



# 15-110 PRINCIPLES OF COMPUTING – F21

LECTURE 9:

WHILE LOOPS

TEACHER:

GIANNI A. DI CARO

# Repeat for a definite number of iterations: `for` construct

- ✓ Repeat a set of actions a **defined number of times** (at *most*)
- ✓ Each time the action *can* be executed on a different input parameter

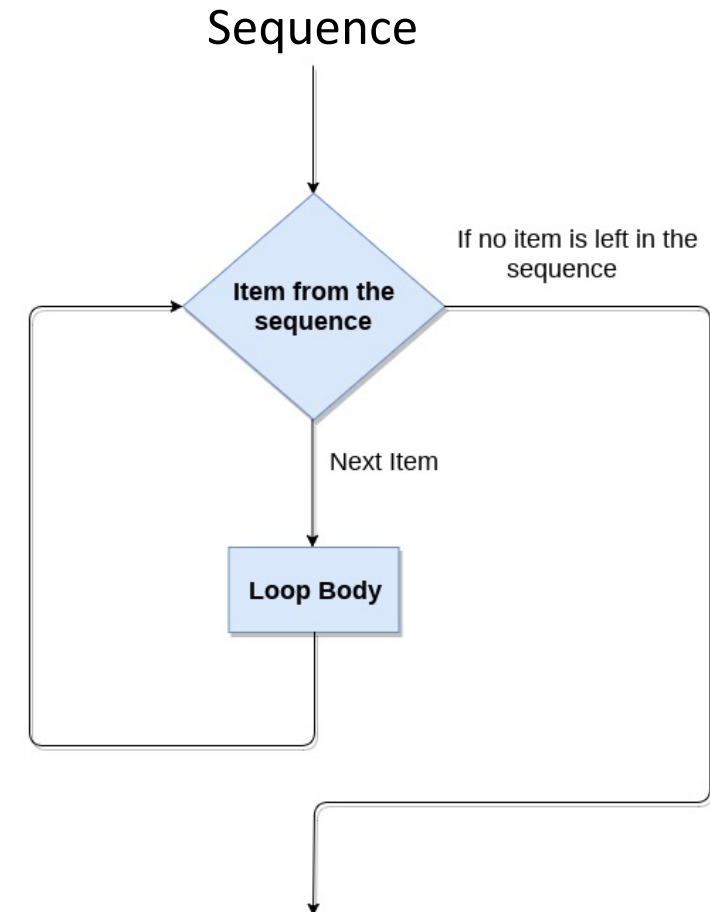
```
def sum_first(n):  
    s = 0  
    for i in range(n+1):  
        s = s + i  
    return s
```

`range(n+1)` → 0, 1, 2, 3, ..., n

The action `s = s + i` is repeated exactly `n+1` times

```
for i in range(1, 13, 3):  
    print(i)
```

How many times the print action will be executed?



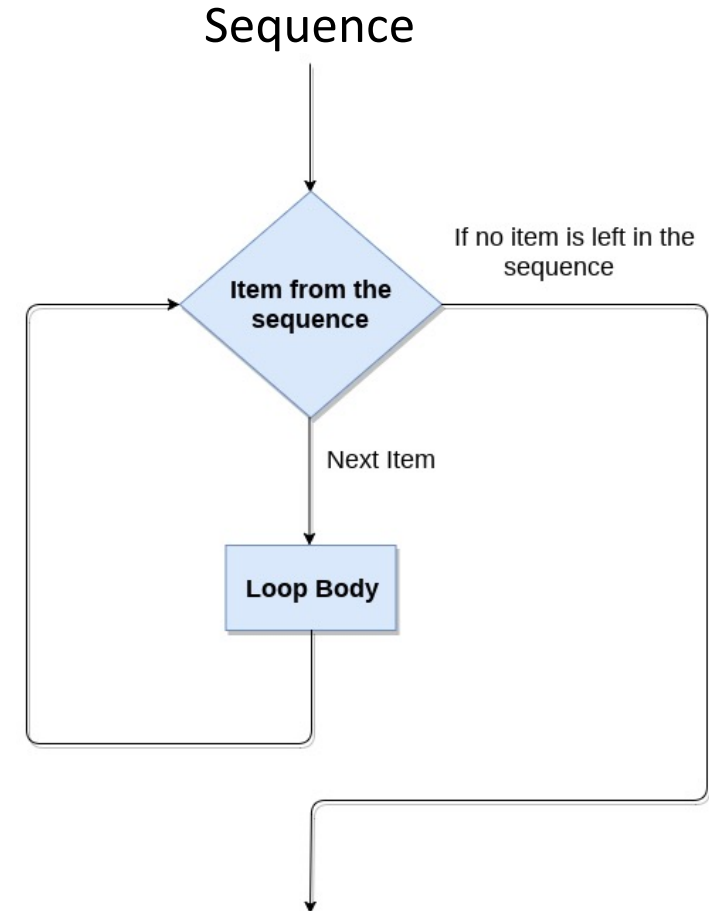
# Repeat for a definite number of iterations: for construct

```
for i in range(1, 13, 3):  
    print(i)
```

How many times the print action will be executed?

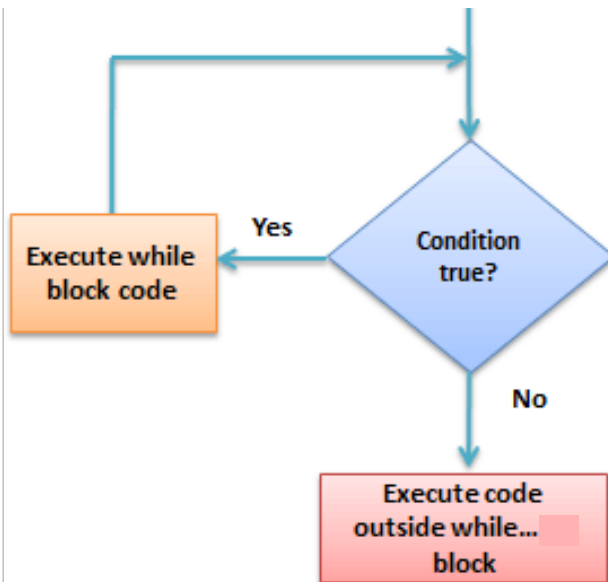
Let's **count** the number of iterations!

```
counter = 0  
for i in range(1, 13, 3):  
    print(i)  
    counter = counter + 1  
print('Number of iterations:', counter)
```



# Repeat for indefinite (or conditional) iterations: `while` loops

- ✓ Repeat a set of actions an **unspecified number of times**: keep doing as far as a given condition is true



```
while condition_is_true:  
    do_something
```

```
def sum_first_10()  
    i = 0  
    s = 0  
    while i <= 10:  
        s = s + i  
        i = i + 1  
    print('Out of while loop!')  
    return s
```

```
def sum_first_10():  
    s = 0  
    for i in range(n+1):  
        s = s + i  
    print('Out of while loop!')  
    return s
```

vs.

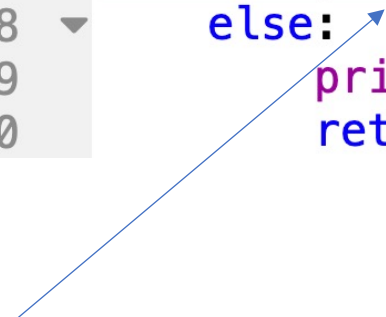
- ✓ `while` loops are more flexible and general than `for` loops, since we are not restricted to iterate over a sequence, but code can be less compact and more prone to errors ...

# while loops unrolled

```
def sum_first_10():  
    i = 0  
    s = 0  
    while i <= 10:  
        s = s + i  
        i = i + 1  
    print('Out of while loop!')  
    return s
```

Equivalent to:

```
1  ▼ def sum_first_10():  
2      i = 0  
3      s = 0  
4  ▼      if i <= 10:  
5          s = s + i  
6  ▼          i = i + 1  
7      go back to line 4  
8  ▼      else:  
9          print('Out of while loop!')  
10         return s
```



This is NOT a legal python instruction!  
(it's just to explain what happens)

# An example: sum up to a maximum value

---

Implement the function `sumUpToMax(max_value)` that incrementally sums up the integer numbers, starting from 1. It stops when the sum gets higher than `max_value`.

For instance, `sumUpToMax(10)` starts at 1 and adds 2, 3, 4. It stops there because  $1 + 2 + 3 + 4$  is 10. In this case, if 5 would be added, the sum would exceed the `max_value` 10.

The function returns the last integer `n` that was used in the sum

```
def sumUpToMax(max_value):  
    sum_n = 0  
    n = 1  
    while sum_n < max_value:  
        sum_n = sum_n + n  
        n = n + 1  
    return n-1
```

Be careful!

- At the exit of the while loop, the variable `n` has been stepped up one extra time
- What is the value of the variable `sum_n` at the end of the loop? Is it always less than `max_value` or not?

# An example: cool the room

---

Implement the function `decreaseTemperature(t, hot, cooling_step)` that decreases the current room temperature `t` stepwise until it reaches a value below or equal to the `hot` threshold. Each cooling step corresponds to a decrease in temperature of `cooling_step` degrees.

The function returns the new temperature and prints out the number of cooling steps.

```
def decreaseTemperature(t, hot, cooling_step):  
    step_num = 0  
    while t > hot:  
        t = t - cooling_step  
        step_num = step_num + 1  
    print('Cooling steps: ', step_num)  
    return t
```

# Never happening loops

---

✓ Summing up starting from 0

```
n = 1
sum_n = 0
while sum_n < 10:
    sum_n = sum_n + n
    n = n + 1
print('n:', n-1)
```

```
def decreaseTemperature(t, hot, cooling_step):
    step_num = 0
    while t > hot:
        t = t - cooling_step
        step_num = step_num + 1
    print('Cooling steps: ', step_num)
    return t
```

Summing up starting from an arbitrary value

```
n = 1
sum_n = 10
while sum_n < 10:
    sum_n = sum_n + n
    n = n + 1
print('sum:', sum_n, 'n:', n-1)
```

```
decreaseTemperature(22, 28, 2)
```

The body of the loop is never executed!



# Never ending loops

---

```
n = 0
sum_n = 5
while sum_n < 10:
    sum_n = sum_n + n
    n = n - 1
```

Condition `sum_n < 10` is ALWAYS satisfied,  
the loop will never end

**Don't run these codes! → Or run them and then interrupt the kernel (in Consoles)**

```
n = 0
sum_n = 0
while True:
    sum_n = sum_n + n
    n = n + 1
```

Condition `True` is ALWAYS  
satisfied, the loop will never end

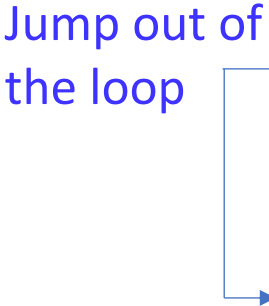
Condition `False` is NEVER  
satisfied, the loop won't start!

```
n = 0
sum_n = 0
while False:
    sum_n = sum_n + n
    n = n + 1
```

# Interrupted (infinite) loops: break and return

Make the sum of first n integers until the sum reaches a value greater than 12

```
n = 0
n = 0
sum_n = 0
while True:
    if sum_n + n > 12:
        break
    else:
        sum_n = sum_n + n
        n = n + 1
print('Sum is:', sum_n)
print('n is:', n-1)
```

A blue arrow points from the text "Jump out of the loop" to the 'break' statement in the code. The text is positioned to the left of the code block, and the arrow originates from the text and points to the 'break' statement within the 'if' condition of the 'while' loop.

Equivalent to:

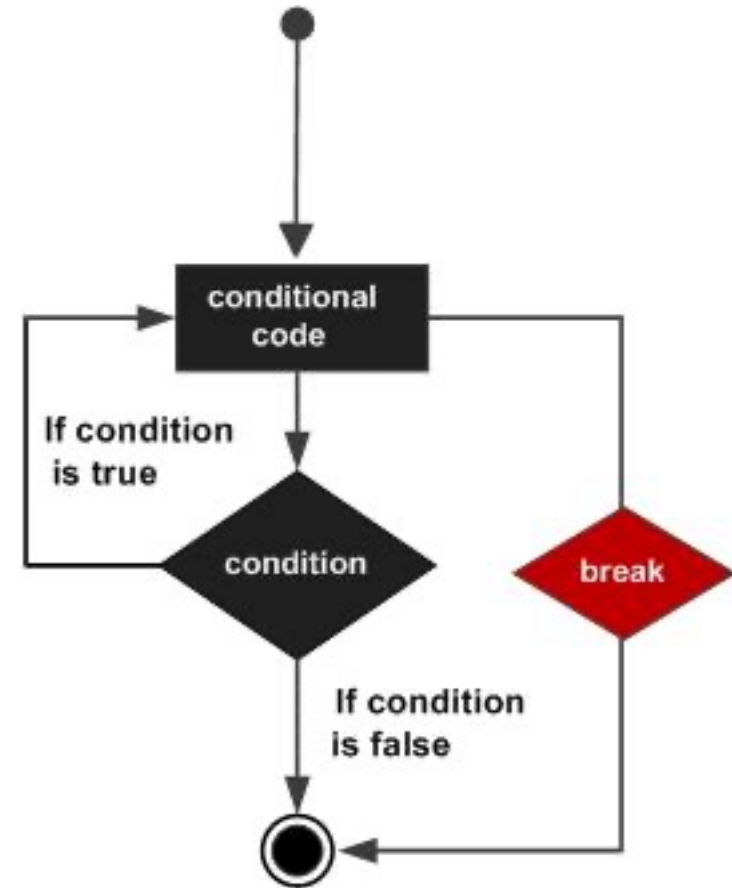
```
n = 0
sum_n = 0
while sum_n <= 12:
    sum_n = sum_n + n
    n = n + 1
print('Sum is:', sum_n - (n-1))
print('n is:', n-2)
```

# Interrupted (infinite) loops: break and return

Make the sum of first n integers until the sum reaches a value greater than 12

```
n = 0
n = 0
sum_n = 0
while True:
    if sum_n + n > 12:
        break
    else:
        sum_n = sum_n + n
        n = n + 1
print('Sum is:', sum_n)
print('n is:', n-1)
```

Jump out of  
the loop



# Interrupted (infinite) loops: break and return

---

- Inside a **function**, `return` exits the loop (and the function!)

```
def sum_return(max_val):  
    n = 0  
    sum_n = 0  
    while True:  
        if sum_n + n > 12:  
            return sum_n, n-1  
        else:  
            sum_n = sum_n + n  
            n = n + 1
```

# Practice: number of digits

---

Implement the function `numberOfDigits(n)` that returns the number of digits in the input integer `n`

E.g., `numberOfDigits(15110)` should return 5.

```
def numberOfDigits(n):  
    d = 0  
    while n > 0:  
        n = n // 10;  
        d = d + 1  
  
    return d
```

```
def numberOfDigits(n):  
    p = 0  
    while (n % (10 ** p)) != n:  
        p = p + 1  
  
    return p
```

# Practice: Population increase

---

You are studying two populations A and B. You know that population A is initially smaller than population B, but it grows at a faster rate. You would like to know how many days it would take for population A to overtake population B. Luckily you have taken 15-110, and you know it would be faster to sit down and implement a program to compute that for you, than having to do the math each time (you work with a lot of populations of many different organisms...).

Implement the function `populationIncrease(pa, pb, ga, gb)` that takes as parameters:

- The number `pa` of individuals in population A
- The number `pb` of individuals in population B
- The growth rate `ga` of population A (in percent per day)
- The growth rate `gb` of population B (in percent per day)

This function should return the number of days it takes for population A to overtake population B.

**Important:** Populations grow by an integer number of individuals. So if the growth rate is 3.6% and the population is 100, in one day there will be 103 (not 103.6) individuals. If the population is 1000, there will be 1036 individuals.

## Practice: Population increase

---

```
import math
def populationIncrease(pa, pb, ga, gb):
    return 42

# assert(populationIncrease(100, 150, 1.0, 0) == 51)

# assert(populationIncrease(90000, 120000, 5.5, 3.5) == 16)
# assert(populationIncrease(56700, 72000, 5.2, 3.0) == 12)
# assert(populationIncrease(123, 2000, 3.0, 2.0) == 300)
# assert(populationIncrease(100000, 110000, 1.5, 0.5) == 10)
# assert(populationIncrease(62422, 484317, 3.1, 1.0) == 100)
```

# Practice: Nearest factorial

---

Implement the `nearestFactorial(n)` function that returns the integer number whose factorial value is the nearest to `n` while not being greater than `n`.

For instance, `nearestFactorial(25)` returns 4. In fact, since  $4! = 1 \times 2 \times 3 \times 4 = 24$ , while  $5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$ , and  $24 \leq 25$ .



# An example: sum up to a maximum value with a start

---

Implement the function `sumUpToMaxWithStart(start, max_value)` that, starting from the integer value `start`, incrementally sums up the integer numbers. It stops when the sum gets higher than `max_value`.

For instance, `sumUpToMaxWithStart(2, 10)` starts at 2 and adds 3 and 4. It stops there because  $2 + 3 + 4$  is 9. In this case, if 5 would be added, the sum would exceed the `max_value` 10.

The function returns the last integer used in the sum.

If `start > max_value`, 0 is returned.

```
def sumUpToMaxWithStart(start, max_value):  
    if start > max_value:  
        return 0  
    sum_n = 0  
    n = start  
    while sum_n < max_value:  
        sum_n = sum_n + n  
        n = n + 1  
    return ???
```