Lecture 13 – range function, for...in loops
Learning outcomes

At the end of this lecture, students should:

- understand the Python range() function and use it to define a sequence of values
- understand the for...in loop structure used with the range() function
- be able to define a for...in loop to implement counter-controlled repetition
- be able to convert a for...in loop (with a range function) into a while loop and vice versa
The Python range() function defines a sequence of integer values within a boundaries.

The range() function has the following syntax: \texttt{range(start, stop, step)}

where the three arguments are:

- \texttt{start} - the lower bound (included) of the sequence defined,
- \texttt{stop} - the upper bound (excluded) of the sequence defined,
- \texttt{step} - the increment between each number in the sequence defined.

Some examples:

- \texttt{range(1, 10, 2)} defines the sequence 1, 3, 5, 7, 9
- \texttt{range(5, 20, 6)} defines the sequence 5, 11, 17
- \texttt{range(14, 4, -3)} defines the sequence 14, 11, 8, 5
- \texttt{range(0, 7, 1)} defines the sequence 0, 1, 2, 3, 4, 5, 6
range(start, stop, step)

- If the step is omitted, the default step is 1.
  - range(0, 7) defines the sequence 0, 1, 2, 3, 4, 5, 6
  - range(0, 4) defines the sequence 0, 1, 2, 3

- If both the start and the step are omitted, the sequence starts from 0 with a step increment of 1.
  - range(5) defines the sequence 0, 1, 2, 3, 4,
  - range(7) defines the sequence 0, 1, 2, 3, 4, 5, 6

Note that printing a range does NOT print the defined sequence of integers, i.e., print(range(6)) does NOT print the sequence 0, 1, 2, 3, 4, 5
The Python range() function continued

range(start, stop, step)

- The step cannot be 0:
  - range(0, 7, 0) gives an error
  - `ValueError: range() arg 3 must not be zero`

- If the step is negative then the start value must be greater than the stop value.
  - range(14, 4, -3) defines the sequence 14, 11, 8, 5
  - range(4, 14, -3) defines an empty sequence

- If the step is positive then the start value must be smaller than the stop value.
  - range(14, 4, 3) defines an empty sequence
  - range(4, 14, 3) defines the sequence 4, 7, 10, 13
Iteration – for...in loops

- The following while loop executes an exactly 100 times (for count = 0 to count = 99). The variable count controls the number of times the loop body is executed.

```python
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

- The **for...in range(...)** loop can provide a compact structure for counter-controlled type of loops.

```python
for count in range(0, 100):
    print("Programming is fun!")
```

Programming is fun!
Programming is fun!
Programming is fun!
...
Iteration – for...in loops

- Note that in the for...in loop on the previous slide the name used for the loop variable can be any identifier. The following two loops

```python
for value in range(0, 100):
    print("Programming is fun!")
```

```python
for number in range(0, 100):
    print("Programming is fun!")
```

behave in exactly the same way.

- Note that in the for...in loops above, the loop body is executed for each value in the sequence defined by the range() function. In the body of the loop, the loop variable takes each value of the sequence defined by the range() function, e.g.,

```python
for number in range(0, 5):
    print(number)
```

```python
for number in range(3, 7):
    print(number * 5)
```

Programming is fun!
Programming is fun!
Programming is fun!
Programming is fun!

...
Complete the loops

- Complete the for...in loop so that the output is:

```python
for number in [7, 10, 13, 16, 19, 22]:
    print(number, end = " ")
print()
```

- Complete the for...in loop so that the output is:

```python
for value in [30, 25, 20, 15, 10, 5, 0, -5, -10]:
    print(value, end = " ")
print()
```
while loop vs for...in loops

- Counter-controlled while loops can be converted into for...in range(...) loops and vice versa.

```python
count = 0
while count < 100:
    print("Programming is fun!")
    count = count + 1
```

```python
for count in range(0, 100):
    print("Programming is fun!")
```

- Not all while loops can be expressed using a for...in range(...) loop (only the ones for which we know exactly how many times the loop body is to be executed).

- All for...in range(...) loops can be expressed as while loops.
Convert - while loop ↔ for...in loop

- Convert the following while loop into a for...in range(...) loop:

```python
counter = 12
while counter < 260:
    print(counter)
    counter = counter + 10
```

- Convert the following for...in range(...) loop into a while loop:

```python
for num in range(45, 3, -5):
    print(num * 2)
```
Do the following two loops give the same output? If not, what is the difference in output and what is the change which needs to be made if I would like the output to be the same?

```python
top = 6
bottom = 0
count = 0
sum = 0

while bottom < top:
    count += 1
    sum = sum + top + bottom
    bottom = bottom + 2

print("count:", count,
      "sum:", sum)
```

```python
top = 6
count = 0
sum = 0

for bottom in range(0, top + 1, 2):
    count += 1
    sum = sum + top + bottom

print("count:", count,
      "sum:", sum)
```
A perfect number is an integer that is equal to the sum of its divisors (excluding the number itself), e.g., \(28 = 1 + 2 + 4 + 7 + 14\). Complete the `get_divisor_sum()` function using a `for...in range(...) loop` for the iteration.

```python
def get_divisor_sum(number):
    # Add your code here

def check_perfection(number):
    if number == get_divisor_sum(number):
        print(number, "is a perfect number")
    else:
        print(number, "is NOT a perfect number")

def main():
    check_perfection(28)
    check_perfection(54)
    check_perfection(496)

main()
```

28 is a perfect number
54 is NOT a perfect number
496 is a perfect number
def get_divisor_sum(number):
    div_sum = 0
    middle_num = number // 2
    for divisor in range(1, middle_num + 1):
        if number % divisor == 0:
            div_sum += divisor

    return div_sum

def check_perfection(number):
    if number == get_divisor_sum(number):
        print(number, "is a perfect number")
    else:
        print(number, "is NOT a perfect number")

def main():
    check_perfection(28)
    check_perfection(54)
    check_perfection(496)

main()
Which to use, while loop or for...in loop?

Which type of loop should you use?

- A while loop is more general. It can be used to handle repetition of a section of code any number of times and to handle user controlled repetitions, e.g., executing a piece of code a known size or when the number of times the loop is executed depends on the user input (or on a condition which depends on a random number).

- A for...in range(...) loop is more compact and particularly useful for processing a sequence of values one by one.
In a Python program:

- the Python range() function is used to define a sequence of values
- a for...in range(...) loop can be used to implement counter-controlled repetition
- a for...in range(...) loop can be converted into a while loop and vice versa
- a for...in range(...) loop has the following syntax:

```python
for a_variable in range(...):
    statement1
    statement2
    ...
```
Examples of Python features used in this lecture

def get_divisor_sum(number):
    divisor = 1
    div_sum = 0
    middle_num = number // 2
    for i in range(middle_num + 1):
        if number % divisor == 0:
            div_sum += divisor
    return div_sum

def fun_stuff():
    total = 0
    for number in range(9, 20):
        if number % 2 == 0 or number % 3 == 0:
            total += 1
    print(total)