code

>>>
>>> [i*i for i in foo]
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

How to find odd numbers between (1-10) using list comprehension?

```
>>> [i for i in range(1, 11) if i % 2 != 0]

[1, 3, 5, 7, 9]
```

Ternary operator

Ternary operator is nothing but single line if-else statement.

Syntax -

```
<expression 1> if <condition> else <expression 2>
```

When condition is evaluated as `True`, final result comes from `expression 1`

When condition is evaluated as `False`, final result comes from `expression 2`

Example,

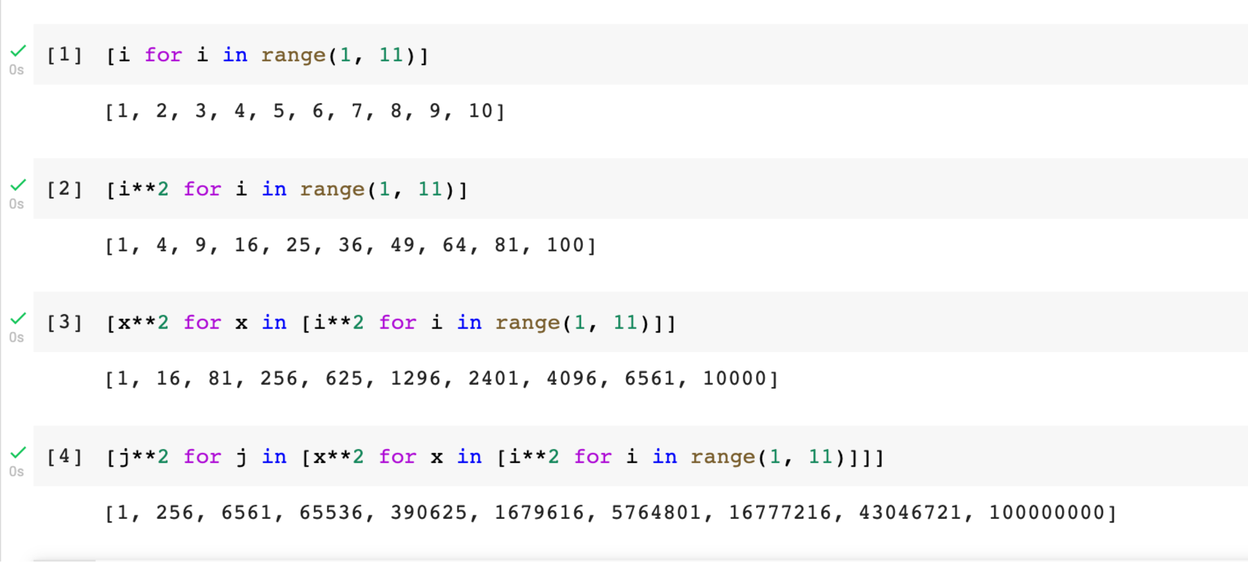
```
>>> [i for i in range(1, 5)]

[1, 2, 3, 4]

# Above code with odd even results using list comprehension
# Where, expression is a ternary operator can be written as following -

>>> [f"{i} (even)" if i % 2 == 0 else f"{i} (odd)" for i in range(1, 5)]
```

```
['1 (odd)', '2 (even)', '3 (odd)', '4 (even)']
```

Nested `list-comprehensions`

We can go on using result of any `list comprehension` (which a `list`) as input for another.

In below example, we are performing square of each element in sub-list returned by another `list comprehension`.



```
✓ 0s  [1]  [i for i in range(1, 11)]

           [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

✓ 0s  [2]  [i**2 for i in range(1, 11)]

           [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

✓ 0s  [3]  [x**2 for x in [i**2 for i in range(1, 11)]]

           [1, 16, 81, 256, 625, 1296, 2401, 4096, 6561, 10000]

✓ 0s  [4]  [j**2 for j in [x**2 for x in [i**2 for i in range(1, 11)]]]

           [1, 256, 6561, 65536, 390625, 1679616, 5764801, 16777216, 43046721, 100000000]
```

In final statement, we have 3 for loops nested.

Iterating on `dict` object

```
# dict function `keys()` returns an iterable
>>>
>>> batter_n_scores = {"virat": 100, "rohit": 200, "hardik": 50, "surya": 0}
>>> batters = batter_n_scores.keys()
>>> batters
dict_keys(['virat', 'rohit', 'hardik', 'surya'])
```

```
# similarly, dict function `values()` returns an iterable
>>>
>>> scores = batter_n_scores.values()
>>> scores
dict_values([100, 200, 50, 0])
```

```
# There is a dict function called `items`.
# You can inspect `dict` attributes using`dir()` function on dict.

>>> dir(dict)
>>> 'items' in dir(dict)
True
```

Iterating using `items()` function on `dict` object

```
batter_n_scores = {"virat": 100, "rohit": 200, "hardik": 50, "surya": 0}

for batter, score in batter_n_scores.items():
    print(batter, score)
```

`dict` comprehension

The syntax for `dict` comprehension is similar to `list comprehension`. The only difference is we are iterating on pair of elements in case of `dict` (as `dict` is nothing but collection of `key: val` pair)

Let's swap keys <---> values
Meaning, keys will be values and values will be keys

```
batter_n_scores = {"virat": 100, "rohit": 200, "hardik": 50, "surya": 0}

after_swap = dict()

for batter, score in batter_n_scores.items():
    after_swap[score] = batter
```

```
print(after_swap)
```

Above code using `dict` comprehension can be -

```
{score: batter for batter, score in batter_n_scores.items()}
```

Let’s take another example to evaluate even numbers (adding a condition at the end)

```
>>> # syntax :: {<key_>: <val_> for <iterator_variable> in <sequence> if <expression>}
>>>
>>> {i: "even" for i in range(1, 7) if i % 2 == 0}
{2: 'even', 4: 'even', 6: 'even'}
```

Let’s take another example to evaluate even numbers and odd numbers

```
>>> # syntax :: {<ternary operator expression> for <iterator_variable> in <sequence>}
>>> # where ternary operator expression,
>>> # <key_>: <val_> if <expression_1> else <expression_2>
>>>
>>> {i: "even" if i % 2 == 0 else "odd" for i in range(1, 7)}
{1: 'odd', 2: 'even', 3: 'odd', 4: 'even', 5: 'odd', 6: 'even'}
```

[TODO] extension for map, filter and reduce

https://gist.github.com/raybuhr/9481077b4c95c80591c6f0736329925a#file-python_loops-md

[Exercise] looping : part 1

[Exercise] Looping : part 2

🎴 Cheat-sheet: how to iterate on structured data types